Getting Started with TRANSACT/V

HP 3000 MPE/iX Computer Systems

Edition 2

Manufacturing Part Number: 32247-90007
E0788

U.S.A. July 1988

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Acknowledgments

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Contents

Contents

Figures

Figures

Figures

Figures

Figures

Figures

Preface

This manual is intended to be used as a supplement to the *Transact Reference Manual*, part number 32247-90001. It illustrates many of the features of Transact through programming examples, and is designed to serve as a task-oriented learning aid.

Readers are expected to be programmers who have a working knowledge of at least one programming language. They should also understand the basic concepts of the IMAGE database system, Dictionary/V and VPLUS.

1 Getting Started

This document is designed to give the novice Transact user a easy-to-use starting point from which to quickly gain an understanding of the language. Beginning with a simple three-line program to display the data in a dataset, we move on to introduce many of the important concepts of Transact, always with simple, yet complete, programs.

Later sections point up some of the ways Transact can be most beneficially used to increase programming efficiency.

The examples throughout this guide are all based on a single Image database, three MPE data files, and a single VPLUS forms file. In other words, they represent a mini working environment.

The database, files, and forms file have all be described in Dictionary/V. The appendices provide complete examples of how to use the dictionary to describe the database, forms, and files used throughout the guide.

Compiling and Executing Transact Programs

Transact programs are written using an editor such as EDIT/V or TDP. The source is then compiled using TRANCOMP and the resulting program is executed using TRANSACT.

If you are using DICTIONARY/V to hold the definition of your database, TRANCOMP expects the dictionary to be called DICT and to reside in the PUB group. However, you can override this by issuing a file equate before running TRANCOMP, such as:

```
FILE DICT.PUB=dictname[.group]
```

Figure 1-1 shows the steps used to compile and execute a TRANSACT program. After the discussion of compiling and executing programs in complete, the same program used in Figure 1-1 is displayed in Figure 1-2, where it is accompanied by a line-by-line explanation.

Figure 1-1. Compiling and Executing a TRANSACT Program

```
1  :file dict.pub=dict
2  :run trancomp.pub.sys

3  TRANSACT/3000 COMPILER   HP32247A.02.02 - (C) Hewlett-Packard Co. 1984

4  SOURCE FILE> ex1

5  LIST FILE>

6  CONTROL>
7  TRANSACT/3000 COMPILER  A.02.02 : TUE, MAR 19, 1985, 11:07 AM COMPILED
8  LISTING OF FILE EX1.HOWTO.MILLER                  PAGE 1

9  COMPILING WITH OPTIONS: LIST,CODE,DICT,ERRS

10     1.000           system ex1,base=orders;
11     2.000  0000     list(auto) customer;
12     3.000  0005     output(serial) customer;

13 CODE FILE STATUS: NEW

14 0 COMPILATION ERRORS
15 PROCESSOR TIME=00:00:04
16  ELAPSED TIME=00:00:09

17 END OF PROGRAM
18 :run transact.pub.sys

19 TRANSACT/3000    HP32247A.02.02 - (C) Hewlett-Packard Co. 1984

20 SYSTEM NAME> ex1

21  PASSWORD FOR ORDERS>
```

Let's look at what is happening in the above example on a line by line basis.

1 By default, the compiler TRANCOMP expects the user dictionary to be in the PUB group of the logon account. We are changing the default to be the logon group by issuing a file equation.

2 We run the Transact compiler to transform the source input into intermediate code which can then be executed using the processor (see line 18 below).

4 The compiler asks for the name of the file containing the source code to be compiled. The source code was previously entered by using a text editor such as EDIT or TDP.

5 By default, the compiler listing goes to $STDLIST (the terminal in this case). To override the default, we could have entered the name of a file or directed the output to the printer.

6 The compiler has many options which can be entered at this prompt. These options control, for example, whether an object file is created, the format of the listing generated, ways to optimize code generated so as to conserve resources, such as stack usage, and so forth.

No options are required. Quite often, you will probably not want to see the compiled listing. You can keep the listing from printing by responding NOLIST.

9 Since we entered no options, TRANCOMP uses the default options, which are shown here. LIST means that a listing will be generated, CODE calls for the generation of an object or code file, DICT specifies that a dictionary is to be used to resolve file and element definitions, and ERRS means that we want to see a listing of any compiler errors.

10 Lines 10 through 12 are the compiler listing. In this example, the listing consists of the source line number, an internal location reference number that is used when test modes are in effect to debug a program, and the source text.

18 At this point we run TRANSACT. The compiler does not create a true object or program file like that produced by COBOL or PASCAL. It creates what is called an intermediate processor code file. TRANSACT interprets the contents of this file and performs the functionality of our source program.

20 The processor asks for the name used in the SYSTEM statement of the source program. It uses this name to figure out the name of the MPE file that contains the intermediate processor code. The MPE file name is always in the format IPname where name is the name used in the SYSTEM statement. For this system, the name is IPEX1.

Reporting From a Dataset

The first program we look at is a simple, three-line program to report the contents of a dataset. Then we will expand this program and give it many more capabilities as the section progresses. However, this should not be viewed as implying that Transact is a report writer. Transact's power lies in several areas: it provides a high-level interface for database and file access, a high level interface with VPLUS for online database updating, and an automatic error handling facility that makes prototyping of a system possible in a very short time frame.

In addition, when Transact is interfaced with Report/V, it is possible to write complex reports that no nonprocedural report writer can do. The procedural power of Transact is used to retrieve and manipulate the data and then Report is used to format the data and provide summarization as necessary. We will see an example of that later on.

Getting a Complete Listing

One of the simplest tasks to perform with TRANSACT is to display the contents of a data set. The three-line program shown in Figure 1-2 lists the contents of a data set called customer.

Figure 1-2. Program to Display a Dataset

```
1     system ex1,base=orders;
2     list(auto) customer;
3     output(serial) customer;
```

1 The first statement in a Transact program is always a SYSTEM statement, in which we give a name to the current program and identify the database (and other files) to be used by the program. In this example, the program is called EX1. This is the name we will provide to Transact when we want to run the program. We also tell Transact that our Image database is named ORDERS.

2 The Image dataset that we want to list is called customer. We don't need to tell Transact what data elements are in the dataset. LIST(AUTO) instructs TRANCOMP to go to the dictionary and extract the names of all the data elements for the named dataset. Appendix G lists the data element or item content of each dataset and file used throughout this manual, and also displays a diagram that shows the relationship of the sets.

In this example, the LIST(AUTO) statement is equivalent to individually listing each of the data elements as in:

```
list cust-no:
       name:
```

```
                    street-addr:
                    city-state:
                    zipcode;
```

A statement is terminated with a semicolon. Within the statement, item names are separated by colons.

3 The OUTPUT verb sets up a data retrieval and reporting loop; the (SERIAL) option specifies that the customer set be read serially and that each item of each record be retrieved and reported.

The output from this program might look like the report below.

Figure 1-3. Report from a Single Dataset

```
CUST-NO: NAME:               STREET-ADDR:         CITY-STATE:       ZIPCODE:
1        Able-1 Answering    2775 Park Av         San Jose, Ca        95111
2        Grand Depression    27 E Main            Santa Clara, Ca     95122
3        Rummage Palace      410 N 10th           South Bend, Ind.    49146
4        Victorian Antiques  476 S 1st            San Francisco, Ca.  94123
5        Vinicator Corp      1092 Steward Drive   San Jose, Ca        95144
7        Frank Leary Racing  590 Laurelwood       Mountain View, Ca.  92123
8        Professor Muldoon's 123 Main Street      Balloon City, Md    12465
9        Bayliner Boats      1548 Maple           San Jose, Ca.       95144
20       Natkin & Co         807 Aldo Av          Redwood City, Ca.   93144

CONTINUE(Y/N)>  N
```

Transact recognizes that our output is going to a terminal screen which can hold 24 lines, each 80 characters long. Since the output is wider then 80 characters, the last column of the report is on a second line.

If we had directed Transact to send our output to a 132 column printer, then all the data would appear on a single line. After a full screen of the report is displayed on the terminal, Transact pauses, waiting for us to tell it to continue. If we direct output to a printer, the report is generated without any pauses between pages. Also, the report page length is adjusted to the length of a page on the printer.

From the above example, we can see that Transact provides a simple format by default. First, Transact reserves a column for each of the data elements in our report. It makes the column either the width of the name of the data element or the actual element length, whichever is longer. It adds one blank space between the columns. It makes up column headings by using the names of the data elements.

Later on we will see examples of how we can control the format ourselves.

Sorting the Data

Let's change the report to be all elements except cust-no so that it will fit on one line. Also, we will sort the report by city-state.

Figure 1-4. Program to Sort and Report Data

```
1     system ex2,base=orders;
2     list name:
           street-addr:
           city-state:
           zipcode;
3     output(serial) customer,sort=(city-state);
```

2 Since we do not want access to all items from the customer set, we must individually list the items that we do want.

3 Appending the SORT= phrase to the OUTPUT verb tells Transact to sort the data in ascending sequence by the city-state item.

By default, the OUTPUT verb retrieves and reports all items that are LISTed to the program.

An alternative way to do the report is:

Figure 1-5. Program to Sort Data and Use FORMAT for Reporting

```
1     system ex2a,base=orders;
2     list (auto) customer:
3     format name:
             street-addr:
             city-state:
             zipcode;
4     output(serial) customer,sort=(city-state);
```

3 Rather than specify through the LIST verb that a subset of the customer items is to be retrieved, all the items are retrieved, but the FORMAT verb specifies to OUTPUT that only the named subset is to be reported.

Either of these programs produces a report that looks like the one below:

Figure 1-6. The Sorted Report on Customers

```
NAME:               STREET-ADDR:          CITY-STATE:          ZIPCODE:
Hold A Hill Planter 651 El Camino         Dallas, Texas        45623
Balcon's Bridal     1775 Capitol Express  Fresno, Ca.          98167
Hobie Cat           3410 Monterey Rd      Gilroy, Ca.          96144
Furtado Inports     1396 E Santa Clara    Los Angeles, Ca.     90189
Cobalt Boats        2250 San Ramon        Louisville, Ky.      33246
Frank Leary Racing  590 Laurelwood        Mountain View, Ca.   92123
Excl Chemical Co    630 Walsh             New York, NY         44636
Bay Repro           123 Hospital Dr       Palo Alto, Ca.       94967
Natkin & Co         807 Aldo Av           Redwood City, Ca.    93144
Drama Books         511 Geary St.         Redwood City, Ca.    92143
Victorian Antiques  476 S 1st             San Francisco, Ca.   94123
Able-1 Answering    2775 Park Av          San Jose, Ca         95111
Vinicator Corp      1092 Steward Drive    San Jose, Ca         95144
Bayliner Boats      1548 Maple            San Jose, Ca.        95144
Pour House          1475 Lipton Place     San Jose, Ca.        95122
Grand Depression    27 E Main             Santa Clara, Ca      95122
Saxon's             518 W San Carlos      Santa Clara, Ca.     94168
Rummage Palace      410 N 10th            South Bend, Ind.     49146

19 RECORDS FOUND

EXIT/RESTART(E/R)?>
```

Formatting Options

The FORMAT verb also provides extensive reporting options for such things as field editing, heading text, line breaks, column positioning, etc.

The following program illustrates a few of the options available on the FORMAT verb.

Figure 1-7. Options for FORMAT

```
 1 system ex2b,base=orders;
 2 list(auto) customer;
 3 format $today,edit="3w. 3m DD, YYYY":
 4        "LIST OF CUSTOMERS REPORT",col=30:
 5        "PAGE",col=60:
 6        $page:
 7        "CUSTOMER",line=2,title:
 8        name,nohead,line=2:
 9        street-addr,line,nohead:
10        city-state,line,nohead:
11        zipcode,nohead,join=1;
12 output(serial) customer,sort=(city-state);
```

3 $TODAY is a special name that means print today's date. The edit options specify to print the first three characters of the day of the week followed by a period, then print the first three characters of the month, then the numeric day of the month, and last the year.

4 Any literal to be displayed is placed in quotation marks. The COL= option indicates an absolute report column number for the start of the display of an item. In the example the literal value begins in column 30.

6 $PAGE is a special name that means print the current page number here.

7 The LINE= option indicates the number of lines to advance before displaying the item that the option is attached to. In the example, the line count is advanced by two lines. The TITLE option indicates that all report definitions that precede this, including the current report item, make up a report heading or title. These items appear at the top of each new page.

8 The NOHEAD option specifies that no column heading is to be generated for this item. We have decided to provide our own column heading called CUSTOMER.

Note that the LINE= option is used here to indicate that the report is to advance two lines before printing the customer name.

9 This line prints the value of the street address immediately under the name and suppresses the generation of a column heading.

10 Line 10 prints rint the city and state immediately under the street address and suppresses generation of a column heading.

11 The JOIN= option specifies that this item is to be joined to the previous item by leaving only one blank space between the two items. For example, the data item CITY-STATE is 20 bytes long. It only takes 13 bytes to store the value “San Jose, Ca.”. Thus there are 7 trailing blanks in the CITY-STATE item for this value. JOIN=1 specifies that only one blank should appear between the last nonblank character of CITY-STATE and the first character of ZIP-CODE.

The program produces a report that looks like this:

Figure 1-8. Report Produced by FORMAT Options

```
Wed. Mar 30, 1988            LIST OF CUSTOMERS REPORT      PAGE 1


CUSTOMER


Able-1 Answering
2775 Park Av
San Jose, Ca. 95111

Grand Depression
27 E Main
Santa Clara, Ca. 95122
```

Selective Reporting

Now let's see some examples of selective reporting. The first program below prints all orders for a particular customer.

Figure 1-9. Program to Select Data for Reporting

```
1     system ex3,base=orders;
2     list(auto) orderhead;
3     data cust-no;
4     set(match) list (cust-no);
5     output(serial) orderhead;
```

3 The DATA verb is a data entry verb. When the program is run, the user is prompted to enter the cust-no. By default, the prompt uses the name of the element. This could be overridden here in several ways. For example, we could define entry text in the dictionary to be used when prompting for this field. We could also specify prompt text as a part of the DATA verb.

4 SET(MATCH) sets up a match criterion so that a record is selected only if the cust-no is equal to the one entered into the program via line 3. This is the default. We could also specify other match operators, such as LT (less than), GT (greater than), NE (not equal), etc.

5 OUTPUT(SERIAL) specifies a serial read through the orderhead dataset. When a cust-no matches the one entered by the user, then the record is selected.

Here is an example of running the program listed in Figure 1-9.

Figure 1-10. Report of Selected Data

```
CUST-NO> 1

ORDER-NO: CUST-NO: ORDER-STATUS: ORDER-DATE:
po1001    1         o              850101
po1002    1         o              850203
po1003    1         o              850127
po1004    1         o              850301
po1005    1         c              850212

5 RECORDS FOUND

EXIT/RESTART(E/R)?>
```

Setting up match criteria and serially reading datasets and files will work against any kind of file. However, in the case of Image, we can take advantage of search keys to retrieve the data more rapidly. Cust-no happens to be a search key in the orderhead dataset. Thus we can retrieve the data desired more rapidly if we change the program to the following:

Figure 1-11. Program to Select Data by Key Value

```
1     system ex4,base=orders;
2     list(auto) orderhead;
3     data cust-no;
4     set(key) list (cust-no);
5     output(chain) orderhead;
```

4 The retrieval is to be by the dataset key cust-no. The value of the cust-no is requested from the user in line 3.

5 The retrieval is to be done by Image CHAINed reads.

These examples show how a single input value can be used to qualify or select data. However, with a slight change, we can put the data selection logic into the hands of the user. Doing so allows a single program to be used to select data on the basis of any number of selection criteria entered by the user.

The following example is an expansion of the example in Figure 1-9.

Figure 1-12. Program to Let User Set Selection Criteria

```
1     system ex4a,base=orders;
2     list(auto) orderhead;
3     data(match) cust-no:
                  order-status:
                  order-date;
4     output(serial) orderhead;
```

We have combined lines 3 and 4 into a single DATA(MATCH) verb. This verb not only provides data entry, but also recognizes a variety of relational conditions that can be provided by the user at run time. These relational conditions are used as the selection criteria for data retrieval.

The following examples demonstrate how to run the program and enter various relational conditions.

Figure 1-13. User-Entered Selection Criteria

```
CUST-NO> 1,2

ORDER-STATUS>

ORDER-DATE>

ORDER-NO: CUST-NO: ORDER-STATUS: ORDER-DATE:
po1001    1        o             850101
po1002    1        o             850203
po1003    1        0             850127
po1004    1        0             850301
po1005    1        c             850212
po2001    2        h             880301
po2002    2        o             880312
```

In this example, the user specified that cust-no 1 and 2 are to be retrieved. If data values are provided without any relational conditions, equality is assumed. That is, retrieve the data if it matches any of the data values provided in the list.

No relational conditions were provided for items order-status or order-date. Thus, no data is excluded based on these two items.

Figure 1-14. More User-Entered Selection Criteria

```
CUST-NO> > 0 and < 2

ORDER-STATUS> <> c,h

ORDER-DATE> 85^^

ORDER-NO: CUST-NO: ORDER-STATUS: ORDER-DATE:
po1001    1        o             850101
po1002    1        o             850203
po1003    1        o             850127
po1004    1        o             850301
```

In this example, the relational conditions input by the user specify that order data is to be retrieved if the cust-no is greater than 0 and less than 2 and order-status is not equal to “c” or “h” and order-date begins with 85.

Reporting from Multiple Datasets

The following program prints the customer's name rather than the customer's cust-no. This requires data to be retrieved from both the customer set and the orderhead set.

Figure 1-15. Program to Report from Two Datasets

```
1     system ex5,base=orders;
2     list(auto) customer;
3     list(auto) orderhead;
4     data cust-no;
5     set(key) list (cust-no);
6     get customer,list=(@);
7     format name:
             order-no:
             order-status;
8     output(chain) orderhead,list=(@);
```

2 LIST(AUTO) reserves space to hold records from the customer set.

3 LIST(AUTO) reserves space to hold records from the orderhead set.

4 DATA asks the user to input the value of the cust-no to be retrieved.

5 SET(KEY) specifies that cust-no is a key element. For Image, this establishes the value to be used for direct retrieval from a master dataset and chained retrieval from a detail dataset.

6 GET CUSTOMER gets the customer record. The LIST=(@) option specifies that we retrieve the entire record. This is equivalent to specifying list=(cust-no,name,street-addr,city-state,zipcode);

7 The default option for the OUTPUT verb is to print all data items that we have listed. To limit our printing, we must use the FORMAT verb first to indicate which data items we want to print.

8 Since the program is accessing data from more than one dataset, we must specify the subset of data that is to be retrieved by OUTPUT. The LIST=(@) limits data retrieval to the elements in the orderhead dataset.

Figure 1-16 shows the report produced by this program.

Figure 1-16. Report from Two Datasets

```
CUST-NO> 1

NAME:                ORDER-NO: ORDER-STATUS:
Able-1 Answering     po1001     o
Able-1 Answering     po1002     o
Able-1 Answering     po1003     o
Able-1 Answering     po1004     o
Able-1 Answering     po1005     c

5 RECORDS FOUND
EXIT/RESTART(E/R)?>
```

Now let's create a program to retrieve all orders that are open, i.e., all orders that have order-status equal to "o". Also, let's print out the details of each order. The operation requires the use of a master and two detail sets: customer, orderhead, and orderline.

Figure 1-17. Program to Report from Three Datasets

```
 1     system ex6,base=orders;
 2     list(auto) orderhead;
 3     move (order-status) = "o";
 4     set(match) list (order-status);
 5     find(serial) orderhead,list=(@)
                             ,perform=get-orderdata;
 6     exit;

 7     get-orderdata:

 8     level;
 9       set(key) list (cust-no);
 10       list(auto) customer;
 11       get customer,list=(@);
 12       set(key) list (order-no);
 13       list(auto) orderline;
 14       format name:
                 order-no:
                 line-no:
                 part-number:
                 quantity;
 15       output(chain) orderline,list=(@)

                            ,nocount;
 16     end(level);
 17     return;
```

3 MOVE sets up the match value that we want for selection. We want to select open orders only.

4 SET(MATCH) establishes the match criterion for the data retrieval verb FIND (in line 5). The match element is order-status and the match value is “o”.

5 The FIND verb retrieves all records from the orderhead set that match the selection criterion of being an open order. The LIST=(@) specifies that all

data elements in the orderhead dataset are to be retrieved (for each record) that is selected. For each record that is retrieved, control is transferred to the label get-orderdata as specified by the PERFORM= option.

8 We use LEVEL to take advantage of some automatic housekeeping provided by Transact. This statement and the END(LEVEL) statement on line 16 specify a program area where data elements retrieved from data sets are temporarily stored. When the level is ended, the storage area for the elements is released. LEVELs will be discussed more in later sections.

9 Lines 9-11 setup and retrieve the customer record for the customer number just retrieved from orderhead.

12 SET(KEY) sets up the retrieval key for order details.

13 LIST(AUTO) reserves the space to hold the orderline record.

14 FORMAT sets up the list of data items that we want to report via output. The name comes from the customer dataset. All other items come from orderline.

15 OUTPUT(CHAIN) retrieves the order details and prints the data. The NOCOUNT option suppresses printing the number of records found.

16 END(LEVEL) releases the storage space for customer and orderline records.

17 RETURN specifies that control is to be returned to the statement that contains the PERFORM instruction. This is the FIND verb in statement 5.

This program produces the report shown below. Since the OUTPUT verb controls retrieval of detail information for each order, the report is formatted with headings for each order number.

Figure 1-18. Report from Three Datasets

```
NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:      :

Able-1 Answering      po1001    10       p121         75
Able-1 Answering      po1001    20       p123         150

NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:      :
Able-1 Answering      po1002    10       p122         50
Able-1 Answering      po1002    20       p124         10
Able-1 Answering      po1002    30       p127         243

NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:      :
Able-1 Answering      po1003    10       p121         10

NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:      :
Able-1 Answering      po1004    10       p122         20

NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:      :
Grand Depression      po2003    10       p126         50

END OF PROGRAM
:
```

If we want to see the same data in a more typical report format, then we can use the other reporting verb in Transact, which is the DISPLAY verb.

The following program demonstrates how the report above can be generated using this verb.

Figure 1-19. Program to Create a Report with DISPLAY(TABLE)

```
1     system ex7,base=orders;
2     list(auto) orderhead;
3     move (order-status) = "o";
4     set(match) list (order-status);
5     find(serial) orderhead,list=(@)
                            ,perform=get-orderdata;
6     exit;

7     get-orderdata:

8     level;
9       set(key) list (cust-no);
10       list(auto) customer;
11       get customer,list=(@);
12       set(key) list (order-no);
13       list(auto) orderline;
14       find(chain) orderline,list=(@)
                               ,perform=displayit;
15    end(level);
16    return;

17    displayit:

18    display(table) name:
                            order-no:
                            line-no:
                            part-number:
                            quantity;
19    return;
```

14 Replace the FORMAT and OUTPUT verbs with a FIND verb which performs a routine to display the data.

18 DISPLAY displays the information. The TABLE option specifies that we want Transact to print the headings at the start of each new page.

The items to be reported are specified as a part of the DISPLAY verb.

Our report now looks like this:

Figure 1-20. Report created by DISPLAY(TABLE)

```
NAME:                ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY:
  Able-1 Answering     po1001    10       p121         75
  Able-1 Answering     po1001    20       p123         150
  Able-1 Answering     po1002    10       p122         50
  Able-1 Answering     po1002    20       p124         10
  Able-1 Answering     po1002    30       p127         243
  Able-1 Answering     po1003    10       p121         10
  Able-1 Answering     po1004    10       p122         20
  Grand Depression     po2003    10       p126         50

END OF PROGRAM
:
```

Data selection can also be specified programmatically by using Transact's IF verb. This verb is very similar in functionality to the IF verb in languages such as COBOL and Pascal.

The following program illustrates how IF can be used to select a report of order details by order line, if the order line generates sales of over $1000. This program brings into play a fourth dataset, PARTS. It also changes the order in which data is retrieved.

Figure 1-21. Program to Select Data by the Conditional Verb IF

```
1     system ex7a,base=orders;
2     list(auto) orderhead;
3     move (order-status) = "o";
4     set(match) list (order-status);
5     find(serial) orderhead,list=(@)
                           ,perform=get-orderdata;
6     exit;

7     get-orderdata:

8     level;
9     set(key) list (order-no);
10     list(auto) orderline;
11     find(chain) orderline,list=(@)
                           ,perform=select-orderline;
12     end(level);
13     return;

14     select-orderline:

15     level;
16     set(key) list (part-number);
17     list(auto) parts;
18     get parts,list=(@);
19     if [(quantity) * (selling-price)] > 1000 then
20       do
21       set(key) list (cust-no);
22       list(auto) customer;
23       get customer,list=(@);
24       display(table) name:
                                order-no:
                                line-no:
                                part-number:
                                quantity:
                                selling-price;
25       doend;
26     end(level);
27     return;
```

9	Lines 9-11 get the order information from orders before retrieving the customer information.
16	Lines 16-18 set up and retrieve the parts record for the current part number.
19	A further screening is done on the record.
20	If the record passes the screening in line 19, then lines 20-25 are performed.
21	Lines 21-23 get the customer name.

An example of the report follows:

Figure 1-22. Data Selected with Conditional Verb IF

```
NAME:               ORDER-NO: LINE-NO: PART-NUMBER: QUANTITY: SELLING-PRICE:
Able-1 Answering    p01001    20       p123         150       20.00
Able-1 Answering    p01002    10       p122         50        21.00
Able-1 Answering    p01002    30       p127         243       50.00
Grand Depression    p02003    10       p126         50        40.00
```

2 Using Character Mode I/O

The next logical step in learning about Transact is to look at how to maintain Image datasets. This section therefore considers the three types of maintenance: adding data, changing data, and deleting data.

As the programs that we examine develop in capability, we will also look at some of the useful features of Transact, like the many ways to loop, or repeat, activity, and command mode, which lets the user control program flow.

Adding Data to a Dataset

This first program adds new customers to the customer set.

Figure 2-1. Program to Add Data to a Dataset

```
1     system ex8,base=orders(";");
2     level;
3       prompt cust-no:
                name:
                street-addr:
                city-state:
                zipcode;
4       put customer;
5     end;
```

1 By including the database password (";") in the SYSTEM statement, we avoid prompting the user for it.

2 The LEVEL verb sets up a looping structure which repeats whenever the END verb is executed (line 5 in the example). Levels can be nested, as we will see in later examples. What happens when the END verb is executed is that Transact keeps track of the start of the code identified by each LEVEL verb. Control returns to the starting point of the current level whenever the END verb is executed.

3 The PROMPT verb sets up data storage and prompts the user, one data item at a time, to enter values for a new record. Transact automatically generates entry text to identify each item as it should be entered. By default, the prompt is the name of the data item.

4 The PUT statement adds the record just entered to the customer set.

Figure 2-2. Interactive Data Entry to a Dataset

```
CUST-NO> 301

NAME> Joe's Bike Shop
STREET-ADDR> 1243 East Julian
CITY-STATE> San Jose, Ca.
CUST-NO> ]
EXIT/RESTART(E/R)?>
END OF PROGRAM
:
```

The program repeats, or loops, indefinitely. When the last customer has been added, The user stops the loop by entering the special character "]". This character is reserved in Transact to signify the end of user interaction.

The program called for the zipcode to be entered, but we see that the user was not prompted for it. The reason is not a bug in the program. Instead, the comma entered between the city and state sent an unexpected message to the Transact processor.

Transact allows user input to be stacked. That is, users who know what the program is going to ask for can enter the data in advance. The default field separator for Transact is the comma.

Thus our program did not work correctly. When we responded to the CITY-STATE prompt, Transact associated San Jose with CITY-STATE and associated CA. with ZIPCODE.

The following program gets around this problem by changing the default field separator.

Figure 2-3. Changing the Default Input Field Delimiter

```
1     system ex9,base=orders(";");
2     set(delimiter) "/";
3     level;
4       prompt cust-no:
                name:
                street-addr:
                city-state:
                zipcode;
5       put customer;
6     end;
```

2 The SET(DELIMITER) statement makes the field separator a slash rather than the comma.

An example of a terminal session is shown in Figure 2-4.

Figure 2-4. Using a Programmer-Defined Field Delimiter

```
CUST-NO> 303
NAME> John's Consulting
STREET-ADDR> 5489 El Camino
CITY-STATE> Santa Clara, Ca.
ZIPCODE> 95143
CUST-NO> 304/The Flower Shop/123 1st Street/San Jose, Ca./95125
CUST-NO> ]
EXIT/RESTART(E/R)?>
END OF PROGRAM
:
```

In the example above, we entered the data for the first customer one item at a time. The data for the second customer was all entered on one line by separating the fields with slashes. Note that Transact did not prompt for the data items that were stacked.

Let's run the program again to see what happens if the customer already exists.

Figure 2-5. Automatic Error Handling for a Duplicate Record

```
CUST-NO> 301                                                    |
|     NAME> New name                                            |
|     STREET-ADDR> New street                                   |
|     CITY-STATE> New city                                      |
|     ZIPCODE> 123                                              |
|     *ERROR:DUPLICATE KEY VALUE IN MASTER  (IMAGE 43,7,CUSTOMER)|
|     CUST-NO> ]                                                |
|     EXIT/RESTART(E/R)?>                                       |
|     END OF PROGRAM                                            |
|     :
```

Transact provides us with an error message and restarts the Transaction at the point of data entry. We can make the program more user friendly by checking to see whether the customer exists before asking for the rest of the customer data. Then if an error occurs, all the input will not have to be entered a second time. The program now looks like this:

Figure 2-6. Program to Check Item Entered by User

```
1     system ex10,base=orders(";");
2     set(delimiter) "/";
3     level;
4       prompt cust-no,checknot=customer;
5       prompt name:
                street-addr:
                city-state:
                zipcode;
6       put customer;
7     end;
```

4 The CHECKNOT=CUSTOMER causes the cust-no input to be validated against the customer dataset to verify that an entry does not already exist. If an entry does exist, Transact provides an error message and prompts for input of cust-no again.

This results in the following dialog when the program is run and the user enters a customer number that already exists:

Figure 2-7. User Interaction with Early Error Checking

```
CUST-NO> 301                                          |
|     *ERROR: ENTRY ALREADY EXISTS  (IMAGE 1,4,CUSTOMER)|
|     CUST-NO> 306                                     |
|     NAME> High Fashions                              |
|     STREET-ADDR> 1 The Embarcadero                   |
|     CITY-STATE> San Francisco, Ca.                   |
|     ZIPCODE> 93245                                   |
|     CUST-NO> ]                                       |
|     EXIT/RESTART(E/R)?>                              |
|     END OF PROGRAM                                   |
|     :
```

Updating Data in a Dataset

The programs that follow demonstrate three types of updating. These are changing data for non-key items, changing data for key items, and deleting records.

Figure 2-8. Program to Update Data in a Dataset

```
1     system ex11,base=orders(";");
2     set(delimiter) "/";
3     list(auto) customer;
4     level;
5       data cust-no,check=customer;
6       set(key) list (customer);
7       get customer;
8       display;
9       data(set) name:
                  street-addr:
                  city-state:
                  zipcode;
10       update customer;
11     end;
```

3 LIST reserves space to hold the customer record. In the previous example we let the PROMPT verb reserve space automatically as we went along. LIST prepares a temporary storage area to receive data. When LIST is used, DATA is used to prompt for data and receive the input. In these simple programs, the two methods produce the same result. For more complex data manipulation, setting up the temporary area before prompting for data provides more flexibility.

5 Since the space has already been reserved, we use the DATA verb rather than the PROMPT verb. Since we are updating existing records, we want to verify that the cust-no that is entered already exists. If it does not exist, an error message is generated and the prompt repeated. CHECK=CUSTOMER does this for us.

6 SET(KEY) sets up the IMAGE key to be used to retrieve the customer record.

8 DISPLAY displays the record. This gives the user a chance to see if this is really the record he meant to update.

9 The SET option specifies that if the user presses [[RETURN]] in response to any prompt, the original data for that item is retained. If we did not use this option, then when the user pressed [[RETURN]] the item would

become spaces or zero, depending on whether it was an alphanumeric or numeric item.

10 UPDATE moves the record into the customer set.

Figure 2-9. Interactive Updating of a Dataset

```
CUST-NO> 305
*ERROR: NO ENTRY FOUND  (IMAGE 17,9,CUSTOMER)
CUST-NO> 301
CUST-NO: NAME:              STREET-ADDR:      CITY-STATE: ZIPCODE:
301      Joe's Bike Shop    1243 East Julian  San Jose Ca.

NAME>
STREET-ADDR>
CITY-STATE>
ZIPCODE> 12345
CUST-NO> 301
CUST-NO: NAME:              STREET-ADDR:         CITY-STATE: ZIPCODE:
301      Joe's Bike Shop     1243 East Julian     San Jose     12345

NAME> ]
EXIT/RESTART(E/R)?>
END OF PROGRAM
:
```

Note that a valid cust-no has to be entered before the program will continue.

In the above example, we corrected the zipcode, but left the other items as they were.

When we enter the same customer number a second time, Transact displays the updated record, thereby giving us an easy way to verify the data just entered--or to further change the record if necessary.

In the next program, we take advantage of the fact that Transact has to retrieve the customer record when it verifies cust-no. In fact, this feature actually makes our program less complicated and eliminates an extra database access.

Figure 2-10. Program Using a Record Already Retrieved

```
1     system ex12,base=orders(";");
2     set(delimiter) "/";
3     list(auto) customer;
4     level;
5       data cust-no,check=customer;
6       get(current) customer;
7       display;
8       data(set) name:
                  street-addr:
                  city-state:
                  zipcode;
9       update customer;
10     end;
```

6 We know that Transact already retrieved the record in order to do the validation check in step 5. Therefore, by using the CURRENT option to get the record out of the temporary storage area, we save both the physical time of retrieving the record a second time and also simplify the program.

What if we need to change the cust-no to another value. Since this is a key item to IMAGE, we have to treat it differently. One way is to delete the old record and add a new. The next example shows this method.

Figure 2-11. Program to Change a Key Field Value

```
1     system ex13,base=orders(";");
2     set(delimiter) "/";
3     list(auto) customer;
4     level;
5       data cust-no,check=customer;
6       get(current) customer;
7       display;
8       delete(current) customer;
9       data cust-no ("enter new cust-no"),checknot=customer;
10       data(set) name:
                   street-addr:
                   city-state:
                   zipcode;
11       put customer;
12     end;
```

8 The DELETE statement removes the old customer record.

9 This DATA statement overrides the default prompt for a data field, substituting the prompt enter new cust-no. It also verifies that this new customer does not already exist.

10 The temporary storage area still contains the values for the old customer that we just deleted. The user may change any of these values or leave them as is by pressing [[RETURN]] to any of the prompts.

11 The new customer record is added to the dataset.

Figure 2-12. Changing a Key Field Value

```
CUST-NO> 301
CUST-NO: NAME:              STREET-ADDR:       CITY-STATE: ZIPCODE:

301      The Cannery        123 Worthy Street   Waltham, Ma. 46534
enter new cust-no> 303
*ERROR: ENTRY ALREADY EXISTS  (IMAGE 1,16,CUSTOMER)
enter new cust-no> 302
NAME>
STREET-ADDR>
CITY-STATE>
ZIPCODE> 46533

CUST-NO> 302
CUST-NO: NAME:             STREET-ADDR:         CITY-STATE: ZIPCODE:
302      The Cannery       123 Worthy Street    Waltham, Ma. 46533
enter new cust-no> 301
NAME>
STREET-ADDR>
CITY-STATE>
ZIPCODE>

CUST-NO> ]
EXIT/RESTART(E/R)?>
END OF PROGRAM
:
```

Most of the time this program works correctly. However, if a problem like a system crash occurs immediately after the delete of the old customer and just before the add of the new customer, we force the user to add the complete customer record after system recovery.

We could modify this program to first add the new customer and then delete the old, but to do this would require setting up a data item within the program to hold the value of the old cust-no while we added the new one. We may do this in a later example, but another way to do this is to use the special verb REPLACE, which does the hard work for us. The following program shows how to use REPLACE to update a key field.

Figure 2-13. Program to Change Key Values Using REPLACE

```
1     system ex14,base=orders(";")
2     set(delimiter) "/";
3     list(auto) customer;
4     level;
5       data(key) cust-no ("enter old cust-no");
6       get customer;
7       display;
8       data(update) cust-no ("enter new cust-no")
                             ,checknot=customer;
9       replace customer;
10     end;
```

5 The KEY option automatically sets up the IMAGE key, preparing us to retrieve the customer record. When the KEY option can be used, the DATA verb performs the input of data and also sets the IMAGE key for database retrieval. Note that we are also supplying our own prompt.

8 DATA(UPDATE) tells Transact that we really do want to change a key field in the record. CHECKNOT verifies that the new customer number does not already exist in the dataset.

9 REPLACE causes the new customer record to be added before the old customer record is deleted.

In this example, we did not allow any other fields to be updated. We could have, by prompting with DATA(UPDATE) for each of the fields. In other words, when you use the REPLACE verb, you must give Transact the new values, using DATA(UPDATE) or its equivalent. Any data items not specified this way, retain the values from the original record.

Figure 2-11 used the DELETE verb to change a key value. Figure 2-14 is an example of a program that uses the DELETE verb to simply delete records.

Figure 2-14. Program to Delete Records

```
1     system ex15,base=orders(";");
2     list(auto) customer;
3     level;
4       data(key) cust-no ("enter cust-no to delete");
5       get customer;
6       display;
7       input "delete this customer?";
8       if input = "Y","YES"
9         then delete(current) customer
10         else display "customer not deleted";
11    end;
```

7 The INPUT verb is used to get a value from the user that is to be tested by an IF statement. Here it gets verification from the user that this is the customer to delete.

8 Whatever the user types in as a response to the INPUT verb is automatically upshifted. Thus we check to see if the response is y or yes. If so, the customer is deleted. If not, we print a confirmation that nothing was done.

Figure 2-15. Interactively Deleting Records

```
enter cust-no to delete> 301

CUST-NO: NAME:                STREET-ADDR:       CITY-STATE: ZIPCODE:
301      The Cannery          123 Worthy Street   Waltham, Ma. 46533

delete this customer? n
customer not deleted

enter cust-no to delete> 301

CUST-NO: NAME:               STREET-ADDR:        CITY-STATE: ZIPCODE:
301      The Cannery         123 Worthy Street    Waltham, Ma. 46533

delete this customer? y

enter cust-no to delete> 301

*ERROR: NO ENTRY FOUND  (IMAGE 17,10,CUSTOMER)

enter cust-no to delete>
```

Looping Structures

In most of the examples, we have used LEVEL as the way to get a program to loop. Transact also has the verbs REPEAT, UNTIL, and WHILE to control looping.

For example, we could have written the last program using REPEAT as follows:

Figure 2-16. Program Using REPEAT to Loop

```
1     system ex16,base=orders(";");
2     list(auto) customer;
3     data cust-no ("enter cust-no to delete");
4     if (cust-no) <> 0
5       then
6         repeat
7           do
8           set(key) list (cust-no);
9           get customer;
10           display;
11           input "delete this customer?";
12           if input = "Y","YES"
13             then delete(current) customer
14             else display "customer not deleted";
15           data cust-no ("enter cust-no to delete");
16           doend
17         until (cust-no) = 0;
```

7 The DO verb designates the start of a block of code that is to be executed under the control of the current verb, in this case the REPEAT verb. The block of code is terminated with a DOEND verb. This is line 16 in the example. Thus lines 8 through 15 are executed under the control of the REPEAT statement.

DO/DOEND can also be used with the WHILE and IF verbs.

Replacing REPEAT with WHILE, the program looks like: WHILE statement”

Figure 2-17. Program Using WHILE to Loop

```
1     system ex17,base=orders(";");
2     list(auto) customer;
3     data cust-no ("enter cust-no to delete");
4     while (cust-no) <> 0
5       do
6       set(key) list (cust-no);
7       get customer;
8       display;
9       input "delete this customer?";
10       if input = "Y","YES"
11         then delete(current) customer
12         else display "customer not deleted";
13       data cust-no ("enter cust-no to delete");
14       doend;
```

It is easier programmatically to use the LEVEL verb. However, when we do so, we ask the user to do a bit more in order to stop things. The user must enter the character] to get the loop to stop.

In order to use the simpler LEVEL structure and still keep things easy for the user, we could put an extra test in our LEVEL program to accomplish the same thing as we did using REPEAT or WHILE. The program now looks like this.

Figure 2-18. Program Using LEVEL to Loop

```
 1     system ex15a,base=orders(";");
 2     set(delimiter) "/";
 3     list(auto) customer;
 4     level;
 5       data cust-no ("enter cust-no to delete");
 6       if (cust-no)
0
 7         then
 8           do
 9           set(key) list (cust-no);
10           get customer;
11           display;
12           input "delete this customer?";
13           if input = "Y","YES"
14             then delete(current) customer
15             else display "customer not deleted";
16           doend
17         else
18           end(level);
```

18 Programmatically, END(LEVEL) is the same as if the user entered the special key]. It terminates the current level of the program.

The preceding three examples allow the user to get out of the loop by just pressing [[RETURN]] in response to the cust-no prompt. That is, each program continues to loop until the cust-no input is zero, which is the result of pressing [[RETURN]] in response to the prompt for cust-no.

Command Mode

Much of the control of a program can be removed altogether from the program logic and put into the hands of the user via built-in commands. The following program illustrates a way to build the functions of adding, updating, and reporting customer data into a program.

Figure 2-19. Program Using Command Mode for Add, Update, and Display

```
1     system ex18,base=orders;
2
3     $$help:
4       display "List of customer transactions:";
5       set(command) command;
6
7     $$add:
8       set(delimiter) "/";
9       prompt cust-no,checknot=customer;
10       prompt name:
11              street-addr:
12              city-state:
13              zipcode;
14       put customer;
15
16     $$update:
17       set(delimiter) "/";
18       list(auto) customer;
19       data cust-no,check=customer;
20       get(current) customer;
21       display;
22       data(set) name:
23                 street-addr:
24                 city-state:
25                 zipcode;
26       update customer;
27
28     $$display:
29       list(auto) customer;
30       data cust-no;
31       set(key) list (cust-no);
32       output customer;
```

3 A command is identified to Transact by the special characters “$$”. In our program we have identified four commands. These are: HELP, ADD, UPDATE, and DISPLAY.

We can think of commands as entry points into a program. Typically a command will do a unit of work or a transaction.

We do not need to do anything special to identify the end of a command sequence. During the processing flow, when Transact detects the starting point of a new command, it returns control to the user and reissues the prompt > for the user to specify a new command or transaction.

Command processing removes a lot of the control over processing flow from the program and puts it in the Transact processor.

5 SET(COMMAND) sets up a mini help system within the program. If the user types HELP, we display the valid commands for this program. Transact also does this automatically if the user types in the special command COMMAND.

The following example shows the execution of this program.

Figure 2-20. Command Mode Interaction

```
> help
List of customer transactions:
HELP
ADD
UPDATE

> add

CUST-NO> 401
NAME> Brown Publishing
STREET-ADDR> 123 Wrong Street
CITY-STATE> Reno, Nevada
ZIPCODE> 36745

> update

CUST-NO> 401
CUST-NO: NAME:                STREET-ADDR:         CITY-STATE: ZIPCODE:
401      Brown Publishing     123 Wrong Street     Reno, Nevada 36745
NAME>
STREET-ADDR> 123 Right Street
CITY-STATE>
ZIPCODE>

>repeat display

CUST-NO> 401
CUST-NO: NAME:                STREET-ADDR:         CITY-STATE: ZIPCODE:
401      Brown Publishing     123 Right Street     Reno, Nevada| 36745
CUST-NO> 1
CUST-NO: NAME:                STREET-ADDR:         CITY-STATE: ZIPCODE:
1        Able-1 Answering     2775 Park Av         San Jose, Ca 95111

CUST-NO> ]

> exit

END OF PROGRAM
:
```

The user merely types in the command to be executed. After Transact finishes the transaction, it prompts for the next command.

The user can cause a command to be iterative by prefacing the command with the special command REPEAT as in REPEAT DISPLAY, where DISPLAY is one of the commands in our program.

We can add one additional level of commands to our program, called a subcommand. For example, we want to have the general facilities to ADD, UPDATE, and DISPLAY in our program. But within each we want to build specific transactions such as adding customers, adding part numbers, etc.

These subcommands are identified to Transact by the special character “$”. Sub-commands must be placed after the command they apply to and are executed by the user by typing in both the command and subcommand names. The following program and sample execution demonstrate this.

Figure 2-21. Program with Subcommands

```
1     system ex19,base=orders;
2
3     $$help:
4       display "List of transactions:";
5       set(command) command;
6
7     $$add:
8     $help:
9      display "types of add maintenance:";
10      set(command) command(add);
11     $customer:
12       set(delimiter) "/";
13       prompt cust-no,checknot=customer;
14       prompt name:
15              street-addr:
16              city-state:
17              zipcode;
18       put customer;
19
20     $parts:
21     set(delimiter) "/";
22     prompt part-number,checknot=parts;
23     prompt description;
24     put parts;
25
26
27     $$update:
28       set(delimiter) "/";
29       list(auto) customer;
30       data cust-no,check=customer;
31       get(current) customer;
32       display;
33       data(set) name:
34                 street-addr:
35                 city-state:
36                 zipcode;
37       update customer;
38
39     $$display:
40       list(auto) customer;
41       data cust-no;
42       set(key) list (cust-no);
43       output customer;
```

Figure 2-22. User Interaction with Subcommands

```
> help
List of transactions:

          HELP
          ADD
          UPDATE
          DISPLAY

> add help
types of add maintenance:

          HELP
          CUSTOMER
          PARTS

> add parts
PART-NUMBER> p001
DESCRIPTION> air socket

> add customer
CUST-NO> 501
NAME> Foot Appeal
STREET-ADDR> 323 Bottom Ave.
CITY-STATE> Atlanta, Ga.
ZIPCODE> 23845

>
```

3 Using VPLUS and IMAGE

This section illustrates how easily VPLUS forms can be used to maintain and get information from an IMAGE dataset.

Adding Data to a Dataset

The SHOW FILE command in Dictionary/3000 reports the following for our form called vcustomer and a dataset called customer. This form and dataset are used in the following examples.

Figure 3-1. Dictionary Definitions of Customer Form and Dataset

```
 > show file
                FILE vcustomer
FILE                 TYPE: RESPONSIBILITY:
VCUSTOMER            FORM

ELEMENT(ALIAS):              PROPERTIES: ELEMENT(PRIMARY):
CUST-NO                      9 (4,0,4)           CUST-NO
NAME                         X (20,0,20)         NAME
STREET-ADDR                  X (20,0,20)         STREET-ADDR
CITY-STATE                   X (20,0,20)         CITY-STATE
ZIPCODE                      X (6,0,6)           ZIPCODE

> show file
               FILE customer

FILE                 TYPE: RESPONSIBILITY:
CUSTOMER             MAST

ELEMENT(ALIAS):              PROPERTIES: ELEMENT(PRIMARY):
CUST-NO              *       9 (4,0,4)           CUST-NO
NAME                         X (20,0,20)         NAME
STREET-ADDR                  X (20,0,20)         STREET-ADDR
CITY-STATE                   X (20,0,20)         CITY-STATE
ZIPCODE                      X (6,0,6)           ZIPCODE
```

The important things to know about the above are that for both form vcustomer and dataset customer, the valid data items or elements are: cust-no, name, street-addr, city-state, and zipcode. These elements occur in the same order in both the dataset and the

form, and the parallel data items have compatible data types and lengths. All of the forms in our formsfile are shown in appendix H. The form vcustomer is also shown below:

Figure 3-2. VPLUS Form for Adding Customer Data

```
vcustomer                       add a customer

                      number  [    ]

                      name    [                    ]

                      address [                    ]

                      city,state   [                    ]

                      zipcode [      ]
```

The program below causes the VPLUS form vcustomer to be displayed on the terminal. After the operator fills in the blanks and presses [[ENTER]], the data is added to the customer dataset.

Figure 3-3. Program accessing a VPLUS form

```
1     system ex20,base=orders,vpls=formfile;
2     list(auto) customer;
3     get(form) vcustomer,init;
4     put customer;
```

1 The option VPLS= on the SYSTEM statement specifies that the name of our formsfile is formfile.

3 GET(FORM) displays the form on the terminal, initialize the fields, and accepts the data entered by the user.

The example above shows how simple it is to code a program for data entry. However, it only allows someone to enter data one time and then it quits. As we have seen in previous sections, there are several ways to get this program to loop until we are done entering data.

First, we can make this a command-driven program. Figure 3-4 illustrates this. If the user enters the command REPEAT ADD instead of just ADD, the program will execute lines 3-5 forever. Just as the special key] is used to stop processing in character mode, so the function key [[F8]] is used to stop processing in block mode. Thus this program will continue to execute until the user presses [[F8]].

Figure 3-4. Using Command Mode with VPLUS for Looping

```
1     system ex21,base=orders,vpls=formfile;
2     $$add:
3       list(auto) customer;
4       get(form) vcustomer,init;
5       put customer
```

Another way to get the program to loop is to build the repeating structure into the program, using either a LEVEL or a REPEAT statement. The following programs show these two possibilities.

Figure 3-5. Using LEVEL with VPLUS for Looping

```
1     system ex22,base=orders,vpls=formfile;
2     level;
3       list(auto) customer;
4       get(form) vcustomer,init;
5       put customer;
```

Lines 3-5 of the above program execute until the user presses [[F8]].

Figure 3-6. Using REPEAT with VPLUS for Looping

```
1     system ex23,base=orders,vpls=formfile;
2     list(auto) customer;
3     repeat
4       do
5       get(form) vcustomer,init;
6       put customer;
7       doend
8     until (cust-no) = 0;
9     exit;
```

The user can terminate this program either by pressing [[ENTER]] without entering any data or by pressing [[F8]]. However, [[ENTER]] causes Transact to create a blank record with a zero customer number.

Since we probably do not want such a record in the dataset, we should change the above program as shown in Figure 3-7 to make it more practical.

Figure 3-7. Preventing a Blank Record

```
1     system ex24,base=orders,vpls=formfile;
2     list(auto) customer;
3     repeat
4       do
5       get(form) vcustomer,init;
6       if (cust-no)
0
7         then put customer;
8       doend
9     until (cust-no) = 0;
10     exit;
```

Updating Data in a Dataset

Now that we have added data to the customer dataset, how do we go about updating this data? This section shows two approaches. The only difference between the two is the way in which we ask for input of the cust-no.

The first example uses the form vcustomer for both input of the cust-no and input of the data. The second example sets up a separate form vcustno for entering the customer number. It then uses form vcustomer for data entry.

Figure 3-8. Using LEVEL with VPLUS to Update Data

```
1 system ex25,base=orders,vpls=formfile;
2 level;
3     list(auto) customer;
4     get(form) vcustomer,init;
5     set(key) list (cust-no);
6     get customer;
7     put(form) vcustomer;
8     get(form) vcustomer;
9     update customer;
```

4 First, GET(FORM) gets the cust-no for the record that we want to update.

5 SET establishes the key or IMAGE path into the customer dataset to retrieve the record to be updated.

6 GET retrieves the record from the customer dataset.

7 PUT(FORM) displays the information that currently exists for the customer on the screen. At this point the operator changes the field(s) that need to be updated.

8 GET(FORM) accepts the data from the screen.

9 UPDATE puts the record in the dataset customer.

As in previous examples, we can exit the LEVEL repeating loop at any time merely by pressing [[F8]].

This same program could be coded using the REPEAT construct as follows:

Figure 3-9. Using REPEAT with VPLUS to Update Data

```
1 system ex26,base=orders,vpls=formfile;
2 list(auto) customer;
3 repeat
4   do
5   get(form) vcustomer,init;
6   if (cust-no) <> 0
7      then
8         do
9         set(key) list (cust-no);
10        get customer;
11        put(form) vcustomer;
12        get(form) vcustomer;
13        update customer;
14        doend;
15     doend
16 until (cust-no) = 0;
```

6 The IF statement instructs Transact to quit if a customer number is not input. If one is input, then the DO/DOEND block of statements retrieve the information, display it on the screen, accept the changed data, and update the dataset.

As was indicated earlier, another way to get the cust-no is to introduce a separate form to ask for it first. Let's assume that we have set up this second form in the dictionary and in the formsfile. The dictionary listing for this form and the format of the form are shown below.

Figure 3-10. Dictionary Definition of Customer Number Form

```
show file
              FILE vcustno


FILE              TYPE: RESPONSIBILITY:
VCUSTNO           FORM

       ELEMENT(ALIAS):        PROPERTIES:         ELEMENT(PRIMARY):
       CUST-NO                9 (4,0,4)           CUST-NO
```

Figure 3-11. VPLUS Form for Customer Number

```
vcustno                              update customer data


                          enter customer number to update [ ]
```

To implement this, we need only change the name of the form in line 5 of Figure 3-9 above and we now have a program working with two forms.

Figure 3-12. Program accessing two VPLUS forms

```
1 system ex27,base=orders,vpls=formfile;
2 list(auto) customer;
3 repeat
4    do
5    get(form) vcustno,init;
6    if (cust-no) <> 0
7      then
8        do
9        set(key) list (cust-no);
10       get customer;
11       put(form) vcustomer;
12       get(form) vcustomer;
13       update customer;
14       doend;
15    doend
16 until (cust-no) = 0;
```

5 GET(FORM) displays the new form and accepts the cust-no from the user.

11 PUT(FORM) replaces the form vcustno with the form vcustomer, displays the current customer data, and allows the user to change it.

Reporting Data from a Dataset

Now let's use VPLUS to display data from the customer dataset.

One way to display the data is to display a record at a time using the same form we used to add and update data.

The following program illustrates this.

Figure 3-13. Using VPLUS to Display Data

```
1     system ex28,base=orders,vpls=formfile;
2     list(auto) customer;
3     find(serial) customer,perform=display-record;
4     exit;

5     display-record:

6       put(form) vcustomer,wait=
                           ,window=("press f1-f7 to continue");
7       return;
```

3 The FIND verb can also be a looping verb. In this example, it serially retrieves each record from the customer dataset. As each record is retrieved, the PERFORM= option passes control to the routine at label display-record. When the RETURN is encountered, control passes back to the FIND.

6 PUT(FORM) displays the information to the screen. The WAIT= option causes the program to wait until the user has pressed any of the function keys before continuing with the next record. The WINDOW= option displays a message in the VPLS window area to remind the user that he must press one of the function keys before the program will continue.

Another way to display the information back to the terminal is to format the data to look more like a report, but yet remain within VPLUS. To do this, we need to set up a form for the report heading information, a form to hold the detail information for each record, and a counter to increment so that we know when we have filled the screen.

Up until now, we have accepted the default selection as to what data elements are extracted and reported. Now we will override the default selection by using a qualifier on the verb to specify which subset of data we want.

The new forms we need are called vcustomerrh, and vcustomerrd. Their definition in the dictionary and the format of the forms are shown below.

Figure 3-14. Dictionary Definitions for Customer Forms to be Appended

```
>show file
                 FILE vcustomerrh

FILE                 TYPE: RESPONSIBILITY:
VCUSTOMERRH          FORM

> show file
                   FILE vcustomerrd

FILE                 TYPE: RESPONSIBILITY:
VCUSTOMERRD          FORM

         ELEMENT(ALIAS):             PROPERTIES: ELEMENT(PRIMARY):
         CUST-NO                     9 (4,0,4)          CUST-NO
         NAME                        X (20,0,20)        NAME
         STREET-ADDR                X (20,0,20)        STREET-ADDR
         CITY-STATE                  X (20,0,20)        CITY-STATE
         ZIPCODE                     X (6,0,6)          ZIPCODE
```

Figure 3-15. VPLUS Form for Customer Header

```
                            customer  address
```

Figure 3-16. VPLUS Form to be Appended

```
                                  [    ]   [                    ]
                                           [                    ]
                                           [                    ]
                                           [      ]
```

We want to FREEZE the report heading on the screen and APPEND each of the customer records. When the screen is full, which in this example is after displaying 5 customer records, we want to pause until the person requesting this report is ready to go on.

The following program illustrates these points.

Figure 3-17. Program with VPLUS Freeze and Append

```
1     system ex29,base=orders,vpls=formfile;
2     define(item) counter i(4);
3     list counter,init;
4     list(auto) customer;
5     put(form) vcustomerrh,freeze;
6     find(serial) customer,list=(cust-no:zipcode)
                          ,perform=display-record;
7     if (counter) > 0
8       then update(form) vcustomerrd,list=()
                                    ,wait=
                                    ,window=("press f1-f7 to
continue");
9     exit;

10     display-record:

11     let (counter) = (counter) + 1;
12     if (counter) = 5
         then
           do
13         let (counter) = 0;
14         put(form) vcustomerrd,wait=
                               ,window=("press f1-f7 to continue")
                               ,append;
           doend
15       else put(form) vcustomerrd,window=(" ")
                                  ,append;
16     return;
```

2 We define an element local to this program called counter. It is an integer that can store a 4-digit number, e.g. <= 9999.

3 LIST includes this new element in the list of variables that our program can access. The INIT option initializes its value to zero.

5 PUT(FORM) displays the report heading form to the terminal. The FREEZE option freezes the form on the terminal so that the scrolling portion of the screen begins below this form.

6 The LIST= option designates the items in the program list that the FIND verb is to retrieve. If we didn't do this, it would attempt to retrieve the item counter from dataset customer and would be unable to do so. The default for IMAGE data access verbs is to retrieve all items known to the program via the LIST verb unless this is overridden on the data access verb with the LIST= option. We will see more of this in the section on data structures.

The colon between cust-no and zipcode specifies a range of data items. In this example, we are telling Transact to limit itself to the range of items starting with cust-no and ending with zipcode.

7 Here we have ended the display of all full screens of customer data. Counter will be greater than zero if we have displayed another customer since the last time the screen was entirely filled. If so, then we must pause one more time to give the user the opportunity to review this data before going on. Lines 7 and 8 take care of the last set of records if there are fewer than five.

8 We only want to update the VPLUS message window and wait for a response. We save data transmission time by not re-sending the form data to the screen, thus the LIST=() option which specifies a null item list.

12 If this customer fills the screen, then wait for the user to indicate that he is ready to continue, otherwise append this new customer to the screen and go on.

This program produces the screen output shown below.

Figure 3-18. Result of VPLUS Freeze and Append

```
                          customer  address
                           [14  ]   [Furtado Inports     ]
                                    [1396 E Santa Clara  ]
                                    [Los Angeles, Ca.    ]
                                    [90189 ]
                           [15  ]   [Saxon's             ]
                                    [518 W San Carlos    ]
                                    [Santa Clara, Ca.    ]
                                    [94168 ]
                           [16  ]   [Pour House          ]
                                    [1475 Lipton Place   ]
                                    [San Jose, Ca.       ]
                                    [95122 ]
                           [18  ]   [Excl Chemical Co    ]
                                    [630 Walsh           ]
                                    [New York, NY        ]
                                    [44636 ]
                           [19  ]   [Hold A Hill Planter ]
                                    [651 El Camino       ]
                                    [Dallas, Texas       ]
                                    [45623 ]
 press f1-f7 to continue
```

Setting Up a Menu-Driven System

Now all we need to do is tie the pieces together with a menu and we have a complete system for adding, updating, and reporting data. A mainmenu form does the trick. The format of the form is:

Figure 3-19. VPLUS Form for Main Menu

```
mainmenu                                Customer module

                                        f1 = add customer

                                        f2 = update customer

                                        f3 = report customer

                                        F8 = exit
```

Now we can integrate the individual programs we wrote earlier into a program driven by the main menu. Depending on the choice entered on the mainmenu, the program will execute the code to add, update, or report a customer.

The program might look like the one below.

Figure 3-20. VPLUS Menu-Driven Program

```
 1 system ex30,base=orders,vpls=formfile;
 2 list(auto) customer;
 3 level;
 4    get(form) mainmenu,f1=add-customer
                        ,f2=update-customer
                        ,f3=report-customer;
 5    end;
 6 add-customer:
 7    get(form) vcustomer;
 8    put customer;
 9    end;
 10 update-customer:
 11   get(form) vcustno;
 12   set(key) list (cust-no);
 13   get customer;
 14   put(form) vcustomer;
 15   get(form) vcustomer;
 16   update customer;
 17   end;
 18 report-customer:
 19   get(form) vcustno;
 20   set(key)list (cust-no);
 21   get customer;
 22   put(form) vcustomer,wait=
 23                        ,window=("press f1-f7 to continue");
 24   set(form) vcustomer,window=(" ");
 25   end;
```

4	The GET statement drives the program. The mainmenu is displayed and, depending on the function key entered, various routines are executed. When any of the three tasks finishes, END returns control to the first line in the present level, which is line 4. The options F1=, F2=, and F3= specify a conditional transfer of control depending on the function key pressed. This is like a GOTO rather than a PERFORM. However, by proper use of the LEVEL and END, the logic flow can be made to resemble a PERFORM more closely.
5	END returns control to the start of the current level which means we return to statement 4.
6	The code to add a customer executes when the user presses [[F1]].
9	END returns control to the start of the current level which means we return to statement 4.
10	The code to update a customer executed when the user presses [[F2]].
17	END returns control to the start of the current level which means we return to statement 4.
18	The code to report a customer executes when the user presses [[F3]].
26	END returns control to the start of the current level which means we return to statement 4.

By default, if [[F8]] is pressed, we terminate the program.

4 Using KSAM and MPE

The examples in this section explain how to use Transact to maintain KSAM and MPE files. They assume that the KSAM and MPE file definitions have been added to a Dictionary/V dictionary. Appendix G contains file definitions for the file referenced by these examples.

Using KSAM

Adding Records

Adding a record to a KSAM file uses the same verbs and syntax as adding a record to an IMAGE dataset. Compare this program with Ex9 in Figure 2-3.

Figure 4-1. Program to Add Data to a KSAM File

```
1     system ex32,ksam=kcust(update);
2     set(delimiter) "/";
3     level;
4       prompt cust-no;
5       prompt name;
6       put kcust;
7     end;
```

1 The KSAM= option names the KSAM file to be accessed. The default access is READ only. The most inclusive capability is UPDATE.

Figure 4-2. Adding Data to a KSAM File

```
CUST-NO> 301
NAME> Wheeler Dealer
CUST-NO> 301
NAME> My Place
*ERROR: DUPLICATE KEY VALUE  (FSERR 171)  (KSAM 171,4,KCUST)
CUST-NO>
```

Figure 4-3. Program to Update Data in a KSAM File

```
 1     system ex33,ksam=kcust(update);
 2     set(delimiter) "/";
 3     list(auto) kcust;
 5     data cust-no;
 6     set(key) list (cust-no);
 7     get kcust;
 8     display;
 9     data name;
10     set(update) list(name);
11     replace kcust;
12     end;
```

10 In the KSAM file, both data items are key fields. Therefore, to change a field we need to add a new record and delete the old. We can code the steps ourselves, or we can take advantage of the REPLACE capability in Transact. SET(UPDATE) prepares to let Transact do the delete and add for us. In this statement, we tell Transact each field that we are going to update.

11 REPLACE does the add and delete, thereby changing the value for name to the new value we input.

Figure 4-4. Updating Data in a KSAM File

```
 CUST-NO> 301

CUST-NO: NAME:
301      Wheeler Dealer
NAME> My Place
CUST-NO> 301
CUST-NO: NAME:
301      My Place

NAME> ]
```

Using MPE Files

Adding Records

Adding a record to an MPE file uses the same verb and syntax as does adding a record to an IMAGE dataset or a KSAM file. We have the option of using LIST with DATA or just using PROMPT alone.

Figure 4-5. Program to Add Data to an MPE File

```
1     system ex34,file=batchinv(update);
2     list(auto) batchinv;
3     level;
4       data part-number:
5            location:
6            quantity;
7       put batchinv;
```

1 The FILE= option names the MPE file we are going to access. The default access is READ. UPDATE provides the most complete update capability to files.

Figure 4-6. Figure 4-6. Adding Data to an MPE File

```
PART-NUMBER> p101
LOCATION> bin1
INV-QUANTITY> 100
PART-NUMBER> p102
LOCATION> bin2
INV-QUANTITY> 105
PART-NUMBER> ]
```

Updating Records

Figure 4-7. Program to Update Records in an MPE File

```
1     system ex35,file=batchinv(update);
2     list(auto) batchinv;
3     level;
4       data part-number;
5       set(match) list (part-number);
6       find(serial) batchinv,perform=check-it-out;
7       end;
8     exit;
9
10     check-it-out:
11
12     display;
13     input "is this the record to update?";
14     if input="Y","YES"
15       then
16         do
17         data(set) location:
18                   quantity;
19         update batchinv;
20         doend;
21     return;
```

5 Since there is no chained access to MPE files, we set up a match criterion to match on part-number.

6 FIND(SERIAL) reads the file serially. For each part-number that matches the part-number value specified in line 5, the check-it-out routine is performed.

19 Updating a record in an MPE file uses the same verb and syntax as updating a record in an IMAGE dataset.

Figure 4-8. Updating Records in an MPE File

```
PART-NUMBER> p102

PART-NUMBER: LOCATION: QUANTITY:
p102         bin2      105
is this the record to update? y
LOCATION>
INV-QUANTITY> 110
PART-NUMBER> p102

PART-NUMBER: LOCATION: QUANTITY:
p102         bin2      110
is this the record to update? n
PART-NUMBER> ]
```

Direct access to an MPE file is possible if the MPE record number is known. Use the FIND verb with both the DIRECT and RECNO= options to accomplish this.

5 Automatic Error Handling and Prototyping

This section will demonstrate how easy it is to develop a working prototype of an application system by taking advantage of Transact's power, in particular the automatic error handling facility.

We will take a particular example and follow it from the initial prototype attempt until the finished product emerges.

Usually, when prototyping of a system is discussed, it is thought of in terms of throwaway code. This need not be the case with Transact. Unlike most other application development tools, Transact is a complete procedural language. However, as an integral part of the language, there are high level, more nonprocedural types of constructs that provide the true power of the language.

These high-level constructs and facilities can be used to advantage to produce a working prototype of a system. The prototype can then be reviewed with and even developed with the user community until mutual agreement has been reached as to the functionality of the system and its appearance to the user.

Because agreement can be reached much more quickly than with traditional languages, prototyping is a viable tool to help get the users more involved. At the same time, the effort of the application developer is not wasted because the code does not need to be thrown away as the production system is developed. Instead, much of the code can be retained as is, and only those procedures where Transact's automatic facilities do not provide the required control need to be expanded to provide the necessary control. The emphasis on expansion is there because typically the code is not rewritten at this point, but additional options are added to existing Transact constructs and additional code is written to provide the necessary control.

The example that we will develop in this chapter consists of providing the functionality to add an order to the database. We could prototype the database design, forms design, etc., as well as the program design, but let's assume that there are valid reasons to have the database design remain as it is. We will concentrate on how we present data to the user and get data from the user.

This program is our first attempted solution:

Figure 5-1. Basic Prototype Program for Adding Data

```
1     system ex36,base=orders,vpls=formfile;
2     list(auto) vorderhead;
3     list(auto) vorderline;
4     get(form) vorderhead,init,freeze;
5     put orderhead,list=(order-no,
6                         cust-no,
7                         order-status,
7.01                      order-date);
8     level;
9       get(form) vorderline,init,append;
10       put orderline,list=(order-no,
11                           line-no,
12                           part-number,
13                           quantity);
```

This program would not be considered a good working version, since it will only add one order. It must be restarted to add another order. However, the program does allow us to demonstrate or prototype for our user the data flow for adding an order to the database.

The program first displays and inputs data using the vorderhead form (line 4), then adds this data to dataset orderhead (lines 5 to 7).

It will then repeatedly (line 8) use form vorderdetail to display, input (line 9), and add data to dataset orderline (lines 10 to 13) each time we press [[ENTER]]. A new vorderdetail form is put onto the screen for each line of the order. Each form is appended to the preceding form. An example of entering an order with three lines follows:

Figure 5-2. Running the Basic Prototype Program

```
vorderhead                    order data

    order number [123 ]  customer   [1    ]  status  [0 ]   date [850101]
line number [1 ]  part-number [p001    ]  quantity [12      ]
line number [2 ]  part-number [p002    ]  quantity [12
line number [3 ]  part-number [p003    ]  quantity [20      ]
line number [  ]  part-number [        ]  quantity [        ]
```

When we have entered the last order line, pressing [[F8]] gets us back to the point where we can either exit the program or restart it.

Thus with just a small amount of code, we can generate a program that adds data to two datasets. We can use this program to verify with the user that we are creating the correct solution and we can also use this program to enter data that can be used to test other modules within the system.

Perhaps our prototype should have started with a version that could add more than one order to the database. Even so, the first version is important to emphasize how much can be accomplished in Transact with a small amount of code.

A prototyping version which will add more than one order follows:

Figure 5-3. Prototype Program to Add Multiple Master Records

```
1     system ex37,base=orders,vpls=formfile;
1.1   define(item) lastkey i(4):
1.2                enter i(4),init=0;
1.3   list lastkey:
1.4        enter;
2     list(auto) vorderhead;
3     list(auto) vorderline;
3.1   level;
4       get(form) vorderhead,init,freeze;
5       put orderhead,list=(order-no,
6                           cust-no,
7                           order-status,
7.01                        order-date);
8       level;
9         get(form) vorderline,init,append,fkey=lastkey;
9.1       if (lastkey) = (enter)
9.2         then
10             put orderline,list=(order-no,
11                                 line-no,
12                                 part-number,
13                                 quantity)
14           else
15             do
16             set(form) vorderline,clear;
17             end(level);
18             doend;
```

Here we have added an additional level (line 3.1) to our program to control entering each order; we still retain the level that controls adding order lines.

We also added a test to see whether the [[ENTER]] key was last pressed (lines 9.1 to 9.2) or whether one of the function keys was pressed (pressing [[ENTER]] sets 0 to lastkey; pressing any function key sets the number of that key to lastkey). Pressing any of the function keys (lines 14 to 18) is our way of indicating that all order lines have been entered and we want to enter a new order now.

In order to find out what key was pressed by the user, line 9 was modified by adding an option to tell Transact where to put this information and lines 1.1 to 1.4 were added to define the variable lastkey.

As a side note, the variable enter was set up to improve program readability when checking if [[ENTER]] was pressed.

What happens if we enter some invalid data using this prototype? Transact's automatic error handling takes over and redisplays the screen, while asking for new input. It also attempts to let us know what caused the error. For example, cust-no is a numeric field. If non-numeric data is entered, Transact displays an error message in the VPLUS window, pauses for a few seconds to give us the opportunity to read the message, and then redisplays the screen that was in error for us to enter data again. The following screen illustrates that process.

Figure 5-4. Automatic Error Handling with VPLUS

```
vorderhead                      order data

order number [123     ] customer [CUST1] status [0 ] date [850101]|




*ERROR: ENTRY NOT NUMERIC  (USER 1,10)
```

Things are more complicated if an error occurs on form vorderline because it is displayed using the APPEND option. After Transact displays the error message, the new form for data entry is appended to the last form. Thus the line in error remains on the screen and a new empty form for correcting the error is added to the screen. The two example screens below illustrate this point.

Figure 5-5. Automatic Error Handling with VPLUS Append

```
vorderhead                    order data

order number [123     ]   customer [1    ]    status [0 ]   date [850101]|
line number [1 ]  part-number [PART1   ]  quantity [10     ]

*ERROR:THERE IS NO CHAIN HEAD (MASTER ENTRY) FOR PATH 2 (IMAGE 102,32,ORDERLIN
```

Figure 5-6. Automatic Error Handling with VPLUS Append

```
vorderhead                    order data

order number [123     ]   customer [CUST1]    status [0 ]   date [850101]|
line number [1 ]  part-number [PART1   ]  quantity [10      ]
line number [  ]  part-number [        ]  quantity [        ]
```

Up to this point, no order line has been added to the database. The valid line number, part-number, and quantity would be entered into the blank form line.

After you have a good understanding of the VPLUS interface using Transact, you will discover that there are ways to get automatic error handling to do what you want it to, even when using the APPEND and FREEZE options. The following version of our program demonstrates this. No functionality has been added. Only code to make automatic error handling properly display the correct screen format has been added.

Figure 5-7. Functional Prototype with Automatic Error Handling

```
1     system ex38,base=orders,vpls=formfile;
1.1   define(item) lastkey i(4):
1.2                enter i(4),init=0;
1.3   list lastkey:
1.4        enter;
2     list(auto) vorderhead;
3     list(auto) vorderline;
3.01  set(form) vorderhead,init,list=();
3.1   level;
4       get(form) vorderhead,fkey=lastkey;
4.1       if (lastkey) <> (enter)
4.2         then
4.3           do
4.4           set(form) vorderhead,init,list=();
4.5           end(level);
4.6           end;
4.7           doend;
5       put orderhead,list=(order-no,
6                           cust-no,
7                           order-status,
7.01                        order-date);
7.1     set(form) vorderhead,freeze;
7.2     put(form) vorderline,init,list=();
8       level;
9         get(form) vorderline,fkey=lastkey,current;
9.1       if (lastkey) <> (enter)
9.2         then
9.3           do
9.4           set(form) vorderhead,init,list=();
9.5           end(level);
9.6           end;
9.7           doend;
10         put orderline,list=(order-no,
11                             line-no,
12                             part-number,
13                             quantity);
19         set(form) vorderline,append;
20         put(form) vorderline,init,list=();
```

This version delays specifying whether a form is to be frozen or appended to another form until the last possible moment. Recall that our goal is to first display a blank form for entering global order information. If there are any errors in the data, then the form should not be cleared, since that would require the user to re-enter all data.

Once the global information has been entered, then each line of the order is entered. If the data is invalid, the form for that line should not be cleared, but left for correction of the data.

Line 3.01 initially clears form vorderhead. Line 4 has been modified to capture the key pressed by the user. If any of the function keys are pressed, it indicates that no more orders are to be entered. Lines 4.1 to 4.7 perform this function and cause the program to end if the user presses any of [[F1]] through [[F8]].

If the user presses [[ENTER]], Transact will continue to loop through lines 4 to 4.7 until the data entered is valid. When this occurs, line 7.1 freezes the form on the screen so that the forms to enter each order line will appear after this form.

Line 7.2 displays the first form for entering an order line after blanking it out. This is an important step and gets us started in our order line data collection loop.

Line 9 is the important line within this loop. It specifies that we want to work with the form currently displayed on the screen. This is also where the automatic error handling will restart if Transact discovers any input errors. It is this line the prevents the problem discussed in the previous example from happening; that is, it prevents automatic error handling from putting a fresh form on the screen when it encountered data errors.

Lines 9.3 through 9.7 detect that the user has pressed any of [[F1]] through [[F8]] to indicate that all lines have been added for the order. It reinitializes vorderhead and terminates the level.

Lines 19 through 20 are reached after a valid order line has been added to the database. They specify that the next form is to be appended to the screen and put a new blank form on the screen so that the loop can be started again.

So far, we are only letting the automatic error handling detect the entering of non-numeric data into a numeric field and trying to add an entry that is missing a master.

There are additional validation checks that can be handled automatically. The next version of our prototype implements these checks.

Figure 5-8. Prototype with Programmatic Data Validation

```
1     system ex39,base=orders,vpls=formfile;
1.1   define(item) lastkey i(4):
1.2                enter i(4),init=0;
1.3   list lastkey:
1.4        enter;
2     list(auto) vorderhead;
3     list(auto) vorderline;
3.01  set(form) vorderhead,init,list=();
3.1   level;
4       get(form) vorderhead,fkey=lastkey;
4.1     if (lastkey)
(enter)
4.2       then
4.3         do
4.5         end(level);
4.6         end;
4.7         doend;
4.8     set(key) list (order-no);
4.9     get order,list=(),nofind;
4.91    set(key) list (cust-no);
4.92    get customer,list=();
5       put orderhead,list=(order-no,
6                           cust-no,
7                           order-status,
7.01                        order-date);
7.1     set(form) vorderhead,freeze;
7.2     put(form) vorderline,init,list=();
8       level;
9         get(form) vorderline,fkey=lastkey,current;
9.1       if (lastkey)
(enter)
9.2         then
9.3           do
9.4           set(form) vorderhead,init,list=();
9.5           end(level);
9.6           end;
9.7           doend;
9.8       set(key) list (part-number);
9.9       get parts,list=(part-number);
10         put orderline,list=(order-no,
11                             line-no,
12                             part-number,
13                             quantity);
19         set(form) vorderline,append;
20         put(form) vorderline,init,list=();
```

Figure 5-8 will help you see how automatic error handling can work for you.

In this version, lines 4.8 and 4.9 have been added to verify that the order now being entered does not already exist. The NOFIND option on line 4.9 specifies that it is not an error if a record is not found. It is an error if a record is found. If the error occurs, automatic error handling will display a message and restart at the data collection point, which is line 4.

Lines 4.91 and 4.92 verify that the customer already exists. If not, the program is restarted at line 4 after displaying the error message.

Lines 9.8 and 9.9 validate the part number. If the part number does not exist, the program is restarted at the last data entry point which is line 9.

The examples below illustrate the messages generated by the automatic error facility when there are errors, such as:

- order already exists
- customer is invalid
- part number is invalid

Figure 5-9. Automatic Error Handling, Duplicate Record

```
vorderhead                    order data

order number [123  ]   customer [1    ]   status [0 ]   date [850101]|

*ERROR: ENTRY ALREADY EXISTS  (IMAGE 1,23,ORDER)
```

Figure 5-10. Automatic Error Handling on Frozen Screen

```
vorderhead                    order data

order number [124 ]   customer [987  ] status [0 ]   date [850101]|

*ERROR: NO ENTRY FOUND  (IMAGE 17,27,CUSTOMER)
```

Figure 5-11. Automatic Error Handling on Appended Screen

```
vorderhead                    order data

order number [124 ]   customer [1    ]   status [0 ]   date [850101]|
line number [1 ]]  part-number [PART1   ]  quantity [10    ]



*ERROR: NO ENTRY FOUND  (IMAGE 17,55,PARTS)
```

When using this version of the prototype to demonstrate to the users how the system operates, we explain what each particular message means, for example, that we were attempting to add an order for a customer, but the customer is not valid. We could also explain that when the production version of the system is implemented, the error message will appropriately say “invalid customer” and that it will highlight the customer field for data correction.

However, these details do not have to be addressed until the user agrees with the overall system design and flow.

Finally, the user agrees with our system design. The user may also agree to use the system “as is” or he may agree to use the system temporarily until the final version is ready. Most production environments will probably need the additional control in order to make the system more user friendly.

The production version of our prototype follows. The automatic error handling provided by Transact has been replaced with programmatic control and user defined error messages.

Figure 5-12. Production Version of Prototype Program

```
1     system ex40,base=orders,vpls=formfile;
1.1   define(item) lastkey i(4):
1.2                enter i(4),init=0;
1.21  define(item) valid i(4):
1.22               yes i(4),init=1:
1.23               no i(4),init=0;
1.24  list valid:
1.25       yes:
1.26       no;
1.3   list lastkey:
1.4        enter;
2     list(auto) vorderhead;
3     list(auto) vorderline;
3.1   level;
3.2     set(form) vorderhead,init,list=();
3.3     repeat
3.4       do
3.5       let (valid) = (yes);
4         get(form) vorderhead,fkey=lastkey;
4.1       if (lastkey)
(enter)
4.2         then
4.3           do
4.5           end(level);
4.6           end;
4.7           doend;
4.8       set(key) list (order-no);
4.9       find order,list=();
4.901     if status > 0
4.902       then
4.903         do
4.904         set(form) vorderhead,window=(order-no,"order already
exists");
4.905         let (valid) = (no);
4.906         doend;
```

Figure 5-13. Production Version of Prototype Program (Continued)

```
4.92      find customer,list=();
4.93      if status = 0
4.94        then
4.95          do
4.96          set(form) vorderhead,window=(cust-no,"customer does not exist");
4.97          let (valid) = (no);
4.98          doend;
4.99      doend
4.991   until (valid) = (yes);
5       put orderhead,list=(order-no,
6                           cust-no,
7                           order-status,
7.01                        order-date);
7.1     set(form) vorderhead,freeze;
7.2     set(form) vorderline,init,list=();
8       level;
8.1       repeat
8.2         do
8.3         let (valid) = (yes);
9           get(form) vorderline,fkey=lastkey;
9.1         if (lastkey)
(enter)
9.2           then
9.3             do
9.5             end(level);
9.6             end;
9.7             doend;
9.8         set(key) list (part-number);
9.9         find parts,list=(part-number);
9.91        if status = 0
9.92          then
9.93            do
9.94            set(form) vorderline,
9.95                      window=(part-number,"invalid part number");
9.96            let (valid) = (no);
9.97            doend;
9.98        if (quantity) <= 0
9.99          then
9.991           do
9.992           set(form) vorderline,
9.993                     window=(quantity,"must be > 0");
9.994           let (valid) = (no);
9.995           doend;
9.996       doend
9.997     until (valid) = (yes);
10         put orderline,list=(order-no,
11                             line-no,
12                             part-number,
13                             quantity);
19         set(form) vorderline,append,init,list=();
```

The main thing done to this version is to replace the automatic looping on data errors until the data is valid with explicit programmatic looping until the errors are corrected.

Lines 1.21 to 1.26 define new variables to programmatically detect whether a data entry form contains valid data or not. Valid can be viewed as a switch which is either yes or no depending on whether the data entered is valid or not.

Lines 3.3, 3.4, 4.99, and 4.991 set up the boundaries of a loop that is executed until the data entered is valid.

Within the loop, the valid flag will be set to indicate the data is invalid if an error is found. However, before any data validation is done, line 3.5 sets the default for the flag to be that the data is valid.

Line 4.9 changes the verb of the previous prototype from GET to FIND. GET was useful when depending on the automatic error handling facility, because when we use GET, Transact assumes that we know the data we want either exists or doesn't exist. Therefore it is an error if the opposite condition occurs. When we use FIND though, Transact does not assume that we know whether the data exists or not. In effect we are asking whether it does exist or not. Therefore, it is not automatically an error if the data does not exist.

FIND tells us how many records it found by putting the number of records in the system variable called STATUS. Lines 4.901 to 4.906 detect whether the user is attempting to add an order that already exists. If so, line 4.904 uses the WINDOW option to highlight the order-no field as being in error and to display the error message “order already exists” in the VPLUS form window.

Lines 4.92 to 4.98 perform a similar validity check on the cust-no field. The only difference is that the error occurs if there is no existing customer record.

Lines 8.1, 8.2, 9.996, and 9.997 set up a loop that will repeat until the data for an order line item is valid.

Lines 9.91 to 9.97 validate the part-number.

Lines 9.98 to 9.995 perform an additional validation on the order quantity which we could not do using automatic error handling.

6 Data Structures

So far, the examples have made little reference to Transact's data structures. However, if you go back and scan each example, you will find that some form of the LIST verb exists in each example. For many applications, what you see in the examples is all that is needed. However, there are times when it is necessary to programmatically take more control over Transact's temporary data storage.

COBOL and Pascal have very well defined data structures. This chapter will compare Transact's data structures to those used by both of these languages.

Typically, high-level application development products have data structures that are not well defined. For many applications, this makes them very easy to work with to generate reports and to update databases or files. By the same token, there are applications which become, if not impossible, then extremely difficult to implement with these products because of their weak data structures.

Let's start by using some lines from EX40 in Figure 5-12.

Figure 6-1. Transact Data Structures

```
1.1   define(item) lastkey i(4):
1.2                enter i(4),init=0;
1.21  define(item) valid i(4):
1.22               yes i(4),init=1:
1.23               no i(4),init=0;
1.24  list valid:
1.25       yes:
1.26       no;
1.3   list lastkey:
1.4        enter;
2     list(auto) vorderhead;
3     list(auto) vorderline;
3.1   level;
```

This same definition in COBOL might be:

Figure 6-2. Comparable COBOL Data Structures

```
1     01 valid pic 9(4) comp.
2     01 yes pic 9(4) comp value 1.
3     01 no pic 9(4) comp value 0.
4     01 lastkey pic 9(4) comp.
5     01 enter pic 9(4) comp value 0.
6     01 ws-vorderhead.
7         02 order-no pic x(8).
8         02 cust-no pic 9(4).
9         02 order-status pic x(2).
10        02 order-date pic x(6).
11     01 ws-vorderline.
12        02 line-no pic 9(2).
13        02 part-number pic x(8).
14        02 quantity pic 9(6) comp.
```

And in Pascal:

Figure 6-3. Comparable PASCAL Data Structures

```
1     const yes=1;
2           no=0;
3           enter=0;
4     type small_int=-32768..32767;
5          char2=packed array[1..2] of char;
6          char4=packed array[1..4] of char;
7          char6=packed array[1..6] of char;
8          char8=packed array[1..8] of char;
9          typ_vorderhead=record
10                           order_no:char8;
11                           cust_no:char4;
12                           order_status:char2;
13                           order_date:char6;
14                         end;
15          typ_vorderline=record
16                           line_no:char2;
17                           part_number:char8;
18                           quantity:integer;
19                         end;
20     var valid:small_int;
21         lastkey:small_int;
22         ws_vorderhead:typ_vorderhead;
23         ws_vorderline:typ_vorderline;
```

NO and ENTER are reserved words in COBOL. These variable names would have to be changed, but that is unimportant to our discussion.

The Pascal example is not exactly the same as the Transact or COBOL example, since Pascal does not have the direct equivalent of an ASCII numeric data type as COBOL and Transact do. However, that is not important. The important part is being able to compare how storage is reserved and how access is gained to it.

In Transact, an item is defined one time. This definition is either done using a data dictionary or within the program. In either case, when space is reserved for the item, the definition part is not included as it is for COBOL and Pascal.

In the Transact example, DEFINE(ITEM) (lines 1.1 to 1.23) defines the name, format, and size of five data items. All five are of single word integer or binary format. In addition, whenever storage space is reserved for enter, yes, or no, the space is initialized to contain the values 0, 1, or 0, respectively. Note that DEFINE does not reserve space for the data items. It more closely resembles the Pascal TYPE construct.

Lines 1.24 to 1.4 actually reserve space or make the data items known to Transact.

LIST(AUTO) in lines 2 and 3 also reserves space for data items. However, in this case the data items have been previously defined in Dictionary/V. Also the VPLUS form names vorderhead and vorderline have been defined in Dictionary/V. These lines are equivalent to the following:

Figure 6-4. LIST(AUTO) Equivalent with LIST

```
2     list order-no:          |
|     2.1        cust-no:          |
|     2.2        order-status:     |
|     2.3        order-date;       |
|     3     list line-no:          |
|     3.1        part-number:      |
|     3.2        quantity;
```

Later on in the program there are verbs that input data from the VPLUS screens and other verbs that update data in an IMAGE dataset. Examples from the program are:

Figure 6-5. VPLUS Default with no LIST=

```
9           get(form) vorderline,fkey=lastkey;
10         put orderline,list=(order-no,
11                             line-no,
12                             part-number,
13                             quantity);
```

Line 9 inputs data to the program from the VPLUS screen vorderline. There is nothing on this line to indicate which data items to retrieve. In this default case, Transact looks at the form definition in Dictionary/V to determine which data items are involved. If we wanted to be explicit, we could have made this line read:

Figure 6-6. VPLUS Explicit LIST=

```
9     get(form) vorderline,fkey=lastkey
9.01                      ,list=(line-no,
9.02                             part-number,
9.03                             quantity);
```

or alternatively:

Figure 6-7. LIST= Item Range

```
9     get(form) vorderline,fkey=lastkey
9.01                      ,list=(line-no:
9.02                             quantity);
```

to indicate a range of data items.

Lines 10 to 13 add a new record to dataset orderline using data items order-no, line-no, part-number, quantity.

Figure 6-8. IMAGE Explicit Item List

```
10         put orderline,list=(order-no,
11                             line-no,
12                             part-number,
13                             quantity);
```

Transact does not automatically figure out which items should be used for IMAGE. If we omit the LIST= option, Transact assumes that all items currently known to the program (via LIST) are to be used. In fact, this is also true of MPE and KSAM files.

However, the items have to be contiguous in temporary storage if LIST= is not included. Since in our example they are not contiguous, we need to specify LIST=.

COBOL also has the facility to redefine storage. We have already seen one example above. In the COBOL program example, by referencing the variable ws-vorderhead, a COBOL program would perform a group level operation on order-no, cust-no, and order-status. The COBOL program could also reference each of these data items individually.

Another COBOL example is the following:

Figure 6-9. COBOL Redefinition of Data Storage

```
1     01 record-data.
2        02 dte pic 9(6).
3        02 date-redef redefines dte.
4          04 yy pic 99.
5          04 mm pic 99.
6          04 dd pic 99.
```

This same example in Transact is handled as follows:

Figure 6-10. Transact Redefinition of Data Storage

```
1     define(item) date 9(6):
2                  yy 9(2)=date:
3                  mm 9(2)=date(3):
4                  dd 9(2)=date(5);
5     list date;
```

In Transact, these items are referred to as parent and child items. A child item as in lines 2 to 4 is equated to a byte offset of the parent item. If no offset is specified, it defaults to the start of the parent item. Although our program may make a reference to yy, this child item is never used in the LIST. Space must be reserved for the parent item which is date.

Lists or arrays are another common data structure. In COBOL, the following is a segment of a program containing both a one and two dimensional array.

Figure 6-11. COBOL Array Definitions

```
1     01 empl-table.
2       02 empl-rec occurs 10.
3         04 empl-no pic x(6).
4         04 empl-name pic x(30).
5         04 empl-salary pic 9(8)v99 comp.
6
7     01 reg-sales-by-mo.
8       02 region-line occurs 10.
9         04 region pic x(4).
10         04 month pic 9(8) comp occurs 12.
```

This would be implemented in Transact as follows:

Figure 6-12. Comparable Transact Array Definitions

```
1     define(item) empl-table 10 x(40):
2                    empl-rec x(40)=empl-table:
3                      empl-no x(6)=empl-rec:
4                      empl-name x(30)=empl-rec(7):
5                      empl-salary i(10,2)=empl-rec(37):
6                  reg-sales-by-mo 10 x(52):
7                    region-line x(52)=reg-sales-by-mo:
8                      region x(4)=region-line:
9                      month-data 12 i(8)=region-line(5):
10                        month i(8)=month-data;
11     list empl-table:
12          reg-sales-by-mo;
```

Line 1 defines the total number of occurrences (10) and total byte length of each occurrence (40) of the one-dimensional array.

Line 2 defines one occurrence of employee data to be 40 bytes long. In a later section, we will see how to access array items by subscripting or using the LET(OFFSET) verb.

Lines 3-5 define detail items making up one employee record. The total byte length of all these fields adds up to the 40 bytes that makes up one record.

Line 6 starts a new array definition.

Line 9 defines a second dimension of this array. The first dimension holds the data for 10 regions. The second dimension is made up of 12 months of data for each region.

As we have seen, Transact does have definite data structures which in many respects correlate quite closely with those in COBOL.

Transact works best when you keep in mind that you only want to add new information to your data structure, not define anything twice. For example, the orders database has several sets which contain the item part-number. Two of these sets are inventory and parts. If a program needs to have access to all information contained in both of these sets, it should only list each item one time. Since part-number is common to both sets, it should only be listed once. To illustrate, if a program first retrieves an inventory record and then needs to retrieve the corresponding parts record to get the part description, a good way to do this is:

Figure 6-13. LISTing Items From Multiple Datasets

```
1     list part-number:
2          location:
3          quantity:
4          description;
5     get(serial) inventory,list=(part-number:quantity);
6     set(key) list (part-number);
7     get parts,list=(description);
```

This technique can be used when an item in two or more datasets refers to the same data. However, there are times when two datasets or files contain the same item name, but they are independent of each other. For example, the database we have been using for examples has two datasets that contain an item called quantity. Even though these data items share a common name, the meaning is quite different for each set. In the inventory set, quantity is the inventory at a particular location. In the orderline set, quantity is the quantity ordered on this line of an order.

Perhaps we need to generate a report which lists all part numbers ordered, the order quantity, and the total inventory on hand. The following program is one way to do this.

Figure 6-14. Use of ALIAS= for Items with Same Name

```
1     system ex54,base=orders;
2     define(item) inv-quantity i(6),alias=(quantity(inventory))
3                  order-quantity i(6),alias=(quantity(orderline))
4                   tot-inv i(6),head="inventory";
5     list part-number:
6          inv-quantity:
7          order-quantity:
8          tot-inv;
9     find(serial) orderline,list=(part-number,order-quantity)
10                             ,perform=100-get-inv;
11     end;
12
13     100-get-inv:
14
15       set(key) list (part-number);
16       let (tot-inv) = 0;
17       find(chain) inventory,list=(inv-quantity)
18                             ,perform=110-accum-inv;
19       display(table) part-number:
20                      tot-inv:
21                      order-quantity;
22       return;
23
24     110-accum-inv:
25
26       let (tot-inv) = (tot-inv) + (inv-quantity);
27       return;
```

The above program is able to create a unique identifier for each type of quantity by using the ALIAS= option. Within the program, the item is always referenced by the first name in the DEFINE. The ALIAS item name identifies the item name and dataset as they are known to IMAGE.

Another way to solve this problem is to use the dynamic feature of Transact's data structures. The data known to a Transact program can be redefined or remapped at any

time during program execution. Unlike COBOL and Pascal, Transact does not map physical data storage at compile time. Storage is allocated at run time as the program processes LIST verbs.

There are also verbs to deallocate storage when it is no longer needed or when it needs to be remapped. These verbs are illustrated in the program below which solves the same problem as the program shown in Figure 6-14.

Figure 6-15. Use of SET(STACK) LIST

```
1     system ex55,base=orders;
4     define(item) tot-inv i(6),head="inventory";
5     list part-number:
6          quantity:
8          tot-inv;
9     find(serial) orderline,list=(part-number,quantity)
10                            ,perform=100-get-inv;
11     end;
12
13     100-get-inv:
14
14.1     list quantity;
15       set(key) list (part-number);
16       let (tot-inv) = 0;
17       find(chain) inventory,list=(quantity)
18                            ,perform=110-accum-inv;
18.1     set(stack) list (tot-inv);
19       display(table) part-number:
20                      tot-inv:
21                      quantity,head="order-quantity";
22       return;
23
24     110-accum-inv:
25
26       let (tot-inv) = (tot-inv) + (quantity);
27       return;
```

In this example program, the data storage always exists in one of two forms. When lines 15 to 18 and 26 to 27 are executed, the data storage looks like:

Figure 6-16. LIST Register Map with Same Item Twice

```
5      part-number
6      quantity
8      tot-inv
14.1    quantity
```

During the remainder of the program, the data storage looks like:

Figure 6-17. LIST Register After SET(STACK)

```
5      part-number
6      quantity
8      tot-inv
```

This is all controlled by the repeated execution of lines 14.1 and 18.1. Line 14.1 reserves space for the item quantity. It will hold the inventory quantity at a location while line 17 is executing. Line 18.1 deallocates this space after we have computed the total inventory balance.

How does Transact know which quantity we want to use in line 26? It doesn't. Whenever a program instructs Transact to perform some action on a data item, Transact always uses the most recently defined (LISTed) version of the item. Once line 14.1 has been executed, there is no way we can ever access the space reserved for quantity by lines 5 to 7 until we execute line 18.1.

In the example above, we could also have written line 18.1 as:

Figure 6-18. Removing the Last Item Name from the LIST

```
                    18.1     set(stack) list (*);
```

This deallocates the last data item defined (LISTed) to Transact.

A slightly more general purpose way to implement the last example is to use marker items. Just as their name implies, these items mark a data storage reference point that we can go back to. Or they can be used in pairs to delimit a set of data that pertains to a file or dataset. A marker is just a name and does not physically take up any data storage space.

In the last example we can take advantage of both usages of marker items as follows:

Figure 6-19. Use of Marker Items

```
1     system ex59,base=orders;
4     define(item) tot-inv i(6),head="inventory";
4.1   define(item) begin-orderline @:
4.2                end-orderline @;
4.3   list tot-inv;
5     list begin-orderline:
5.1        part-number:
6          quantity:
7          end-orderline;
9     find(serial) orderline,list=(begin-orderline:end-orderline)
10                            ,perform=100-get-inv;
11     end;
12
13     100-get-inv:
14
14.1     list quantity;
15       set(key) list (part-number);
16       let (tot-inv) = 0;
17       find(chain) inventory,list=(quantity)
18                            ,perform=110-accum-inv;
18.1     set(stack) list (end-orderline);
19       display(table) part-number:
20                      tot-inv:
21                      quantity,head="order-quantity";
22       return;
23
24     110-accum-inv:
25
26       let (tot-inv) = (tot-inv) + (quantity);
27       return;
```

Lines 4.1 and 4.2 define the marker items. Lines 5 and 7 put the markers around the items that we want to use from the orderline set. Line 9 now uses the markers to indicate the data item range to be used. With this implemented, at some later date we could decide that the program also needs to retrieve order-no from the orderline set and no changes would have to be made to line 9. Order-no would only need be added to the LIST verb between lines 5 and 7.

In line 18.1, we can now use end-orderline as a reference to reset to and thus make the section of code that sums the part inventory independent of the code that retrieves order information.

Transact's dynamic data storage can be illustrated by another example. However, keep in mind that even though data storage is dynamic, it is not boundless. This kind of program is fun to write, but it is not the example that you want to base your future Transact development on. In fact, this example should be viewed simply as demonstration of Transact's dynamic storage capabilities.

Figure 6-20. Illustration of Dynamic Data Storage

```
 1     system ex60,base=orders;
2     define(item) temp-part x(8):
3                  from-part x(8):
4                  to-part x(8);
5     define(item) count i(4),init=0:
6                  dun x(4),init="no":
7                  yes x(4),init="yes":
8                  no x(4),init="no";
9     define(item) parts-this-pass i(4):
10                  no-of-parts i(4);
11     define(item) start-of-parts @;
12     list temp-part:
13          parts-this-pass:
14          no-of-parts;
15     list count:
16          dun:
17          yes:
18          no;
19     list start-of-parts;
20     repeat
21       do
22       list part-number;
23       get(serial) parts,list=(part-number),status;
24       if status = 0
25         then let (count) = (count) + 1
26         else let (dun) = (yes);
27       doend
28     until (dun) = (yes);
29     let (no-of-parts) = (count);
30     let (parts-this-pass) = (no-of-parts);
31     while (parts-this-pass) > 0
32       perform 100-bubble-sort;
```

Figure 6-21. Illustration of Dynamic Data Storage (Continued)

```
33     set(stack) list (start-of-parts);
34     let (count) = 0;
55     while (count) < (no-of-parts)
36       do
37       list part-number;
38       let (count) = (count) + 1;
39       doend;
40     let (count) = (no-of-parts);
41     while (count) > 0
42       do
43       display(table) part-number;
44       set(stack) list (*);
45       let (count) = (count) - 1;
46       doend;
47     end;
48
49     100-bubble-sort:
50
51       set(stack) list (start-of-parts);
52       let (count)= 1;
53       while (count) < (parts-this-pass)
54         do
55         let (count) = (count) + 1;
56         list from-part:
57              to-part;
58         if (from-part) < (to-part)
59           then
60             do
61             move (temp-part) = (to-part);
62             move (to-part) = (from-part);
63             move (from-part) = (temp-part);
64             doend;
65           set(stack) list (*);
66         doend;
67       let (parts-this-pass) = (parts-this-pass) - 1;
68       return;
```

The above program performs a bubble sort of valid part-numbers. You will never want to do this, since it is much easier to let Transact do the sort for you via the SORT=(part-number) option, but it does illustrate the dynamics of Transact's data structures.

Lines 20 to 28 serially read the master set that contains part-numbers and dynamically adds each to the program data storage. Line 22 reserves an additional 8 bytes of storage each time it is executed.

Lines 29 to 31 and 49 to 68 perform the bubble sort. The idea of a bubble sort is that for each pass over the data, the lowest key item is floated to the top. Also, the second pass over the data knows that the first pass found the lowest value item, so it only has to look at all data except the one at the top (which is already lowest).

No-of-parts is the count of the total number of items. Parts-this-pass is the total number of items that need to be scanned during this pass of the bubble sort. Count is just a counter that is reinitialized and reused as an index through the data storage throughout the program.

Line 30 sets things up for the first pass of the sort. Line 67 sets things up for the next pass of the bubble sort after the previous pass has been completed.

Lines 31 and 32 control the passes of the sort.

Line 51 makes use of a marker to set the data storage pointer to the start of the part numbers. Lines 53 to 66 increment through the part numbers, comparing one entry to the next. The integral part of this loop is lines 56, 57, and 65. Lines 56 and 57 name or remap the next two parts to be compared. Line 65 backs the program up one item so that the next loop will compare the winner of the last compare with the next item.

Lines 58 to 64 perform the compare of the next two parts and when completed, the lowest value part-number has been floated up. It is then compared to the next item during the next loop.

After the bubble sort is completed, lines 33 to 39 remap the data storage to just contain the name part-number. This needs to be done because the bubble sort routine kept calling items from-part and to-part.

Lines 40 to 46 start at the top and print out the value of each part-number in ascending order.

7 Using Transact Without a Dictionary

Transact can be used with either Dictionary/3000 or System Dictionary. It can also be used without a dictionary and can still interface with IMAGE, MPE, KSAM, and VPLUS.

This section uses examples from previous sections, modifying them so that they work independent of the dictionary. All that has to be done is to replace the common data definitions in the dictionary with program data definitions. These examples are shown below with the changes highlighted. For the most part, no additional discussion is needed.

IMAGE

The following program shows the modifications made to the program listed in Figure 1-17 in order to make it independent of the dictionary.

Figure 7-1. IMAGE Access Without the Dictionary

```
 1     system ex61,base=orders;
2     define(item) order-no x(8):
2.1                cust-no 9(4):
2.2                order-status x(2):
2.21               order-date x(6):
2.3                name x(20):
2.4                street-addr x(20):
2.5                city-state x(20):
2.6                zipcode x(6):
2.61               line-no 9(2):
2.62               part-number x(8):
2.63               quantity i(6);
2.7   list order-no:
2.8        cust-no:
2.9        order-status:
2.91       order-date;
3     move (order-status) = "o";
4     set(match) list (order-status);
5     find(serial) orderhead,list=(order-no:order-date)
6                           ,perform=get-orderdata;
6.1   exit;
```

Figure 7-2. IMAGE Access Without the Dictionary (Continued)

```
 8     get-orderdata:
 9
 9.01  level;
 9.1     set(key) list (cust-no);
 9.11  list name:
 9.12       street-addr:
 9.13       city-state:
 9.14       zipcode;
 9.2     get customer,list=(name:zipcode);
 10       set(key) list (order-no);
 10.01    list line-no:
 10.02         part-number:
 10.03         quantity;
 11       find(chain) orderline,list=(line-no:quantity)
 11.01                           ,perform=displayit;
 11.1   end(level);
 12     return;
 13
 14     displayit:
 15
 16     display(table) name:
 17                    order-no:
 18                    line-no:
 19                    part-number:
 20                    quantity;
 21     return;
```

Lines 2 to 2.63 provide the data item definition normally done centrally in the dictionary.

Lines 2.7 to 2.9, 9.11 to 9.14, and 10.01 to 10.03 replace the LIST(AUTO) construct which automatically extracts the record definition from the dictionary.

Lines 5,9.2, and 11 are modified to replace the LIST=(@) with an item range list.

MPE

The following program is a modification of the program listed in Figure 4-5. The items are explicitly defined in lines 1.1 to 1.3, and LIST(AUTO) is replaced with a list of the items.

Figure 7-3. MPE Access Without the Dictionary

```
1     system ex62,file=batchinv(update);
1.1   define(item) part-number x(8):
1.2                location x(4):
1.3                quantity i(6);
2     list part-number:
2.1         location:
2.2         quantity;
3     level;
4       data part-number:
5            location:
6            quantity;
7       put batchinv;
```

KSAM

The following program is a modification of the program listed in Figure 4-3. Changes are the same as for the preceding two programs.

Figure 7-4. KSAM Access Without the Dictionary

```
1     system ex63,ksam=kcust(update);
2     set(delimiter) "/";
2.1   define(item) cust-no 9(4):
2.2                name x(20);
3     list cust-no:
3.1        name;
4     level;
5       data cust-no;
6       set(key) list (cust-no);
7       get kcust;
8       display;
9       data name;
10     set(update) list (name);
11     replace kcust;
12     end;
```

VPLUS

For forms access without a dictionary, the form and its fields must be spelled out in the SYSTEM statement, then the fields must be defined. The following program is a modification of the program listed in Figure 3-3.

Figure 7-5. VPLUS Access Without the Dictionary

```
1     system ex64,base=orders,vpls=formfile(vcustomer(cust-no,name,
1.1                                 street-addr,city-state,zipcode));
1.2   define(item) cust-no 9(4):
1.3                name x(20):
1.4                street-addr x(20):
1.5                city-state x(20):
1.6                zipcode x(6);
2     $$add:
4     list cust-no:
4.1        name:
4.2        street-addr:
4.3        city-state:
4.4        zipcode;
5     get(form) vcustomer,init;
6     put customer;
```

Lines 1 and 1.1 replace the form and data item information that is normally obtained from the dictionary.

8 Special Topics

Interface to Report/V

In previous sections, we looked at examples of using Transact as a report generator. These examples illustrated features of Transact but were not meant to indicate that one of the strengths of Transact is its report writing capability.

In fact, Transact is not meant to be a report writer. Although some high level benefits are gained, most reporting controls, like sort control breaks and subtotals, still have to be coded in Transact. This is where Report/V can help out.

Report/V is a report writer. It can be run as a free standing program and most times this is the way it is run.

However for complex reporting needs, combining Transact and Report/V will solve the problem. Transact can be used to extract the data and do the data manipulation creating a file which is then given to Report/V to generate the report.

The following example illustrates a typical Transact solution to a complicated reporting requirement. For those of you who use report writers, think about what you would have to do if you needed to prepare a report like this. If the report writer you are familiar with could prepare this report, the chances are it would consist of several intermediate passes over the data before the final report could be generated.

More typically, you would have to do the data extraction and manipulation using a language like COBOL and would probably let COBOL create the report, since interfacing to a report writer at that point would be difficult.

The example consists of preparing a backorder report. Our database maintains inventory of each part-number by location code. Thus it is possible that there are many records to add up to compute the total inventory on hand for a part. Likewise, there may be many orders for the same part-number. For illustration purposes, inventory is arbitrarily allocated to orders based upon the order-date. Thus, the backlog report will identify which orders or parts of an order can be filled from inventory and which can only be filled from future production.

Assume that the database contains the following inventory and order information.

Figure 8-1. Part Number Balances by Location

```
                INVENTORY DETAILS
            PART-NUMBER LOCATION QUANTITY
            ------------------------------
            PART1       LOC1         100
                        LOC2         200
                        LOC3         300
            PART2       LOC1         100
            PART3       LOC2         200
```

Figure 8-2. Part Number Open Orders

```
                          ORDER DETAILS
      ORDER-NO CUST-NO ORDER-DATE LINE-NO PART-NUMBER QUANTITY
  ------------------------------------------------------------
  ORDER1   1000    850101     10      PART1           100
                              20      PART2           200
                              30      PART3           300

  ORDER2   2000    850102     10      PART3           200
                              20      PART2           300
                              30      PART1           400

  ORDER3   1000    850103     10      PART1           300
                              20      PART2           400
                              30      PART3           500
```

The report looks like this:

Figure 8-3. Backlog Detail by Customer and Part

BACKLOG DETAIL BY CUSTOMER AND PART

CUSTOMER	PART NUMBER	ORDER	DATE	LINE	ORDERED	BACKORDERED
1000	PART1	ORDER1	85/01/01	10	100	
1000	PART2				600	500
	PART3	ORDER1	85/01/01	30	300	100
		ORDER3	85/01/03	30	500	500
1000	PART3				800	600
1000	TOTAL				1,800	1,300
2000	PART1	ORDER2	85/01/02	30	400	
2000	PART1				400	
	PART2	ORDER2	85/01/02	20	300	300
2000	PART2				300	300
2000	PART3	ORDER2	85/01/02	10	200	200
2000	PART3				200	200
2000	TOTAL				900	500
GRAND TOTAL					2,700	1,800

Note that the inventory was applied to the oldest orders based on order-date.

The Transact and Report/V program that generated this report follow.

Figure 8-4. Transact Program to Create Backlog Report

```
 1     system ex65,base=orders,file=shortage(update)
 2                            ,file=temp(sort);
 3     define(item) tot-inv i(6):
 4                  inv-part x(8):
 5                  inv-quantity i(6),alias=(quantity(inventory));
 6     list part-number:
 7          cust-no:
 8          order-date:
 9          order-no:
10          line-no:
11          quantity:
12          back-order:
13          tot-inv:
14          inv-part:
15          inv-quantity;
16 find(serial) orderline,list=(part-number,order-no,line-no,quantity
17                         ,perform=each-orderline;
18     find(serial) temp,list=(part-number:quantity)
19                       ,sort=(part-number,order-date)
20                       ,perform=each-temp-orderline;
21     call ex65r,report;
22     exit;
23
24     each-orderline:
25
26       set(key) list (order-no);
27       find(chain) orderhead,list=(cust-no,order-date);
28       put temp,list=(part-number:quantity);
29       return;
```

Figure 8-5. Transact Program to Create Backlog Report (Continued)

```
31     each-temp-orderline:
32
33       if (part-number) <> (inv-part)
34         then perform each-inv-part;
35       let (tot-inv) = (tot-inv) - (quantity);
36       if (tot-inv) < 0
37         then
38           do
39           let (back-order) = 0 - (tot-inv);
40           let (tot-inv) = 0;
41           doend
42         else let (back-order) = 0;
43   put shortage,list=(cust-no,part-number,order-no,order-date,
44                      line-no,quantity,back-order);
45       return;
46
47     each-inv-part:
48
49       move (inv-part) = (part-number);
50       set(key) list (part-number);
51       let (tot-inv) = 0;
52       find(chain) inventory,list=(inv-quantity)
53                             ,perform=accum-inv;
54       return;
55
56     accum-inv:
57
58       let (tot-inv) = (tot-inv) + (inv-quantity);
59       return;
```

Figure 8-6. Report/V Program to Create Backlog Report

```
 1     report ex65r;
 2     option nohead;
 3     access shortage,list=(cust-no,part-number,order-no,order-date,
 4                           line-no,quantity,back-order);
 5     sort(1) cust-no:part-number;
 6     page heading "BACKLOG DETAIL BY CUSTOMER AND PART",col=25:
 7                  "CUSTOMER",LINE=2,COL=1:
 8                  "PART NUMBER",COL=10:
 9                  "ORDER",COL=22:
10                  "DATE",COL=32:
11                  "LINE",COL=42:
12                  "ORDERED",COL=50:
13                  "BACKORDERED",COL=70:
14                  " ",line=1;
15     detail cust-no,col=1:
16            part-number,col=10:
17            order-no,col=22:
18            order-date,col=32,edit="^^^^/^^^^/^^^^":
19            line-no,col=43:
20            quantity,col=50,edit="ZZZ,ZZZ":
21            back-order,col=70,edit="ZZZ,ZZZ";
22     group(2) summary cust-no,col=1:
23                      part-number,col=10:
24                      total(quantity),col=50,edit="ZZZ,ZZZ":
25                      total(back-order),col=70,edit="ZZZ,ZZZ":
26                      " ",line=1;
27     group(1) summary cust-no,col=1:
28                      "TOTAL",col=10:
29                      total(quantity),col=50,edit="ZZZ,ZZZ":
30                      total(back-order),col=70,edit="ZZZ,ZZZ":
31                      " ",line=1;
32     report   summary "GRAND TOTAL",col=1:
33                      total(quantity),col=50,edit="ZZZ,ZZZ":
34                      total(back-order),col=70,edit="ZZZ,ZZZ";
```

The Transact program extracts the data desired and writes it to an MPE file called shortage. When it is finished, it calls the report program in line 21.

That's all there is to it. There is a similar interface between Transact and Inform. If the report were defined using Inform, line 21 would become:

```
call ex65r,inform;
```

The example above took advantage of Dictionary/V by defining the file shortage in the dictionary. When interfacing with Report/V, the file does not have to be defined in the dictionary nor do the data elements that are contained in the file have to be defined in the dictionary.

We have already seen how to use Transact without the dictionary. If the data items were defined in the dictionary, but the file was not, there would not be any changes required to run the above program.

If the data items were not defined in the dictionary, then the report program would have to contain the item definitions using DEFINE(ITEM). The resulting Report/V program is shown below.

Figure 8-7. Using Report/V Without the Dictionary

```
 1     report ex65r;
 2     option nohead;
 2.1   define(item) cust-no 9(4):
 2.2                part-number x(8):
 2.3                order-no x(8):
 2.4                order-date x(6):
 2.5                line-no 9(2):
 2.6                quantity i(6):
 2.7                back-order i(6);
 3     access shortage,list=(cust-no,part-number,order-no,order-date,
 4                           line-no,quantity,back-order);
 5     sort(1) cust-no:part-number;
 6     page heading "BACKLOG DETAIL BY CUSTOMER AND PART",col=25:
 7                  "CUSTOMER",LINE=2,COL=1:
 8                  "PART NUMBER",COL=10:
 9                  "ORDER",COL=22:
10                  "DATE",COL=32:
11                  "LINE",COL=42:
12                  "ORDERED",COL=50:
13                  "BACKORDERED",COL=70:
14                  " ",line=1;
15     detail cust-no,col=1:
16            part-number,col=10:
17            order-no,col=22:
18            order-date,col=32,edit="^^^^/^^^^/^^^^":
19            line-no,col=43:
20            quantity,col=50,edit="ZZZ,ZZZ":
21            back-order,col=70,edit="ZZZ,ZZZ";
22     group(2) summary cust-no,col=1:
23                      part-number,col=10:
24                      total(quantity),col=50,edit="ZZZ,ZZZ":
25                      total(back-order),col=70,edit="ZZZ,ZZZ":
26                      " ",line=1;
27     group(1) summary cust-no,col=1:
28                      "TOTAL",col=10:
29                      total(quantity),col=50,edit="ZZZ,ZZZ":
30                      total(back-order),col=70,edit="ZZZ,ZZZ":
31                      " ",line=1;
32     report   summary "GRAND TOTAL",col=1:
33                      total(quantity),col=50,edit="ZZZ,ZZZ":
34                      total(back-order),col=70,edit="ZZZ,ZZZ";
```

Arrays

It is possible to set up multi-dimensional arrays in Transact. We will look at examples of one- and two-dimensional arrays. The bubble sort example illustrated in Figure 6-20 can be implemented replacing the dynamic part-number data structure with a static list or one-dimensional array of part numbers. The resulting program might look like this:

Figure 8-8. One Dimensional Array

```
 1     system ex66,base=orders;
 2     define(item) temp-part x(8):
 2.1                part-table 20 x(8):
 2.2                each-part x(8)=part-table:
 3                  from-part x(8)=part-table:
 4                  to-part x(8)=part-table(9);
 5     define(item) count i(4),init=0:
 6                  dun x(4),init="no":
 7                  yes x(4),init="yes":
 8                  no x(4),init="no";
 9     define(item) parts-this-pass i(4):
10                  no-of-parts i(4);
11     <<deleted>>
12     list temp-part:
12.1        part-number:
12.2        part-table:
13          parts-this-pass:
14          no-of-parts;
15     list count:
16          dun:
17          yes:
18          no;
19     <<deleted>>
20     repeat
21       do
22       <<deleted>>
23       get(serial) parts,list=(part-number),status;
24       if status = 0 then
24.1       do
25         let (count) = (count) + 1;
25.1       move (each-part((count))) = (part-number);
25.2       doend
26       else let (dun) = (yes);
27       doend
```

Figure 8-9. One Dimensional Array (Continued)

```
28      until (dun) = (yes);
29      let (no-of-parts) = (count);
30      let (parts-this-pass) = (no-of-parts);
31      while (parts-this-pass) > 0
32        perform 100-bubble-sort;
33      <<deleted 33-39>>
40      let (count) = (no-of-parts);
41      while (count) > 0
42        do
43        display(table) each-part((count)),head="part-number";
44        <<deleted>>
45        let (count) = (count) - 1;
46        doend;
47      end;
48
49      100-bubble-sort:
50
51        <<deleted>>
52        let (count)= 1;
53        while (count) < (parts-this-pass)
54          do
55          <<moved to 65.1>>
56          <<deleted 56-57>>
58          if (from-part((count))) < (to-part((count)))
59            then
60              do
61              move (temp-part) = (to-part((count)));
62              move (to-part((count))) = (from-part((count)));
63              move (from-part((count))) = (temp-part);
64              doend;
65            <<deleted>>
65.1        let (count) = (count) + 1;
66          doend;
67        let (parts-this-pass) = (parts-this-pass) - 1;
68        return;
```

This example was purposely left as close as possible to the bubble sort example. The differences between the two examples are highlighted.

Lines 2.1 through 4 define the part number array and the individual items within the array that we need to access. Line 2.1 defines the array to contain 20 occurrences of 8 bytes each. Lines 2.2 through 4 define individual items within the array that we will use for storing, retrieving, and comparing.

The verbs used for IMAGE access cannot refer to subscripted items. The items must be parent items. Thus, line 23 retrieves the next part-number from the database. If we have not reached the end of the dataset, line 25.1 moves the part-number retrieved into the next array occurrence of part-table. Notice the double set of parentheses around the item count. The innermost set denotes that we are subscripting a reference to each-part. The next set denotes that we are using an item named count to contain the array occurrence to be accessed.

Using a fixed data structure rather than a dynamic data structure allows us to delete several lines of code that were needed to manipulate the dynamic data structure. These lines are identified as <<deleted>> throughout the example.

The balance of the example is very similar to the dynamic data structure implementation of the bubble sort problem. However, there is one other area of the example worth mentioning. Notice that line 4 defines to-part to start in the ninth byte of the part-table array. This differs from the definition of from-part which starts in the first byte of the part-table array (line 3). This allows us to access two different occurrences of part-number in the array using the same subscript value, as is illustrated in lines 58-64. For example, if count has the value 1, then using from-part references the first part-number in the array and using to-part references the second part-number in the array.

Sometimes one-dimensional arrays contain multiple occurrences of more than one item. The following example illustrates this.

Figure 8-10. One-Dimensional Record Array (Multiple Items)

```
 1     system ex67,base=orders;
 2     define(item) order-table 10 x(10):
 3                  ot-line 9(2)=order-table:
 4                  ot-part x(8)=order-table(3):
 5                  index i(4):
 6                  end-of-table i(4);
 7     list order-no:
 8          line-no:
 9          part-number:
10          index:
11          end-of-table:
12          order-table;
13     data order-no;
14     set(key)list (order-no);
15     let (index) = 1;
16     find(chain) orderline,list=(line-no:part-number)
17                           ,perform=100-each-line;
18     let (end-of-table) = (index);
19     let (index) = 1;
20     while (index) < (end-of-table)
21       do
22       display(table) ot-line((index)):
23                      ot-part((index));
24       let (index) = (index) + 1;
25       doend;
26     exit;
27
28     100-each-line:
29
30       move (ot-part((index))) = (part-number);
31       move (ot-line((index))) = (line-no);
32       let (index) = (index) + 1;
33       return;
```

Line 2 defines an array consisting of 10 occurrences of 10 bytes each. Each occurrence is made up of a 2-byte line number and an 8-byte part number (lines 3 and 4).

Notice again that the IMAGE access verbs cannot reference array items. Thus lines 16-17 retrieve the next values of line-no and part-number, then lines 30-31 move these values into the next array occurrence, and line 32 increments the array occurrence subscript.

Lines 20-25 are a loop that subscripts through the array and prints out the contents of line-no (ot-line) and part-number (ot-part).

Transact allows subscripting of only one dimension of an array. Consequently, if an array has more than one dimension, other methods must be used to qualify the array access of all but the outermost dimension. This qualification is made possible by the LET OFFSET verb.

The next example is a two-dimensional array and illustrates using a subscript to qualify the outermost dimension and using LET OFFSET to qualify the inner dimension.

Figure 8-11. Two-Dimensional Array

```
 1 system ex68,base=orders;
 2     define(item) order-table 10 x(50):
 3                  ot-yr-indx x(50) = order-table:
 4                  ot-year 9(2)=ot-yr-indx:
 5                  ot-mo-indx 12 9(4)=ot-yr-indx(3):
 6                  ot-mo 9(4)=ot-mo-indx;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4),init=1;
12     define(item) dun i(4):
13                  no i(4),init=0:
14                  yes i(4),init=1;
15     list dun:
16          no:
17          yes;
18     list order-table,init:
19          end-of-table:
20          indx:
21          date:
22          order-no:
```

Figure 8-12. Two-Dimensional Array (Continued)

```
23          order-date:
24          quantity;
25     find(serial) orderhead,list=(order-no,order-date),|
26                           perform=100-each-order;
27     display order-table;
28     exit;
30     100-each-order:
31
32       move (date) = (order-date);
33       set(key) list (order-no);
34       find(chain) orderline,list=(quantity)
35                            ,perform=200-each-line;
36       return;
37
38     200-each-line:
39       let (indx) = 0;
40       let (dun) = (no);
41       while (dun) = (no)
42         do
43         let (indx) = (indx) + 1;
44         if (ot-year((indx))) = (date-yy)
45           then let (dun) = (yes)
46           else
47         if (indx) = (end-of-table)
48           then
49             do
50             let (end-of-table) = (end-of-table) + 1;
51             let (ot-year((indx))) = (date-yy);
52             let (dun) = (yes);
53             doend;
54         doend;
55       let offset(ot-mo) = [(date-mm) - 1] * 4;
56       let (ot-mo((indx))) = (ot-mo((indx))) + (quantity);
57       return;
```

Lines 2 through 6 define the array. The array is intended to hold up to 10 years of data by month. The data that is to be accumulated into the table is the order quantity for each order and part number.

Line 2 specifies that the first dimension of the array is 10 years of information. At this time, we must also specify the total byte length of all information for a year (which is 50). As discussed below, the 50 bytes for each year contain a 2-byte year number and 12 monthly quantities of 4 bytes each.

Line 3 defines a parent item for one year in the array.

Lines 4 and 5 define the child items that make up a year. A subscript will be used to access a year and a byte offset via the LET OFFSET verb will be used to access a month within a year.

Line 5 also defines the parent item for one year of information by month. Line 6 defines a month within the year.

Line 44 uses the subscript indx to access the year in the array. If the value of the year in the array is the same as the value of the year in the input record, we have resolved the year to accumulate to. If it does not, then the loop is iterated until the matching year is found or the end of the array is reached. If the end of the array is reached, a new entry is made into the array, storing the new year number (line 51).

Line 55 specifies the byte offset for a month relative to its parent item. Since the length of data for each month is 4 bytes, the offset for the n th month is (n - 1) * 4. Use of LET OFFSET specifies the starting byte position relative to the base. The base is zero. Thus the first month has a zero offset, the second month has an offset of 4, which is the length of the item ot-mo.

The above example illustrates using both subscripts and LET OFFSET to access a multi-dimensional array.

The following example illustrates using LET OFFSET exclusively to access the array.

Figure 8-13. Two-Dimensional Array with LET OFFSET

```
 1     system ex68a,base=orders;
 2     define(item) order-table 10 x(50):
 3                  ot-yr-indx x(50) = order-table:
 4                  ot-year 9(2)=ot-yr-indx:
 5                  ot-mo-indx 12 9(4)=ot-yr-indx(3):
 6                  ot-mo 9(4)=ot-mo-indx;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4),init=0;
12     define(item) dun i(4):
13                  no i(4),init=0:
14                  yes i(4),init=1;
15     list dun:
16          no:
17          yes;
18     list order-table,init:
19          end-of-table:
20          indx:
21          date:
22          order-no:
23          order-date:
24          quantity;
25     find(serial) orderhead,list=(order-no,order-date),
26                            perform=100-each-order;
27     display order-table;
28     exit;
30     100-each-order:
31
32       move (date) = (order-date);
33       set(key) list (order-no);
```

Figure 8-14. Two-Dimensional Array with LET OFFSET (Continued)

```
 34       find(chain) orderline,list=(quantity)
 35                            ,perform=200-each-line;
 36       return;
 37
 38     200-each-line:
 39       let (indx) = -1;
 40       let (dun) = (no);
 41       while (dun) = (no)
 42         do
 43         let (indx) = (indx) + 1;
43.1        let offset(ot-yr-indx) = (indx) * 50;
 44         if (ot-year) = (date-yy)
 45           then let (dun) = (yes)
 46           else
 47         if (indx) = (end-of-table)
 48           then
 49             do
 50             let (end-of-table) = (end-of-table) + 1;
 51             let (ot-year) = (date-yy);
 52             let (dun) = (yes);
 53             doend;
 54         doend;
 55       let offset(ot-mo) = [(date-mm) - 1] * 4;
 56       let (ot-mo) = (ot-mo) + (quantity);
 57       return;
```

The differences between this example and example EX68 are highlighted.

Line 43.1 specifies the byte offset for a year. At this point, we have resolved addresses for the child items of ot-yr-indx which are ot-year and ot-mo-indx. Since the length of the data for each year is 50 bytes (2 byte year and 12 months of 4 bytes each), the offset for the n th year is (n -1) * 50. However, in the example, indx is used to indicate a year and has been specified relative to zero. Thus the example does not need to convert n to be relative to zero.

Line 55 specifies the byte offset for a month relative to its parent item. Since the length of data for each month is 4 bytes, the offset for the n th month is (n - 1) * 4.

Notice that once the proper offsets for an item have been established, the item can be referenced directly without any further qualification (lines 44, 51, and 56).

The above example is not the only way a two-dimensional array can be implemented. It is perhaps as close as it is possible to get to the way COBOL or Pascal might define the same array, recognizing that Transact always requires you to specify byte offsets rather than an occurrence number.

The following example illustrates an alternative way to implement the same array. It is not any better than the first and in fact, the first is probably easier to follow. However, it is presented with the hope of improving your understanding of the ways you can manipulate data storage.

Figure 8-15. Two-Dimensional Array, Special Use of LET OFFSET

```
 1     system ex69,base=orders;
 2     define(item) order-table x(500):
 3                  <<deleted>>
 4                  ot-year 9(2)=order-table:
 5                  <<deleted>>
 6                  ot-mo 9(4)=order-table;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4),init=0;
12     define(item) dun i(4):
13                  no i(4),init=0:
14                  yes i(4),init=1;
15     list dun:
16          no:
17          yes;
18     list order-table,init:
19          end-of-table:
20          indx:
21          date:
22          order-no:
23          order-date:
24          quantity;
```

Figure 8-16. Two-Dimensional Array, Special Use of LET OFFSET (Continued)

```
25     find(serial) orderhead,list=(order-no,order-date),
26                             perform=100-each-order;
27     display order-table;
28     exit;
30     100-each-order:
31
32       move (date) = (order-date);
33       set(key) list (order-no);
34       find(chain) orderline,list=(quantity)
35                              ,perform=200-each-line;
36       return;
37
38     200-each-line:
39       let (indx) = -1;
40       let (dun) = (no);
41       while (dun) = (no)
42         do
43         let (indx) = (indx) + 1;
43.1       let offset(ot-year) = (indx) * 50;
44         if (ot-year) = (date-yy)
45           then let (dun) = (yes)
46           else
47         if (indx) = (end-of-table)
48           then
49             do
50             let (end-of-table) = (end-of-table) + 1;
51             let (ot-year) = (date-yy);
52             let (dun) = (yes);
53             doend;
54         doend;
55   let offset(ot-mo) = offset(ot-year) + 2 + [(date-mm) - 1] * 4;
56       let (ot-mo) = (ot-mo) + (quantity);
57       return;
```

In this example, the definition of the array makes no reference to the number of years or the number of months in a year. It is just 500 bytes long, which is the same as the previous example of 10 years by month where the data for each year is a 2-digit year and 12 months of 4 bytes each.

The only items that will ever be referred to in this array are ot-year and ot-mo. These are the only items defined and the definition of them does not indicate where they are within the array. The definition only establishes that they are child items of the array, and references to them will be made relative to the start of the array.

This means that before we address either of these items, we must fully establish the byte offset of each.

Line 43.1 establishes the byte offset for ot-year. There is no difference between this and the previous example. The reason is that both examples define ot-year relative to the start of the array.

Line 55 establishes the byte offset for ot-mo. Here there is a big difference between this and the previous example. The previous example defined the start of the months to be relative to the start of a year plus 2 bytes. The start of a year was defined relative to the start of the array. In the current example, ot-mo was defined relative to the start of the array. Therefore its byte offset must account for the year offset plus the month offset within year plus the fact that the start of the months for a year are offset from the start of a year by 2 bytes to allow for storage of a 2-byte year number.

Looking at line 55 and relating it to the above, we already know the year offset since we set it in line 43.1. Thus we can include it by specifying OFFSET(ot-year) which is equivalent to the calculation (indx) * 50. Each month is 4 bytes long, so the offset of the month we want to reference relative to zero is [(date-mm) - 1] * 4. Finally, the first month of a year is offset by 2 bytes from the beginning of each year's data.

Subprograms

Transact allows you to call other programs from within Transact. We have already seen an example of calling a Report/V program from Transact. You can also call another Transact Program, Inform report, or a program written in another language such as COBOL or Pascal.

The CALL verb is used to call another Transact, Report/V, or Inform program. The PROC verb is used to call a program written in another language.

Calling a Transact program is easily demonstrated by taking the example program listed in Figure 8-9 and dividing it into a main program which does all the database access and a subprogram that takes care of the array processing. The following two programs result.

Figure 8-17. Calling a Subprogram

```
 1     system ex70,base=orders;
 2     define(item) order-table 10 x(50):
 3                  ot-yr-indx x(50) = order-table:
 4                  ot-year 9(2)=ot-yr-indx:
 5                  ot-mo-indx 12 9(4)=ot-yr-indx(3):
 6                  ot-mo 9(4)=ot-mo-indx;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4),init=1;
12     define(item) dun i(4):
13                  no i(4),init=0:
14                  yes i(4),init=1;
15     list dun:
16          no:
17          yes;
18     list order-table,init:
19          end-of-table:
20          indx:
21          date:
22          order-no:
23          order-date:
24          quantity;
25     find(serial) orderhead,list=(order-no,order-date),
26                            perform=100-each-order;
27     display order-table;
28     exit;
29
30     100-each-order:
31
32       move (date) = (order-date);
33       set(key) list (order-no);
34       find(chain) orderline,list=(quantity)
35                             ,perform=200-each-line;
36       return;
37
38     200-each-line:
39       call ex70a;
40       return;
```

Figure 8-18. The Called Subprogram

```
 1     system ex70a,base=orders;
 2     define(item) order-table 10 x(50):
 3                  ot-yr-indx x(50) = order-table:
 4                  ot-year 9(2)=ot-yr-indx:
 5                  ot-mo-indx 12 9(4)=ot-yr-indx(3):
 6                  ot-mo 9(4)=ot-mo-indx;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4);
12     define(item) dun i(4):
13                  no i(4):
14                  yes i(4);
15     list dun:
16          no:
17          yes;
18     list order-table:
19          end-of-table:
20          indx:
21          date:
22          order-no:
23          order-date:
24          quantity;
25     let (indx) = 0;
26     let (dun) = (no);
27     while (dun) = (no)
28       do
29       let (indx) = (indx) + 1;
30       if (ot-year((indx))) = (date-yy)
31         then let (dun) = (yes)
32         else
33       if (indx) = (end-of-table)
34         then
35           do
36           let (end-of-table) = (end-of-table) + 1;
37           let (ot-year((indx))) = (date-yy);
38           let (dun) = (yes);
39           doend;
40       doend;
41     let offset(ot-mo) = [(date-mm) - 1] * 4;
42     let (ot-mo((indx))) = (ot-mo((indx))) + (quantity);
43     exit;
```

In this example, the main program and subprogram share the same data. The two programs do not have to define the data storage identically, but both must be aware that they are working with the same space. In our example, both programs do define the data storage the same way.

The example can be modified so that the two programs only share the data that both need.

Figure 8-19. Calling a Subprogram Using DATA=

```
 1     system ex71,base=orders;
 2     define(item) order-table 10 x(50);
 3     define(item) end-of-table i(4),init=1;
 4     list order-no:
 5          order-date:
 6          quantity:
 7          order-table,init:
 8          end-of-table;
 9     find(serial) orderhead,list=(order-no,order-date),
10                           perform=100-each-order;
11     display order-table;
12     exit;
13
14     100-each-order:
15
16       set(key) list (order-no);
17       find(chain) orderline,list=(quantity)
18                            ,perform=200-each-line;
19       return;
20
21     200-each-line:
22       call ex71a,data=order-date;
23       return;
```

Figure 8-20. The Called Subprogram with DATA=

```
 1     system ex71a,base=orders;
 2     define(item) order-table 10 x(50):
 3                  ot-yr-indx x(50) = order-table:
 4                  ot-year 9(2)=ot-yr-indx:
 5                  ot-mo-indx 12 9(4)=ot-yr-indx(3):
 6                  ot-mo 9(4)=ot-mo-indx;
 7     define(item) date x(6):
 8                  date-yy 9(2)=date:
 9                  date-mm 9(2)=date(3):
10                  indx i(4):
11                  end-of-table i(4);
12     define(item) dun i(4):
13                  no i(4),init=0:
14                  yes i(4),init=1;
15     list date:
16          quantity:
17          order-table:
18          end-of-table:
19          indx:
20          dun:
21          no:
22          yes;
23     let (indx) = 0;
24     let (dun) = (no);
25     while (dun) = (no)
26       do
27       let (indx) = (indx) + 1;
28       if (ot-year((indx))) = (date-yy)
29         then let (dun) = (yes)
30         else
31       if (indx) = (end-of-table)
32         then
33           do
34           let (end-of-table) = (end-of-table) + 1;
35           let (ot-year((indx))) = (date-yy);
36           let (dun) = (yes);
37           doend;
38       doend;
39     let offset(ot-mo) = [(date-mm) - 1] * 4;
40     let (ot-mo((indx))) = (ot-mo((indx))) + (quantity);
41     exit;
```

In this example, the main program only contains definitions for the data that it needs and at the level that it needs access. For example, it does not contain a detail definition of the array. The subprogram does contain a detail definition of the array.

The main program also specifies how much data the subprogram has access to by way of the DATA=ORDER-DATE option on the CALL. This protects order-no from being accessed by the subprogram. The subprogram must be aware of all data being passed to it and define its storage first. Then it can define its own local storage following this.

We can also use this same array example to illustrate calling a program written in another language. In this example, the main program does the database access and the subprogram written in COBOL performs the array handling for us.

Figure 8-21. Calling a COBOL Procedure

```
 1     system ex72,base=orders;
 2     define(item) order-table 10 x(50);
 3     define(item) end-of-table i(4),init=1;
 4     define(item) year-subx 9(2)=order-date:
 5                  month-subx 9(2)=order-date(3);
 6     list order-no:
 7          order-date:
 8          quantity:
 9          order-table,init:
10          end-of-table;
11     find(serial) orderhead,list=(order-no,order-date),
12                           perform=100-each-order;
13     display order-table;
14     exit;
15
16     100-each-order:
17
18       set(key) list (order-no);
19       find(chain) orderline,list=(quantity)
20                            ,perform=200-each-line;
21       return;
22
23     200-each-line:
24       proc ex72a((order-table),(end-of-table),(year-subx),(month-subx),
25                  (quantity));
26       return;
```

The COBOL subprogram looks like this.

Figure 8-22. The Called COBOL Procedure

```
1     $control dynamic
1.1    identification division.
1.2    program-id. ex72a.
1.3    environment division.
1.4    data division.
1.5    working-storage section.
1.6    01 i pic 9(4) comp.
1.7    linkage section.
1.8    01 yr pic 99.
1.9    01 mo pic 99.
2      01 data-table.
2.1      02 yrs occurs 10.
2.2        04 tab-yr pic 99.
2.3        04 tab-qty pic 9(4) occurs 12.
2.4    01 end-tab pic 9(4) comp.
2.5    01 qty pic 9(6) comp.
2.6    procedure division using data-table end-tab yr mo qty.
2.7    100-start.
2.8    perform 200-find varying i from 1 by 1 until yr = tab-yr (i)
2.9                                              or i = end-tab.
3          if i = end-tab
3.1            move yr to tab-yr (i)
3.2            compute end-tab = end-tab + 1.
3.3        compute tab-qty (i, mo) = tab-qty (i, mo) + qty.
3.4        goback.
3.5
3.6    200-find.
```

The subprogram can only be accessed from an SL. A way to load the COBOL subprogram into an SL is:

Figure 8-23. Adding a COBOL Procedure to an SL

```
:cobol ex72a
PAGE 0001   HP32213C.02.12   (C) HEWLETT-PACKARD CO. 1983

      DATA AREA IS %000172 WORDS.
      CPU TIME = 0:00:01.  WALL TIME = 0:00:03.
END COBOL/3000 COMPILATION.  NO ERRORS.  NO WARNINGS.
 END OF COMPILE

:segmenter
HP32050A.01.09  SEGMENTER/3000 (C) HEWLETT-PACKARD CO 1983
-buildsl sl,1000,100
-usl $oldpass
-listusl

USL FILE $OLDPASS.HOWTO.MILLER

EX88A'
   EX88A'            230  P  A C N R
EX88A
   EX88A             525  P  A C N R
   EX88A'S                CP A C   R

FILE SIZE     377600( 1777.  0)
DIR. USED        311(    1.111)    INFO USED       1027(    4. 27)
DIR. GARB.         0(    0.  0)    INFO GARB.         0(    0.  0)
DIR. AVAIL.    37267(  175. 67)    INFO AVAIL.   336751( 1573.151)
-addsl ex72a'
-addsl ex72a
-exit
END OF PROGRAM

:run transact.pub.sys
TRANSACT/3000    HP32247A.01.09 - (C) Hewlett-Packard Co. 1983

SYSTEM NAME> ex72

PASSWORD FOR ORDERS>

ORDER-TABLE:
  860600
  850900    1200

END OF PROGRAM
```

Intrinsics

The PROC verb can also be used to call intrinsics. These intrinsics can be the MPE intrinsics, VPLUS intrinsics, IMAGE, KSAM, etc., intrinsics.

The example below illustrates the use of several of the MPE intrinsics.

Figure 8-24. Accessing Intrinsics with PROC

```
 1     system ex73;
 2     define(item) filerec x(40):
 3                  fileref x(8):
 4                  fopt i(4):
 5                  aopt i(4):
 6                  bitmap i(4):
 7                  fnum i(4):
 8                  icount i(4):
 9                  icontrol i(4):
10                  filesize i(9),init=1023;
11     define(item) comimage x(40):
12                  comcr x(1)=comimage(40):
13                  ncr i(4):
14                  xncr2 x(1)=ncr(2):
15                  error i(4):
16                  parm i(4);
17     list comimage:
18          ncr:
19          error:
20          parm;
21     move (comimage) = "build ex73f;rec=-40,,,ascii;disc=10";
22     let (ncr) = 13;  <<decimal equivalent of a carriage return>>|
23     move (comcr) = (xncr2);
24     proc command(%(comimage),(error),(parm));
25     list filerec:
26          fileref:
27          fopt:
28          aopt:
29          bitmap:
30          fnum:
31          icount:
32          icontrol:
33          filesize;
34     move (filerec) = "this is test data";
35     move (fileref) = "ex73f";
36     let (fopt) = 1;     <<bits 14-15 = 01  = old file>>
37     let (aopt) = 1;     <<bite 12-15 = 0001 = write only>>
38     let (bitmap) = 7176; << = 1110000001000 = parms 1,2,3,10>>
39     let (icount) = -40;  << write 40 bytes>>
40     let (icontrol) =0;     <<single space>>
41     proc fopen(%(fileref),#(fopt),#(aopt),,,,,,,#(filesize),,,,
42                 #(bitmap),&(fnum));
43     proc fwrite(#(fnum),(filerec),#(icount),#(icontrol));
```

The example is a program that builds a file using the COMMAND intrinsic and then writes a record to the file using the FOPEN and FWRITE intrinsics.

Line 21 sets up the buffer for the COMMAND intrinsic to be the BUILD command. The buffer for a command has to end with a [[RETURN]] which is what lines 22 and 23 accomplish.

Line 41 executes the FOPEN intrinsic. This line illustrates several important points to remember when calling intrinsics. The intrinsics manual describes the information we need to know about both the intrinsic and the data being passed to it.

The first parameter passed to the FOPEN intrinsic is the formal designator or filename of the file to be opened. It is a byte array. The % in front of the first parameter tells Transact that this parameter is to be passed as a byte array.

The next parameter is the foptions. This is a word that FOPEN interprets groups of bits as a description of the attributes of the file. The bit numbering convention is from left to right starting with bit 0 and ending with bit 15. In line 36, we established a value for fopt with only bit 15 on. This specifies the file domain to be an old permanent file. The # tells Transact to pass this parameter by value to the intrinsic.

The parameter to satisfy the aoptions is also passed by value.

The remainder of the FOPEN parameters do not need to be specified for our use of the intrinsic. However, there is one special case parameter which we need to consider, which is the filesize parameter. This parameter is a double word parameter that is passed by value. Transact does not know anything about the data types that intrinsics expect. It expects you to provide that information. This works great and is easy to do for all data types except optional double by value. When an option variable intrinsic contains this kind of parameter, pass the parameter with either the default value in those cases where you don't care what the parameter value is or with the needed value in those cases where you do care. This assures that the correct number of words are passed to the intrinsic when called.

Another special thing about this intrinsic is that it is an option variable intrinsic. This means a bit map must be passed to the intrinsic telling it which parameters are being passed to it. The intrinsic expects 13 parameters. Thus one word is passed to the intrinsic with the first 13 bits of the word designating those parameters passed. In our example, parameters 1, 2, 3, and 10 were passed, so the first three and the 10th bits of the bit map word are set on in line 38 (1110000001000 to the base 2 is equal to 7176 to the base 10).

The last special thing about this intrinsic is that it is a function intrinsic which means it returns a value to you. The very last parameter we pass it is the parameter to hold this data. The & in front of the parameter tells Transact that this is where the value returned is to be placed.

Line 43 executes the FWRITE intrinsic to write the data we specified to the file opened in line 41.

Test Facility

Transact provides an extensive test facility to aid in the debugging of a program. The *Transact Reference Manual* provides a detailed description of this facility.

Three of the options that provide the answers to most debugging problems are options 25, 34, and 42. Option 25 provides information whenever a database or file access occurs. Option 34 provides information whenever a VPLUS action occurs and option 42 provides information whenever the data storage description changes.

Depending on the program being debugged and the test option chosen, Transact may display a considerable amount of data for you. Another useful option which helps reduce the amount of data displayed is to indicate a range of internal location reference numbers where the display is to occur. Data is then displayed only when the program is executing within the range specified. Before using this option, you must first compile your program with the LIST option in order to get a listing of internal location reference numbers.

The test facility by default displays the data on your terminal. Quite often, it is desirable to direct the data to a file so as not to interfere with terminal activity. This is especially true when debugging a VPLUS program. The file designator used for displaying test facility results is TRANDUMP. The default of the terminal can be altered to some other device, typically a spooler file by doing two things. First set up a file equate such as:

```
FILE TRANDUMP;DEV=LP,1
```

Second, when specifying the option desired to the test facility, precede the option with a minus sign.

Below is an example of debugging a VPLUS program with test option 34 and directing the output to a spooler file. First, the program is compiled to get the internal location reference numbers, then the file equation is entered to direct the test results to the TRANDUMP file. When the program is run, test mode 34 is specified, and the debugging range is limited to the update-customer paragraph (internal location reference numbers 21 to 28). The last step is to look at the SPOOK output.

```
:run trancomp.pub.sys

TRANSACT/3000 COMPILER HP32247A.02.02 - (C) Hewlett-Packard Co. 1984

SOURCE FILE> ex74

LIST FILE>

CONTROL>
TRANSACT/3000 COMPILER A.02.02 : MON, MAR 18, 1985, 8:11 AM COMPILED
LISTING OF FILE EX13.HOWTO.MILLER PAGE 1

COMPILING WITH OPTIONS: LIST,CODE,DICT,ERRS
   1.000 system ex74,base=orders,vpls=formfile;
   2.000 0000 list(auto) customer;
   3.000 0005 level;
   4.000 0006   get(form) mainmenu,f1=add-customer
   5.000 0006                     ,f2=update-customer
   6.000 0006                     ,f3=report-customer;
```

```
   6.100 0017   end;
   7.000 0018
   8.000 0018 add-customer:
   9.000 0018
  10.000 0018   get(form) vcustomer;
  11.000 0019   put customer;
  12.000 0020   end;
  13.000 0021
  14.000 0021 update-customer:
  15.000 0021
  16.000 0021   get(form) vcustno;
  16.100 0022   set(key) list (cust-no);
  17.000 0023   get customer;
  18.000 0025   put(form) vcustomer;
  19.000 0026   get(form) vcustomer;
  20.000 0027   update customer;
  21.000 0028   end;
  22.000 0029
  23.000 0029 report-customer:
  24.000 0029
  25.000 0029   get(form) vcustno;
  25.100 0030   set(key)list (cust-no);
  26.000 0031   get customer;
  27.000 0033   put(form) vcustomer,wait=
  28.000 0033                       ,window=("press f1-f8 to continue");
  29.000 0037   set(form) vcustomer,window=(" ");
  30.000 0041   end;
CODE FILE STATUS: REPLACED

0 COMPILATION ERRORS
PROCESSOR TIME=00:00:05
 ELAPSED TIME=00:00:09

END OF PROGRAM
:file trandump;dev=lp,1
:run transact.pub.sys

TRANSACT/3000 HP32247A.02.02 - (C) Hewlett-Packard Co. 1984

SYSTEM NAME> ex74,,-34,21,28

 END OF PROGRAM

:run spook5.pub.sys

SPOOK5 G.01.00 (C) HEWLETT-PACKARD CO., 1983
> s
#FILE    #JOB    FNAME     STATE    OWNER
#O3449   #S344   TRANDUMP  READY    MARV.MILLER
> t3449
> l all 0 1 +-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+
  2 |
  3 | GET(FORM) CODE: 0        FKEY: 2
  4 | FORM: ..MAINMENU FILE: ..FORMFILE
  5 | 6 +-----------------------------------+
  7 1         00021 160:009
  8 +-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+
  9 |
 10 | GET(FORM) CODE: 0 FKEY: 0 11 | FORM: ..VCUSTNO . FILE: ..FORMFILE
 12 |
```

```
 13 | OFFSET: LIST:          DATA:
 14 | 0       CUST-NO       1
 15 |
 16 +--------------------------------------+
 17 1         00022 198:000
 18 1         00023 064:139
 19 1         00025 161:002
 20 +-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+
 21 |
 22 | PUT(FORM) CODE: 0     FKEY: 0
 23 | FORM: ..VCUSTOMER FILE: ..FORMFILE
 24 |
 25 | OFFSET: LIST:          DATA:
 26 | 0       CUST-NO       1
 27 | 4       NAME          updated name
 28 | 24      STREET-ADDR   street
 29 | 44      CITY-STATE    city
 30 | 64      ZIPCODE       zip
 31 |
 32 +--------------------------------------+
 33 1         00026 160:002
 34 +-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+
 35 |
 36 | GET(FORM) CODE: 0      FKEY: 0
 37 | FORM: ..VCUSTOMER FILE: ..FORMFILE
 38 |
 39 | OFFSET: LIST:          DATA:
 40 | 0       CUST-NO       1
 41 | 4       NAME          new name
 42 | 24      STREET-ADDR   street
 43 | 44      CITY-STATE    city
 44 | 64      ZIPCODE       zip
 45 |
 46 +--------------------------------------+
 47 1         00027 049:011
 48 1         00028 000:000
 49 +-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+
 50 |
 51 | GET(FORM) CODE: 0      FKEY: 8
 52 | FORM: ..MAINMENU FILE: ..FORMFILE
 53 |
 54 +--------------------------------------+
```

9 Creating Custom Applications

In this section, we look at coding techniques that use the power of Transact and Dictionary/V. One of the most time-consuming functions of an application system's programming team is program maintenance. Quite often this time is related to the kinds of activity that can be greatly reduced or even eliminated by taking full advantage of the integration of Transact and the dictionary.

Some of these time-consuming activities are: adding new input forms, changing or deleting existing input forms, adding new data elements, deleting existing elements, or changing existing elements in size.

We will follow through examples of localizing a program, that is, making it independent of the form name and of the number of forms that execute the same code, independent of the form contents, and provide user exits for additional processing. The changes we make always apply to the dictionary. The changes become effective in the program when the program is recompiled in order to pull in the new dictionary definitions.

An application system can be divided into at least two parts. The first part is made up of the data that is needed by the system processing logic. This data or these data elements are critical to the proper functioning of the system. For example, a manufacturing system no doubt has an element called part-number which is a critical part of practically all system transactions.

A typical application may have several critical data elements. It is fair to say that a localizable application cannot allow critical fields to be deleted from the application. Application programs rely upon these fields to be present in transactions.

However, a localizable application should allow these elements to be changed in size. It should also allow the physical placement of these elements within an input form to be changed.

The other part of an application is made up of noncritical elements. Many of these elements may be supplied as a part of the original application, if for no other reason than the typical generic application has these data elements. Other noncritical elements may be added by the particular user of the application.

A localizable application should allow noncritical elements to be added, deleted, changed in size, and changed in physical placement within an input form.

Also, generic transactions should be localizable. A generic transaction is defined here to be a transaction that provides a basic function such as adding a customer or updating a customer. For example, one organization may be responsible for adding new customers to the database, but several organizations need to be able to update portions of the customer data. Each organization should be provided with a form which only accesses the data they need. The same program logic that provides customer update capability should be able to handle any number of these variations.

Finally, a localizable application should allow logic to be added to handle such things as: special field edits for any of the transaction's data elements, data calculations, etc. With this brief background, let's look at how Transact can achieve this level of localization. Our objective is to write an application program such that if we choose to change the application as described above, we need not modify the program. We only need to record the changes in the dictionary, recompile the program to make the changes known to it, modify our VPLUS forms file to reflect the changes, and possibly unload and reload the database, if its structure has been modified.

We will start with a simple transaction that only applies to one dataset. This will demonstrate all of the concepts we want to achieve through localization. Later we will extend this example to include a transaction that applies to several datasets, in order to demonstrate the general case of how to write generic code.

Let's use the customer dataset of our example database for illustration. Our generic transaction provides the capability to update information for a customer. Breaking this application into the two parts discussed above, the critical element in this transaction is cust-no. The noncritical elements are: name, street-addr, city-state, and zip-code.

The VPLUS form used for each of these functions is vcustomer.

Figure 9-1. VPLUS Form to Maintain a Dataset

```
vcustomer                  customer data

                              number  [    ]

                              name    [                    ]

                              address [                    ]

                           city,state [                    ]

                              zipcode [      ]
```

The dataset definition looks like this:

Figure 9-2. Dictionary Definitions for Customer VPLUS Form

```
                          FILE customer

FILE                TYPE: RESPONSIBILITY:
  CUSTOMER            MAST

ELEMENT(ALIAS):             PROPERTIES:       ELEMENT(PRIMARY):
CUST-NO             *       I+(4,0,2)           CUST-NO
NAME                        X (20,0,20)         NAME
STREET-ADDR                 X (20,0,20)         STREET-ADDR
CITY-STATE                  X (20,0,20)         CITY-STATE
ZIPCODE                     X (6,0,6)           ZIPCODE
```

The following program illustrates how a transaction to update a customer might be written without allowing for any localization. We will expand this program to illustrate most of the localization concepts.

Figure 9-3. Basic Program to Maintain Customers (VPLUS)

```
 1     system custfm,base=orders,vpls=formfile;
 2     list(auto) customer;
 3
 4     level;
 5     get(form) vcustomer,init;
 6
 7     set(key) list (cust-no);
 8     get customer,list=(@);
 9     put(form) vcustomer,window=("update? - f1=yes, f2=no");
10     get(form) vcustomer,f1(autoread)=modify-f1
11                        ,f2=modify-f2;
12
13     modify-f1:
14
15       update customer,list=(@);
16
17     modify-f2:
18
19       end;
20
21
22     exit:
23
24       exit;
```

The program uses the same form to initially input the customer to be updated (line 5), display the current data for the customer (line 9), and input the new data for the customer (line 10).

Rearranging the Form

Perhaps the easiest form of localization is to rearrange the order of elements on the form. Our program form specification does not include any element ordering information. This is controlled through the dictionary. Thus, this localization can be accomplished by modifying the form using FORMSPEC, changing the element sequence on the form definition in the dictionary, and recompiling our program using TRANCOMP.

The same program could then handle input from a form such as this:

Figure 9-4. Rearranged Customer VPLUS Form

```
            vcustomer                  customer data

                    name [                    ]       number [   ]

                 address [                    ]

              city,state [                    ]

                 zipcode [      ]
```

Changing the form definition in the dictionary might go something like this:

Figure 9-5. Dictionary Changes to Specify a Rearranged Screen

```
 :run dictdbm.pub.sys

DICTIONARY/3000  HP32244A.02.01  - (C) Hewlett-Packard Co. 1984


  PASSWORD FOR DICT.PUB>

FORMS ENTRY(Y/N)?>

> show file
                    FILE vcustomer


FILE                  TYPE: RESPONSIBILITY:
VCUSTOMER             FORM

      ELEMENT(ALIAS):           PROPERTIES:        ELEMENT(PRIMARY):
      CUST-NO                 I+(4,0,2)          CUST-NO
      NAME                    X (20,0,20)        NAME
      STREET-ADDR             X (20,0,20)        STREET-ADDR
      CITY-STATE              X (20,0,20)        CITY-STATE
      ZIPCODE                 X (6,0,6)          ZIPCODE

> resequence file
                   FILE vcustomer


               ELEMENT name
          NEW POSITION cust-no


             ELEMENT
> show file
                   FILE vcustomer


FILE                  TYPE: RESPONSIBILITY:
VCUSTOMER               FORM


      ELEMENT(ALIAS):         PROPERTIES:         ELEMENT(PRIMARY):
      NAME                     X (20,0,20)         NAME
      CUST-NO                  I+(4,0,2)           CUST-NO
STREET-ADDR                    X (20,0,20)         STREET-ADDR
CITY-STATE                     X (20,0,20)         CITY-STATE
ZIPCODE                           X (6,0,6)             ZIPCODE
```

Form Independence

Now let's see how to implement generic transaction code that can handle multiple form formats. This will be illustrated by modifying the program above. Keep in mind that we could have written the program this way to start with. It is not a program modification that we must make every time we want to add another form.

The following program provides generic update customer capability and is form independent. That is, the program has no idea of which data elements exist on a form, nor does it know how many possible different forms may be used to update a customer.

Figure 9-6. Screen Independence Via Indirect Referencing

```
 1     system custup,base=orders
 2                  ,vpls=formfile
 3                  ,file=formxref;
 4     define(item) menuname x(16):
 5                  fkey     9(2):
 6                  screen   x(16):
 7                  lastkey i(4);
 8     list menuname:
 9          lastkey;
10
11     data menuname; <<to simulate transfer of control to this subroutine>>
12
13     <<
14     ************************************************************
15     Subroutine: to update customer information
16
17     input: menuname - contains the name of the screen to be displayed
18
19     output: none
20     ************************************************************
21     >>
22     level;
23     list fkey:
24          screen;
25     list(auto) customer;
26     get(form) (menuname),init
27                         ,window=(" ")
28                         ,fkey=lastkey
29                         ,autoread;
30     if (lastkey) = 0
31       then perform modify
32     else
33       do
34       set(match) list (menuname);
35       let (fkey) = (lastkey);
36       set(match) list (fkey);
37       get(serial) formxref,list=(menuname,fkey,screen);
38       reset(option) match;
39       perform modify;
40       doend;
41     end;
43     modify:
44
45       set(key) list (cust-no);
46       get customer,list=(@);
47       put(form) (screen),window=("update? - f1=yes, f2=no");
48       get(form) (screen),f1(autoread)=modify-f1
49                         ,f2=modify-f2;
50
51     modify-f1:
52
53       update customer,list=(@);
54
55     modify-f2:
56
57       end;
```

This program uses Transact's indirect referencing capability for forms. Notice that all verbs which reference a form name do not actually specify the form name. Each verb specifies the name of an element which contains the name of the form to be referenced.

The program sets up a menu-driven customer update capability such as the following series of forms depict.

Figure 9-7. Screen Independence, Customer Main Menu

```
custupdatemm            customer update main menu


            enter customer number      [1   ]
               f1 -  marketing         (custupdate1)
               f2 -  finance           (custupdate2)
               f3 -  accounts payable  (custupdate3)
          ********************************************
                           or
                enter screen name [               ]

market- finance  accounts                                    exit
ing               payable
```

Figure 9-8. Screen Independence, Marketing Customer Update

```
custupdate1                marketing customer update

                            customer number  [1   ]
                            name             [name of customer 1  ]


update? - f1=yes, f2=no
```

Figure 9-9. Screen Independence, Finance Customer Update

```
custupdate2                finance customer update

                           customer number [1   ]
                           zip code        [12345 ]


update? - f1=yes, f2=no
```

Figure 9-10. Screen Independence, Accounts Payable Customer Update

```
custupdate3                accounts payable customer update

                               customer number [1   ]
                                       name [name of customer 1  ]
                                    address [108 Lincoln Ave.    ]
                                  city,state [So. Bend, Ind.     ]
                                    zipcode [12345 ]

update? - f1=yes, f2=no
```

There are many ways to implement a form-independent program. The above is just one illustration. The key to this implementation is the MPE file called FORMXREF which provides the indirection we need to establish form independence.

The content of FORMXREF is as follows:

Figure 9-11. Screen Independence, Form Cross Reference File

```
MENUNAME:        FKEY: SCREEN:
-----------------------------------
CUSTUPDATEMM     1     CUSTUPDATE1
CUSTUPDATEMM     2     CUSTUPDATE2
CUSTUPDATEMM     3     CUSTUPDATE3
```

MENUNAME and FKEY are the index into the file specifying the menu that the user is currently working with and the function key just pressed by the user to indicate the next form to go to. SCREEN contains the name of the next data entry form to use.

When this program begins, the element menuname contains the name of the menu that controls its functionality. Line 11 simulates this by prompting for the menu name. When prompted for the menu name, we typed in CUSTUPDATEMM.

The menu we have set up allows the user to specify the next form in either of two ways. The name of the form can be entered in the box titled enter screen name. The [[ENTER]] enters this data and lines 30 and 31 detect this and perform the update routine. Or, the form can be indicated via a function key. If this way is chosen, the file formxref is accessed to determine the form name to be used by the modify routine. Lines 34 through 39 accomplish this.

The cross reference file has a record for each function key of each form that defines the name of the form to use when that function key is pressed. In our example, if the user presses [[F1]], then form custupdate1 is used.

Another form for updating a customer could now be designed and used by this program merely by recompiling the program. Of course, the form would have to be designed in FORMSPEC and defined in the data dictionary first.

Adding, Deleting, and Changing Elements

Now let's take on a major localization step. Let's redefine the customer dataset, expanding the size of cust-no from 4 to 6 digits long, deleting the zipcode field and adding a new field called area. The following Figures illustrate what must be done to accomplish all of this. The important point for Transact is that we only need to recompile the program to incorporate the new structure.

First we modify the form file, changing all the forms that reference the customer data.

custupdatemm is changed to reflect the 6-digit customer number.

Figure 9-12. Customer Main Menu, Change Size of cust-no

```
custupdatemm            customer update main menu

                   enter customer number      [      ]

                       f1 -  marketing         (custupdate1)
                       f2 -  finance           (custupdate2)
                       f3 -  accounts payable  (custupdate3)
               ********************************************
                                  or
                       enter screen name [               ]

market-  finance  accounts                                  exit
ing               payable
```

custupdate1 is changed to reflect the 6-digit customer number.

Figure 9-13. Marketing Customer Update, Change Size of cust-no

```
custupdate1                marketing customer update

                       customer number  [      ]
                       name             [                   ]
```

custupdate2 is changed to reflect the 6-digit customer number, delete of zipcode, and addition of area, because finance needs to be able to update this new code.

Figure 9-14. Finance Customer Update, Add, Change, Delete Elements

```
custupdate2                  finance customer update


                          customer number [      ]
                                    area [      ]
```

custupdate3 is changed to reflect the 6-digit customer number, delete of zipcode, and addition or area, because accounts payable needs to be able to update all fields.

Figure 9-15. AP Customer Update, Add, Change, Delete Elements

```
custupdate3                  accounts payable customer update


                          customer number [      ]
                                    name [                    ]
                                 address [                    ]
                              city,state [                    ]
                                    area [      ]
```

These changes, as well as the database changes, are recorded in the dictionary using DICTDBM.

Figure 9-16. Changing dictionary definitions, add, change, Delete Element

```
:run dictdbm.pub.sys


DICTIONARY/3000  HP32244A.02.01  - (C) Hewlett-Packard Co. 1984
PASSWORD FOR DICT.PUB>
FORMS ENTRY(Y/N)?>


> modify element
                     ELEMENT cust-no


EDIT DESCRIPTION(Y/N)?> n


ELEMENT              TYPE: SIZE: DEC: LENGTH: COUNT: RESPONSIBILITY:
CUST-NO               i+    4     0    4       1
LONG NAME:
HEADING TEXT:
ENTRY TEXT:
EDIT MASK:
MEASUREMENT UNITS:
BLANK WHEN ZERO: NO


                TYPE i+
                 SIZE 6
               DECIMAL !
```

Figure 9-17. Changing Dictionary Definitions

```
> create element
                ELEMENT area
             LONG NAME
                 TYPE x
                 SIZE 6
STORAGE LENGTH(6)    !

> delete file
                FILE custupdate3

                ELEMENT zipcode
ENTRY DELETED
                ELEMENT

> add file
                FILE custupdate3

              ELEMENT area
        ELEMENT ALIAS
         FIELD NUMBER
         DESCRIPTION
              ELEMENT

> delete file
                 FILE custupdate2

             ELEMENT zipcode
ENTRY DELETED
             ELEMENT
> add file
                FILE custupdate2

            ELEMENT area
      ELEMENT ALIAS
       FIELD NUMBER
        DESCRIPTION

           ELEMENT
> delete file
               FILE customer
            ELEMENT zipcode
ENTRY DELETED
            ELEMENT
> add file
               FILE customer
            ELEMENT area
      ELEMENT ALIAS
        DESCRIPTION
            ELEMENT
> exit

END OF PROGRAM
:
```

Next the database is unloaded using DICTDBU.

Figure 9-18. Unloading the Database with DICTDBU

```
:run dictdbu.pub.sys
DICTIONARY/3000 DB UNLOADER    P32244A.02.01-(C)Hewlett-Packard Co. 1984
STORE FILE> mpestore
LIST FILE>
BASE> orders
BASE PASSWORD>
MODE> 1
UNLOAD AUTOMATIC MASTER SETS(N/Y)?>
UNLOAD DETAIL SETS BY CHAIN(Y/N)?>
UNLOAD EDIT(N/Y)?>
PROCESSING SETS
CUSTOMER          M:2/100
2 ENTRIES UNLOADED IN <1 CPU-SEC
PARTS             M:2/100
2 ENTRIES UNLOADED IN <1 CPU-SEC
ORDER             A:2/100
AUTO NOT UNLOADED
INVENTORY         D:3/108
3 ENTRIES UNLOADED IN <1 CPU-SEC
ORDERHEAD         D:2/112
2 ENTRIES UNLOADED IN <1 CPU-SEC
ORDERLINE         D:3/100
3 ENTRIES UNLOADED IN <1 CPU-SEC
UNLOAD COMPLETED
END OF PROGRAM
:
```

Then the current database is purged using DBUTIL.

Figure 9-19. Purging the Database with DBUTIL

```
:run dbutil.pub.sys


HP32215B.04.61 IMAGE/3000: DBUTIL C)COPYRIGHT HEWLETT-PACKARD COMPANY 1978

>>pur orders
Data base has been PURGED.
>>exit


END OF PROGRAM
:
```

Then the new database root file is created using DICTDBC.

Figure 9-20. Creating the Database with DICTDBC

```
:run dictdbc.pub.sys
DICTIONARY/3000 DB CREATOR   HP32244A.02.01 - (C) Hewlett-Packard Co. 1984
DICTIONARY PASSWORD>
BASE> orders
CONTROL LINE>
SCHEMA FILE>
LISTING FILE>
APPLY SECURITY JUST TO SET LEVEL(N/Y)?>
SCHEMA GENERATION
DBSCHEMA PROCESSOR
PAGE 1      HEWLETT-PACKARD 32215B.04.50 IMAGE/3000: DBSCHEMA
         TUE, MAY 14, 1985,  9:21 AM  (C) HEWLETT-PACKARD CO. 1978

BEGIN DATA BASE ORDERS;
PASSWORDS:
ITEMS:
     AREA,               X6        ;
     CITY-STATE,         X20       ;
     CUST-NO,            I2        ;
     DESCRIPTION,        X20       ;
     LINE-NO,            X2        ;
     LOCATION,           X4        ;
     NAME,               X20       ;
     ORDER-DATE,         X6        ;
     ORDER-NO,           X8        ;
     ORDER-STATUS,       X2        ;
     PART-NUMBER,        X8        ;
     QUANTITY,           I2        ;
     STREET-ADDR,        X20       ;
```

Figure 9-21. Creating the Database with DICTDBC (Continued)

```
SETS:
NAME:   CUSTOMER,         MANUAL    ;
ENTRY:  CUST-NO          (1),
NAME,
STREET-ADDR,
CITY-STATE,
AREA;
CAPACITY: 100;
NAME:   PARTS,            MANUAL    ;
ENTRY:  PART-NUMBER      (2),
DESCRIPTION;
CAPACITY: 100;
NAME:   ORDER,            AUTOMATIC ;
ENTRY:  ORDER-NO         (2);
CAPACITY: 100;
NAME:   INVENTORY,        DETAIL    ;
ENTRY:  PART-NUMBER      ( PARTS           ),
LOCATION,
QUANTITY;
CAPACITY: 100;
NAME:   ORDERHEAD,        DETAIL    ;
ENTRY:  ORDER-NO         ( ORDER           ),
CUST-NO          ( CUSTOMER        ),
ORDER-STATUS,
ORDER-DATE;
CAPACITY: 100;
NAME:   ORDERLINE,        DETAIL    ;
ENTRY:  ORDER-NO         ( ORDER           )
LINE-NO,
PART-NUMBER      ( PARTS           ),
QUANTITY;
CAPACITY: 100;
END.
DATA SET      TYPE  FLD  PT  ENTR  MED  CAPACITY  BLK  BLK   DISC
  NAME              CNT  CT  LGTH  REC            FAC  LGTH  SPACE
CUSTOMER          M  5    1   36    46    100      11   507   44
PARTS             M  2    2   14    29    100      13   378   27
ORDER             A  1    2   4     19    100      20   382   18
INVENTORY         D  3    1   8     12    120      40   483   16
ORDERHEAD         D  4    2   11    19    100      20   382   18
ORDERLINE         D  4    2   11    19    100      20   382   18
                                TOTAL DISC SECTORS INCLUDING ROOT: 152
NUMBER OF ERROR MESSAGES: 0
ITEM NAME COUNT: 13      DATA SET COUNT: 6
ROOT LENGTH: 587     BUFFER LENGTH: 507     TRAILER LENGTH: 256
ROOT FILE ORDERS CREATED.
END OF PROGRAM
:
```

The new database is created using DBUTIL.

Figure 9-22. Creating the New Database with DBUTIL

```
run dbutil.pub.sys

HP32215B.04.61 IMAGE/3000: DBUTIL  (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1978

>>cre orders
Data base ORDERS has been CREATED.
>>exit
END OF PROGRAM
:
```

The database is reloaded using DICTDBL.

Figure 9-23. Reloading the Database with DICTDBL

```
run dictdbl.pub.sys

DICTIONARY/3000 DB LOADER HP32244A.02.01 - (C) Hewlett-Packard Co. 1984

STORE FILE> mpestore
LIST FILE>
BASE: ORDERS.CUSTOMIZ.MILLER
RUN MODE(LOAD/EDIT/SHOW/EXIT)>
NEW BASE NAME>
BASE PASSWORD>
MODE>
FAST I/O(Y/N)?>
CUSTOMER          M 2/100
ZIPCODE         ITEM NOT FOUND, NEW ITEM NAME>   <cr> = not reloaded
2 ENTRIES LOADED IN <1 CPU-SEC
PARTS             M 2/100
2 ENTRIES LOADED IN <1 CPU-SEC
INVENTORY         D 3/120
3 ENTRIES LOADED IN <1 CPU-SEC
ORDERHEAD         D 2/100
2 ENTRIES LOADED IN <1 CPU-SEC
ORDERLINE         D 3/100
3 ENTRIES LOADED IN <1 CPU-SEC
LOAD COMPLETED
END OF PROGRAM
:
```

Finally, we recompile the Transact program and implement the new application.

Figure 9-24. Compiling Transact Program to Resolve Data Changes

```
run trancomp.pub.sys
TRANSACT/3000 COMPILER   HP32247A.02.02 - (C) Hewlett-Packard Co. 1984

SOURCE FILE> custup
LIST FILE>
CONTROL> dBlist
TRANSACT/3000 COMPILER  A.02.02 : TUE, MAY 14, 1985,  9:32 AM COMPILED

LISTING OF FILE CUSTUP.CUSTOMIZ.MILLER            PAGE 1
COMPILING WITH OPTIONS: CODE,DICT,ERRS
CODE FILE STATUS: REPLACED
0 COMPILATION ERRORS
PROCESSOR TIME=00:00:08
END OF PROGRAM
:
```

DICTDBU and DICTDBL cannot handle all types of element changes. For example, cust-no was changed from a 9(4) definition to an I+(4) definition in this example because these utilities will not convert a numeric ASCII element properly. This is because Image does not have a data type corresponding to numeric ASCII. DICTDBC creates an element defined as numeric ASCII as an alphanumeric element of type X. Thus, if cust-no were being changed from 9(4) to 9(6), DICTDBU would unload it as X(4). DICTDBL would reload it as X(6) causing the new field to be left justified with two spaces inserted on the right. Transact would no longer be able to interpret the field as numeric.

Thus, when writing a custom application, avoid using data type 9 or write a utility to convert data after DICTDBU has run and before DICTDBL has run.

User Exits

In general, there are two types of application development environments. First there are application software companies who build application solutions to sell to other companies. Second, there are companies who build application solutions for use internally. Localization is attractive to both environments. Ignoring the fact that the software is sold in one case, both types of environments have a similar structure. There is a central group responsible for maintaining and enhancing the core system. There are one or more user organizations who accept the basic system functionality, but who have unique needs from the other users. Until these needs become the common needs of the majority of the system users, the central group typically resists adding the functionality to the core system.

If, however, the central group provides ways in which the individual users can modify the system to include the functionality they need without destroying the functionality of the core system, then both groups become winners.

The concepts discussed earlier, plus the concept of user exits provide this capability.

It is probably much easier for the software engineers developing software for sale to build in user exits, since they are acutely aware of the many unique demands their customers make.

It is no doubt much more difficult for a software engineer building internal software to distinguish between capabilities that should be a part of the core system versus capabilities that should be extensions or localization of the core system, since his users are also internal.

Providing the capability for user exits and deciding on the timing of this capability are up to the designer of the core system. For example, a designer may only want to allow user intervention after data has been entered. Another designer may want to allow user intervention before and after data entry, as well as before and after database update.

Naturally, the more a designer provides this capability, the better the possibility that the user can solve his unique problem outside of the core system.

The example below illustrates one way to implement user exits within Transact. It implements a structure that allows the user to do some processing just after data has been entered.

Figure 9-25. Providing User Exits

```
 1     system custex,base=orders
 2                  ,vpls=formfile
 3                  ,file=formxref,exitxref;
 4     define(item) menuname x(16):
 5                  fkey     9(2):
 6                  screen   x(16):
 7                  lastkey i(4):
 8                  userexit-prog x(6):
 9                  userexit-marker @;
10     list menuname:
11          lastkey:
12          userexit-prog;
13
14    data menuname; <<to simulate transfer of control to this subroutine>>
15
16     <<
17     ************************************************************
18     Subroutine: to update customer information
19
20     input: menuname - contains the name of the screen to be displayed
21
22     output: none
23     ************************************************************
24     >>
25     list fkey:
26          screen;
27     list userexit-marker;
28     list(auto) custupd-global;
29     let (yes) = 1;
30     let (no) = 0;
31     let (error) = (no);
32     level;
33     list(auto) customer;
34     if (error) = (no)
35       then
36         get(form) (menuname),init
37                              ,window=(" ")
38                              ,fkey=lastkey
39                              ,autoread
40       else
41         get(form) (menuname)
42                              ,fkey=lastkey
43                              ,autoread;
44     if (lastkey) = 8
45       then exit;
46     set(match) list (menuname);
47     get(serial) exitxref,list=(menuname,userexit-prog);
```

Figure 9-26. Providing User Exits (Continued)

```
48      call (userexit-prog),data=userexit-marker;
49      if (error) = (yes)
50        then end;
51      if (lastkey) = 0
52        then perform modify
53      else
54        do
55        set(match) list (menuname);
56        let (fkey) = (lastkey);
57        set(match) list (fkey);
58        get(serial) formxref,list=(menuname,fkey,screen);
59        reset(option) match;
60        perform modify;
61        doend;
62      end;
63
64      modify:
65
66        set(key) list (cust-no);
67        get customer,list=(@);
68        put(form) (screen),window=("update? - f1=yes, f2=no");
69        get(form) (screen),f1(autoread)=modify-f1
70                           ,f2=modify-f2;
71
72      modify-f1:
73
74        update customer,list=(@);
75
76      modify-f2:
77
78        end;
```

The user exit is established in a way similar to that used to achieve form independence. A cross reference file is set up to contain the name of the subprogram to be called based upon the name of the form that the program is currently processing.

The content of this cross-reference file is:

Figure 9-27. User Exit cross Reference Table

```
              MENUNAME:         USEREXIT-PROG:
              -------------------------------
              CUSTUPDATEMM      CU1
```

MENUNAME is the index into the program specifying the current menu or form. USEREXIT-PROG contains the name of the Transact subprogram to be called.

As this program illustrates, there can be some data defined within the program for its own use (lines 10 through 26). This data could also be defined in the dictionary. There can also be global data that is of importance to both the program and the user exit program (lines 28 through 31). This data should be defined in the dictionary in order to make coding of the user exit program easier. Included in this data are elements for handling form data input errors (validation) and the data the user wants to add to the transaction. Finally, there is the dataset definition needed specifically for this generic transaction, also defined in the dictionary (line 33). The dictionary description of custupd-global and customer is as follows:

Figure 9-28. Setting up Transaction-Specific Data in the Dictionary

```
                        FILE custupd-global
FILE                  TYPE: RESPONSIBILITY:
  CUSTUPD-GLOBAL        FORM


     ELEMENT(ALIAS):             PROPERTIES:        ELEMENT(PRIMARY):
     PASSWORD                     X (8,0,8)          PASSWORD
     ERROR                        I (4,0,2)          ERROR
     YES                          I (4,0,2)          YES
     NO                           I (4,0,2)          NO

                        FILE customer


FILE                  TYPE: RESPONSIBILITY:
  CUSTOMER              MAST


     ELEMENT(ALIAS):             PROPERTIES:        ELEMENT(PRIMARY):
     CUST-NO              *       I+(6,0,4)          CUST-NO
     NAME                         X (20,0,20)        NAME
     STREET-ADDR                  X (20,0,20)        STREET-ADDR
     CITY-STATE                   X (20,0,20)        CITY-STATE
     AREA                        X (6,0,6)           AREA
```

Note that the user of the system has added the element password to the custupd-global list. This element is not a part of the core application.

The program depends upon the existence of error, yes, and no as the way in which the subprogram indicates to the main program that an error has been detected. The main program initializes these variables in lines 29 to 31.

The program sets up a marker element which it uses to denote the point in the list register that the subprogram has access to (line 27 and 48).

The form is displayed without erasing the information, if the user exit program detected an error. Otherwise an initialized form is displayed. Lines 34 to 45 handle this.

Lines 46 to 48 implement the user exit by searching for a match on form name in the cross-reference file. This code and file could be expanded to provide for multiple user exits during the same transaction and to make a user exit optional.

The user exit program follows. The user has decided to add a password to the customer update menu and has added the logic to his subprogram to validate the password.

Figure 9-29. User Exit Subprogram for Data Validation

```
 1     system cu1,vpls=formfile;
 2     list(auto) custupd-global;
 3     list(auto) customer;
 4     let (error) = (no);
 5     if (password) <> "OKAY"
 6       then
 7         do
 8         set(form) *,window=(password,"invalid password");
 9         let (error) = (yes);
10         doend;
11     exit;
```

Note that the user exit program must contain the LIST definition corresponding to the main program definition that occurs after the marker item. Standardized procedures for communicating error conditions, etc. must also exist.

The modified user version of the form custupdatemm and the dictionary description of this form follows:

Figure 9-30. User-Modified Customer Main Menu, Adding an Element

```
custupdatemm            customer update main menu

             enter customer number      [      ]

                        f1 -  marketing       (custupdate1)
                        f2 -  finance        (custupdate2)
                        f3 -  accounts payable  (custupdate3)

              ********************************************
                                or
                       enter screen name [               ]

                        password [       ]

market-  finance  accounts                                    exit
ing               payable
```

Figure 9-31. Dictionary Definitions of Modified Customer Main Menu

```
FILE                TYPE: RESPONSIBILITY:
CUSTUPDATEMM        FORM

ELEMENT(ALIAS):           PROPERTIES: ELEMENT(PRIMARY):
CUST-NO                   I+(6,0,4)          CUST-NO
SCREEN                    X (16,0,16)        SCREEN
PASSWORD                  X (8,0,8)          PASSWORD
```

Transactions Across Multiple Datasets

All of the above concepts are still valid even if the transaction affects multiple datasets. The following program illustrates a way to write generic code that accesses more than one dataset. This code could be expanded to include the topics previously discussed to provide form independence, user exits, etc.

Figure 9-32. One Screen, Multiple Dataset Generic Transaction

```
 1     system addprt,base=orders
 2                  ,vpls=formfile;
 3     list(auto) addpart-global;
 4     level;
 5     list(auto) partvendors;
 6     level;
 7     list(auto) inventory;
 8     level;
 9     list(auto) parts;
10     get(form) addpart,init;
11     put parts,list=(@);
12     move (global-part) = (part-number);
13     end(level);
14     move (part-number) = (global-part);
15     put inventory,list=(@);
16     end(level);
17     move (part-number) = (global-part);
18     put partvendors,list=(@);
19     end;
```

The generic transaction adds a new record to the parts dataset, which is the master dataset. It then adds a record to each of two detail sets, inventory and partvendors. Part-number is a critical element and is common to all three sets.

The dictionary description of the lists used by the program are as follows:

Figure 9-33. Dictionary Definitions for One Screen, Multi Dataset Transaction

```
FILE                TYPE: RESPONSIBILITY:
ADDPART             FORM

      ELEMENT(ALIAS):            PROPERTIES:        ELEMENT(PRIMARY):
      PART-NUMBER                X (8,0,8)           PART-NUMBER
      DESCRIPTION                X (20,0,20)         DESCRIPTION
      LOCATION                   X (4,0,4)           LOCATION
      QUANTITY                   I (6,0,4)           QUANTITY
      VENDOR-CODE                X (6,0,6)           VENDOR-CODE
      VENDOR-NAME                X (20,0,20)         VENDOR-NAME

FILE                TYPE: RESPONSIBILITY:
ADDPART-GLOBAL      FORM

      ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
       GLOBAL-PART                X (8,0,8)           GLOBAL-PART

FILE                TYPE: RESPONSIBILITY:
INVENTORY           DETL

      ELEMENT(ALIAS):                PROPERTIES:        ELEMENT(PRIMARY):
      PART-NUMBER          *         X (8,0,8)           PART-NUMBER
                            CHAIN MASTER SET: PARTS
      LOCATION                       X (4,0,4)           LOCATION
      QUANTITY                       I (6,0,4)           QUANTITY

FILE                TYPE: RESPONSIBILITY:
PARTS               MAST

      ELEMENT(ALIAS):                PROPERTIES:        ELEMENT(PRIMARY):
      PART-NUMBER          *         X (8,0,8)           PART-NUMBER
      DESCRIPTION                    X (20,0,20)         DESCRIPTION

FILE                TYPE: RESPONSIBILITY:
PARTVENDORS         DETL

      ELEMENT(ALIAS):                PROPERTIES:        ELEMENT(PRIMARY):
      PART-NUMBER          *         X (8,0,8)           PART-NUMBER
                            CHAIN MASTER SET: PARTS
      VENDOR-CODE                    X (6,0,6)           VENDOR-CODE
      VENDOR-NAME                    X (20,0,20)         VENDOR-NAME
```

Note that the global definitions for this transaction include an element called global-part. This element is used to store the value of part-number between dataset updates as explained below.

The form addpart looks like this:

Figure 9-34. Multiple Dataset Screen

```
addpart                    add a part

              part number [        ]

              description [                    ]

              location    [    ]

              quantity    [       ]

              vendor code [      ]

              vendor name [                    ]
```

The key to understanding how to write generic code is to understand how the VPLUS and Image interface work with the list register.

The first thing to understand is that the list register can have as many definitions of an element on it as you want. However, Transact always references the latest definition. Thus when we do a LIST(AUTO) for each dataset that the transaction is to access, we are putting three definitions of part-number in the list register.

The VPLUS interface with the dictionary does not require us to use the LIST= option. If this is left off, Transact matches the elements that are a part of the form with the current contents of the list register. These elements can occur anywhere physically in the list register. Elements are resolved by starting at the end (most recent change) of the list register, and working back until the element definition is found (line 10).

The Image interface through LIST=(@) requires the element list to be contiguous. We cannot list the individual elements, since this would defeat the idea of creating custom code. Thus after updating the parts dataset (line 11), we need to save the value of part-number (line 12) and then remove all of the parts dataset elements (line 13), then restore part-number which now will be the part-number defined for dataset inventory (line 14). A record is then added to the inventory dataset.

Since part-number has already been saved, we do not need to save it again, but can now remove the elements that belong to the inventory set from the LIST and then restore part-number which now becomes the part-number for the partvendors set.

This same logic can be repeated any number of times. Similar logic also handles data retrieval from different sets.

A Building the Dictionary

The following is an example of a terminal session creating the dictionary.

After the dictionary is created, then it is loaded with database definitions, form definitions, and elements. Appendices B,C, E, and F are examples of this.

```
     :run dictinit.pub.sys
DICTIONARY/3000 INITIALIZATION HP32244A.02.00 - (C) Hewlett-Packard Co. 1983
     Initialization/Re-initialization (I/R) >i
     USER PASSWORD >
     ACCOUNT PASSWORD >
     GROUP PASSWORD >
      #J70
     END OF PROGRAM
     :
     FROM/J70 MARV.MILLER/PLEASE RUN DICTINIT.PUB.SYS,UPDATE
     run dictinit.pub.sys,update
DICTIONARY/3000 INITIALIZATION HP32244A.02.00 - (C) Hewlett-Packard Co. 1983
     Initialization/Re-initialization (I/R) >i
     Dictionary capacities: Default or Provided (D/P) >d
     DATA-ELEMENT     will have capacity 1001
     DATA-FILE        will have capacity 503
     DATA-PROCEDURE   will have capacity 203
     DATA-CATEGORY    will have capacity 203
     DATA-GROUP       will have capacity 503
     DATA-CLASS       will have capacity 203
     DATA-LOCATION    will have capacity 203
     LINK-FILE        will have capacity 401
     LINK-ELEMENT     will have capacity 401
     LINK-DESCRIPTION will have capacity 2003
     DATA-REPORTLOC   will have capacity 503
     ELEMENT-REFTYPE  will have capacity 500
     ELEMENT-ELEMENT  will have capacity 500
     FILE-FILE        will have capacity 250
```

```
PROCEDURE-PROCED will have capacity 100
CATEGORY-CATEGOR will have capacity 100
GROUP-GROUP      will have capacity 250
FILE-ELEMENT     will have capacity 2000
FILE-EL-SECOND   will have capacity 100
FILE-PATH        will have capacity 400
FILE-SORT        will have capacity 400
Press RETURN to continue >
PROCEDURE-ELEMEN will have capacity 500
CATEGORY-ELEMENT will have capacity 500
GROUP-ELEMENT    will have capacity 1000
CLASS-CLASS      will have capacity 500
CLASS-ELEMENT    will have capacity 3000
CLASS-FILE       will have capacity 500
CLASS-GROUP      will have capacity 500
FILE-LOCATION    will have capacity 500
DESCRIPTION-TEXT will have capacity 5000
REPORT-LIST      will have capacity 500
Are the capacities correct? (Y/N) >y
Password for MANAGER access >dbmgr
Password for PROGRAMMER access >dbprog
Password for INFORM access >dbinf
Password for DOCUMENTATION access >dbdoc
Password for REPORT access >dbrpt
Password for MANAGER access will be dbmgr
Password for PROGRAMMER access will be dbprog
Password for INFORM access will be dbinf
Password for DOCUMENTATION access will be dbdoc
Password for REPORT access will be dbrpt
Are the passwords correct? (Y/N) >y
USER PASSWORD >
ACCOUNT PASSWORD >
GROUP PASSWORD >
 #J74
END OF PROGRAM
:
FROM/J74 MARV.MILLER/Dictinit is complete
```

B Entering the Database Definition

The following is an example of entering database and file definitions into the dictionary. The IMAGE database ORDERS, KSAM file KCUST, and MPE files BATCHINV and SHORTAGE which are all used by the examples of this manual, are the definitions entered by this example. The example is also complete, that is it completely defines the files and database.

The example shows the minimum data that must be added. The other data, such as heading text, entry text, and edit masks, when entered helps to customize the application.

```
:file dict.pub=dict
:run dictdbm.pub.sys
DICTIONARY/3000  HP32244A.02.00  - (C) Hewlett-Packard Co. 1983
 PASSWORD FOR DICT.PUB>
FORMS ENTRY(Y/N)?>
 > repeat create file
                   FILE orders
              LONG NAME
                   TYPE base
          RESPONSIBILITY
            DESCRIPTION
                   FILE customer
              LONG NAME
                   TYPE mast
          RESPONSIBILITY
            DESCRIPTION
                   FILE order
              LONG NAME
                   TYPE auto
          RESPONSIBILITY
            DESCRIPTION
                   FILE parts
              LONG NAME
                   TYPE mast
          RESPONSIBILITY
```

```
            DESCRIPTION
                    FILE orderhead
              LONG NAME
                    TYPE detl
          RESPONSIBILITY
            DESCRIPTION
                    FILE orderline
              LONG NAME
                    TYPE detl
          RESPONSIBILITY
            DESCRIPTION
                    FILE inventory
              LONG NAME
                    TYPE detl
          RESPONSIBILITY
            DESCRIPTION
                    FILE kcust
              LONG NAME
                    TYPE ksam
          RESPONSIBILITY
            DESCRIPTION
ADDITIONAL FILE ATTRIBUTES(N/Y)?>
                    FILE batchinv
              LONG NAME
                    TYPE mpef
          RESPONSIBILITY
            DESCRIPTION
ADDITIONAL FILE ATTRIBUTES(N/Y)?>
                    FILE shortage
              LONG NAME
                    TYPE mpef
          RESPONSIBILITY
            DESCRIPTION
ADDITIONAL FILE ATTRIBUTES(N/Y)?>
                    FILE
> relate file
             PARENT FILE orders
              CHILD FILE customer
```

```
                CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE order
                 CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE parts
                 CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE orderhead
                 CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE orderline
                 CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE inventory
                 CHILD ALIAS
                   CAPACITY 100
                   BLOCKMAX
                DESCRIPTION
                  CHILD FILE
> repeat create element
                    ELEMENT cust-no
                  LONG NAME
                        TYPE 9
                        SIZE 4
                    DECIMAL
   STORAGE LENGTH(4)
                   COUNT(1)
```

```
            HEADING TEXT
              ENTRY TEXT
               EDIT MASK
       MEASUREMENT UNITS
BLANK WHEN ZERO(N/Y)?>
          RESPONSIBILITY
            DESCRIPTION
                 ELEMENT name
              LONG NAME
                    TYPE x
                    SIZE 20
    STORAGE LENGTH(20)
                COUNT(1)
            HEADING TEXT
              ENTRY TEXT
               EDIT MASK
       MEASUREMENT UNITS
    RIGHT JUSTIFY(N/Y)?>
          RESPONSIBILITY
            DESCRIPTION
                 ELEMENT street-addr
              LONG NAME
                    TYPE x
                    SIZE 20
    STORAGE LENGTH(20)
                COUNT(1)
            HEADING TEXT
              ENTRY TEXT
               EDIT MASK
       MEASUREMENT UNITS
    RIGHT JUSTIFY(N/Y)?>
          RESPONSIBILITY
            DESCRIPTION
                 ELEMENT city-state
              LONG NAME
                    TYPE x
                    SIZE 20
    STORAGE LENGTH(20)
```

```
            COUNT(1)
         HEADING TEXT
           ENTRY TEXT
            EDIT MASK
   MEASUREMENT UNITS
RIGHT JUSTIFY(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
             ELEMENT zipcode
          LONG NAME
               TYPE x
               SIZE 6
STORAGE LENGTH(6)
            COUNT(1)
         HEADING TEXT
           ENTRY TEXT
            EDIT MASK
   MEASUREMENT UNITS
RIGHT JUSTIFY(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
             ELEMENT order-no
          LONG NAME
               TYPE x
               SIZE 8
STORAGE LENGTH(8)
            COUNT(1)
         HEADING TEXT
           ENTRY TEXT
            EDIT MASK
   MEASUREMENT UNITS
RIGHT JUSTIFY(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
             ELEMENT order-status
          LONG NAME
               TYPE x
               SIZE 2
```

```
        STORAGE LENGTH(2)
                    COUNT(1)
                HEADING TEXT
                  ENTRY TEXT
                   EDIT MASK
           MEASUREMENT UNITS
        RIGHT JUSTIFY(N/Y)?>
              RESPONSIBILITY
                DESCRIPTION
                     ELEMENT order-date
                  LONG NAME
                        TYPE x
                        SIZE 6
        STORAGE LENGTH(2)
                    COUNT(1)
                HEADING TEXT
                  ENTRY TEXT
                   EDIT MASK
           MEASUREMENT UNITS
        RIGHT JUSTIFY(N/Y)?>
              RESPONSIBILITY
                DESCRIPTION
                     ELEMENT line-no
                  LONG NAME
                        TYPE 9
                        SIZE 2
                     DECIMAL
        STORAGE LENGTH(2)
                    COUNT(1)
                HEADING TEXT
                  ENTRY TEXT
                   EDIT MASK
           MEASUREMENT UNITS
      BLANK WHEN ZERO(N/Y)?>
              RESPONSIBILITY
                DESCRIPTION
                     ELEMENT quantity
                  LONG NAME
```

```
                TYPE i
                SIZE 6
             DECIMAL
STORAGE LENGTH(4)
           COUNT(1)
        HEADING TEXT
          ENTRY TEXT
           EDIT MASK
   MEASUREMENT UNITS
 SYNCHRONIZED(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
             ELEMENT part-number
          LONG NAME
                TYPE x
                SIZE 8
STORAGE LENGTH(8)
           COUNT(1)
        HEADING TEXT
          ENTRY TEXT
           EDIT MASK
   MEASUREMENT UNITS
RIGHT JUSTIFY(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
             ELEMENT description
          LONG NAME
                TYPE x
                SIZE 20
STORAGE LENGTH(20)
           COUNT(1)
        HEADING TEXT
          ENTRY TEXT
           EDIT MASK
   MEASUREMENT UNITS
RIGHT JUSTIFY(N/Y)?>
      RESPONSIBILITY
        DESCRIPTION
```

```
                    ELEMENT location
                LONG NAME
                      TYPE x
                      SIZE 4
      STORAGE LENGTH(4)
                 COUNT(1)
              HEADING TEXT
                ENTRY TEXT
                 EDIT MASK
         MEASUREMENT UNITS
      RIGHT JUSTIFY(N/Y)?>
            RESPONSIBILITY
              DESCRIPTION
                   ELEMENT back-order
                LONG NAME
                      TYPE i
                      SIZE 6
      STORAGE LENGTH(4)
                 COUNT(1)
              HEADING TEXT
                ENTRY TEXT
                 EDIT MASK
         MEASUREMENT UNITS
      RIGHT JUSTIFY(N/Y)?>
            RESPONSIBILITY
              DESCRIPTION
                   ELEMENT
  > repeat add file
                      FILE order
                KEY ELEMENT order-no
              ELEMENT ALIAS
               DESCRIPTION
                      FILE customer
                KEY ELEMENT cust-no
              ELEMENT ALIAS
               DESCRIPTION
                   ELEMENT name
              ELEMENT ALIAS
```

```
            DESCRIPTION
                 ELEMENT street-addr
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT city-state
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT zipcode
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT
                    FILE parts
             KEY ELEMENT part-number
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT description
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT
                    FILE orderhead
                 ELEMENT order-no
           ELEMENT ALIAS
       PATH MASTER FILE order
           SORT ELEMENT
PRIMARY PATH(N/Y)?>
           DESCRIPTION
                 ELEMENT cust-no
           ELEMENT ALIAS
       PATH MASTER FILE customer
           SORT ELEMENT
PRIMARY PATH(N/Y)?>
           DESCRIPTION
                 ELEMENT order-status
           ELEMENT ALIAS
       PATH MASTER FILE
           DESCRIPTION
                 ELEMENT order-date
           ELEMENT ALIAS
```

```
        PATH MASTER FILE
            DESCRIPTION
                 ELEMENT
                    FILE orderline
                 ELEMENT order-no
           ELEMENT ALIAS
        PATH MASTER FILE order
            SORT ELEMENT
PRIMARY PATH(N/Y)?>
            DESCRIPTION
                 ELEMENT line-no
           ELEMENT ALIAS
        PATH MASTER FILE
            DESCRIPTION
                 ELEMENT part-number
           ELEMENT ALIAS
        PATH MASTER FILE parts
            SORT ELEMENT
PRIMARY PATH(N/Y)?>
            DESCRIPTION
                 ELEMENT quantity
           ELEMENT ALIAS
        PATH MASTER FILE
            DESCRIPTION
                 ELEMENT
                    FILE inventory
                 ELEMENT part-number
           ELEMENT ALIAS
        PATH MASTER FILE parts
            SORT ELEMENT
PRIMARY PATH(N/Y)?>
            DESCRIPTION
                 ELEMENT location
           ELEMENT ALIAS
        PATH MASTER FILE
            DESCRIPTION
                 ELEMENT quantity
           ELEMENT ALIAS
```

```
        PATH MASTER FILE
            DESCRIPTION
                 ELEMENT
                    FILE shortage
PRIMARY/SECONDARY (P/S)?>
                 ELEMENT cust-no
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT part-number
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT order-no
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT order-date
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT line-no
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT quantity
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT back-order
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT
                    FILE batchinv
PRIMARY/SECONDARY (P/S)?>
                 ELEMENT part-number
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT location
           ELEMENT ALIAS
            DESCRIPTION
                 ELEMENT quantity
           ELEMENT ALIAS
            DESCRIPTION
```

```
                ELEMENT
                   FILE kcust
PRIMARY/SECONDARY (P/S)?>
                ELEMENT cust-no
          ELEMENT ALIAS
     KEY ELEMENT(N/Y)?> y
     PRIMARY KEY(N/Y)?> y
     DUPLICATES (N/Y)?> n
           DESCRIPTION
                ELEMENT name
          ELEMENT ALIAS
     KEY ELEMENT(N/Y)?> y
     DUPLICATES (N/Y)?> y
           DESCRIPTION
                ELEMENT
                   FILE
>exit
END OF PROGRAM
```

C Loading Definitions from IMAGE

Appendix B showed how to enter a database definition manually into the dictionary.

If the IMAGE database already exists, it is not necessary to load the data manually into the dictionary. DICTDBD is a dictionary utility that loads the database definitions for you. The following is an example of loading the definitions for the database ORDERS into the dictionary.

```
     :file dict.pub=dict
     :run dictdbd.pub.sys
DICTIONARY/3000 DB INFO LOADER HP32244A.02.00 - (C) Hewlett-Packard Co. 1983
     DICTIONARY PASSWORD>
     BASE> orders
     BASE PASSWORD>
     MODE> 1
     LOADING DATA DICTIONARY
     END OF PROGRAM
     :
```

D Creating the Physical Database

If the IMAGE database has not been created, then after entering in the database definition to the dictionary as shown in appendix B, this definition is used by DICTDBC to create the IMAGE root file for the database.

The example below shows the physical creation process for the database ORDERS, KSAM file KCUST, and MPE files SHORTAGE and BATCHINV. These are the database and files used by the examples throughout the manual.

```
     :run dictdbc.pub.sys
DICTIONARY/3000 DB CREATOR HP32244A.02.00 - (C) Hewlett-Packard Co. 1983
     DICTIONARY PASSWORD>
     BASE> orders
     CONTROL LINE> nolist
     SCHEMA FILE>
     LISTING FILE>
     APPLY SECURITY JUST TO SET LEVEL(N/Y)?>
     SCHEMA GENERATION
     DBSCHEMA PROCESSOR
     PAGE 1      HEWLETT-PACKARD 32215B.03.10 IMAGE/3000: DBSCHEMA
             THU, OCT 18, 1984,  2:27 PM  (C) HEWLETT-PACKARD CO. 1978
     $CONTROL NOLIST
      DATA SET      TYPE  FLD  PT  ENTR  MED  CAPACITY  BLK  BLK   DISC
        NAME              CNT  CT  LGTH  REC            FAC  LGTH  SPACE
     CUSTOMER          M  5    1   35    45    100      11   496   44
     PARTS             M  2    2   14    29    100      13   378   27
     ORDER             A  1    2   4     19    100      20   382   18
     ORDERHEAD         D  3    2   7     15    100      25   377   15
     ORDERLINE         D  4    2   11    19    100      20   382   18
     INVENTORY         D  3    1   8     12    120      40   483   16
                               TOTAL DISC SECTORS INCLUDING ROOT: 149
     NUMBER OF ERROR MESSAGES: 0
     ITEM NAME COUNT: 13      DATA SET COUNT: 6
     ROOT LENGTH: 585     BUFFER LENGTH: 496     TRAILER LENGTH: 256
     ROOT FILE ORDERS CREATED.
```

```
     END OF PROGRAM
     :run dbutil.pub.sys
HP32215B.03.10 IMAGE/3000: DBUTIL  (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1978
     >>cre orders
       data base ORDERS has been CREATED.
     >>exit
     END OF PROGRAM
     :build shortage;rec=-36,,,ascii;disc=100
     :build batchinv;rec=-16,,,ascii;disc=100
     :run ksamutil.pub.sys
     >build kcust;rec=-24,,,ascii;keyfile=keycust >
;key=numeric,1,4 >           ;key=byte,5,20,,duplicate
     KCUST.group.acct & KEYCUST ARE CREATED.
     >exit
     END OF PROGRAM
     :
```

E Entering Form Definitions

The following is an example of entering VPLUS form definitions into the dictionary. The appendix uses the formfile used by the examples in this manual to illustrate loading the definitions. The example is complete in that it shows how to load each form used in the manual.

```
:run dictdbm
DICTIONARY/3000  HP32244A.02.00  - (C) Hewlett-Packard Co. 1983
 PASSWORD FOR DICT.PUB>
FORMS ENTRY(Y/N)?>
> repeat create file
                    FILE formfile
               LONG NAME
                    TYPE vpls
          RESPONSIBILITY
             DESCRIPTION
                    FILE mainmenu
               LONG NAME
                    TYPE form
          RESPONSIBILITY
             DESCRIPTION
                    FILE vcustno
               LONG NAME
                    TYPE form
          RESPONSIBILITY
             DESCRIPTION
                    FILE vcustomer
               LONG NAME
                    TYPE form
          RESPONSIBILITY
             DESCRIPTION
                    FILE vcustomerrd
               LONG NAME
                    TYPE form
```

```
            RESPONSIBILITY
               DESCRIPTION
                      FILE vcustomerrh
                 LONG NAME
                      TYPE form
            RESPONSIBILITY
               DESCRIPTION
                      FILE vorderhead
                 LONG NAME
                      TYPE form
            RESPONSIBILITY
               DESCRIPTION
                      FILE vorderline
                 LONG NAME
                      TYPE form
            RESPONSIBILITY
               DESCRIPTION
                      FILE
> repeat add file
                      FILE vcustno
                   ELEMENT cust-no
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT
                      FILE vcustomer
                   ELEMENT cust-no
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT name
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT street-addr
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
```

```
      ELEMENT city-state
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT zipcode
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT
         FILE vcustomerrd
      ELEMENT cust-no
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT name
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT street-addr
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT city-state
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT zipcode
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT
         FILE vorderhead
      ELEMENT order-no
ELEMENT ALIAS
 FIELD NUMBER
  DESCRIPTION
      ELEMENT cust-no
ELEMENT ALIAS
```

```
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT order-status
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT order-date
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT
                      FILE vorderline
                   ELEMENT line-no
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT part-number
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT quantity
             ELEMENT ALIAS
              FIELD NUMBER
               DESCRIPTION
                   ELEMENT
                      FILE
> relate file
             PARENT FILE formfile
              CHILD FILE mainmenu
             CHILD ALIAS
             DESCRIPTION
              CHILD FILE vcustomer
             CHILD ALIAS
             DESCRIPTION
              CHILD FILE vorderhead
             CHILD ALIAS
             DESCRIPTION
              CHILD FILE vorderline
```

```
CHILD ALIAS
DESCRIPTION
 CHILD FILE vcustomerrh
CHILD ALIAS
DESCRIPTION
 CHILD FILE vcustomerrd
CHILD ALIAS
DESCRIPTION
 CHILD FILE vcustno
CHILD ALIAS
DESCRIPTION
 CHILD FILE
```

F Loading Form Definitions

The utility DICTVPD may be used to load form definitions to the dictionary. When the forms are defined using FORMSPEC, you must be careful to define the field names to be the same as the names you want to identify with the form in the dictionary.

For example, the following is the FORMSPEC listing of the form vorderhead.

```
********* ********* ********* ********* ********* ********* ********* *********
 vorderhead                    order data
     order number [order_no]   customer [cust ]    status  [st]  date [date  ]
                   ________              _____              __ ______
********* ********* ********* ********* ********* ********* ********* **********
 Field: order_no
    Num:14    Len: 8     Name: ORDER_NO         Enh: HI    FType: O DType: CHAR
    Init Value:
 Field: cust
    Num: 2    Len: 5     Name: CUST_NO          Enh: HI    FType: O DType: CHAR
    Init Value:
 Field: st
    Num: 3    Len: 2     Name: ORDER_STATUS     Enh: HI    FType: O DType: CHAR
    Init Value:
 Field: date
    Num: 4    Len: 6     Name: ORDER_DATE       Enh: HI    FType: O DType: CHAR
     Init Value:
```

This form definition was loaded into the dictionary as follows:

```
     :run dictvpd.pub.sys
DICTIONARY/3000 VPLUS LOADER HP32244A.02.00 - (C) Hewlett-Packard Co. 1983
     DICTIONARY PASSWORD>
     FORMS FILE NAME> formfile
     SELECT DATA CONVERSION (Default/Char)>
     DATA ELEMENTS ALREADY DEFINED (Y/N)> y
     LIST FILE>
     CHANGE UNDERSCORE TO HYPHEN (Y/N)> y
     LOADING DATA DICTIONARY
     FORM NAME TO BE LOADED (or "@"/"?")> vorderhead
     LOADING FORM: VORDER_HEAD
     FORM NAME TO BE LOADED (or "@"/"?")>
```

```
Name            Alias           NEW/OLD Type
FORMFILE                        NEW     VPLS
VORDERHEAD                      NEW     FORM
ORDER-NO                        OLD     X ( 8, 0, 8)
CUST-NO                         OLD     9 ( 4, 0, 4)
ORDER-STATUS                    OLD     X ( 2, 0, 2)
ORDER-DATE                      OLD     X ( 6, 0, 6)
END OF PROGRAM
:
```

Note that the form definition used the underscore character. DICTVPD will convert the underscore to a hyphen if directed to do so.

G Element and File Dictionary Reports

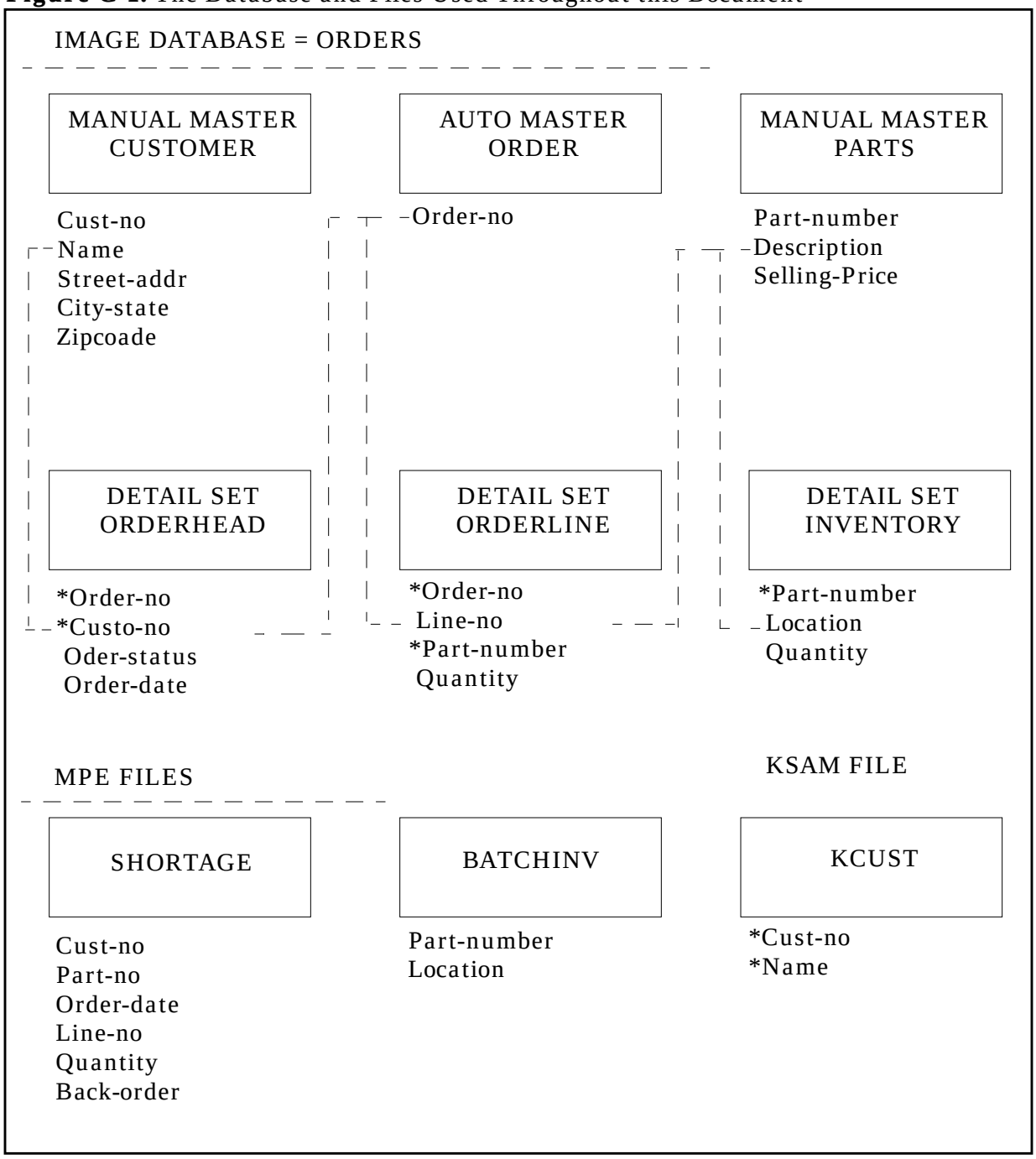
The following reports from DICTDBM list the elements and file definitions for the database and files used in the examples throughout this manual. Figure G-1 graphically portrays the relationship of the datasets and data items.

```
> report element
LIST OF DATA ELEMENTS DEFINED IN THE DICTIONARY
ELEMENT(PRIMARY):    TYPE: SIZE: DEC: LENGTH: COUNT:
 BACK-ORDER           I     6     0    4       1
 CITY-STATE           X     20    0    20      1
 CUST-NO              9     4     0    4       1
 DESCRIPTION          X     20    0    20      1
 LINE-NO              9     2     0    2       1
 LOCATION             X     4     0    4       1
 NAME                 X     20    0    20      1
 ORDER-DATE           X     6     0    6       1
 ORDER-NO             X     8     0    8       1
 ORDER-STATUS         X     2     0    2       1
 PART-NUMBER          X     8     0    8       1
 QUANTITY             I     6     0    4       1
 STREET-ADDR          X     20    0    20      1
 ZIPCODE              X     6     0    6       1
> show file
                  FILE orders
SHOW ALL FILE ELEMENTS(Y/N)?> y
FILE                 TYPE: RESPONSIBILITY:
 ORDERS               BASE
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 CUSTOMER         MAST      CUSTOMER             100
        ELEMENT(ALIAS):            PROPERTIES:        ELEMENT(PRIMARY):
          CUST-NO              *      9 (4,0,4)          CUST-NO
          NAME                        X (20,0,20)        NAME
          STREET-ADDR                 X (20,0,20)        STREET-ADDR
```

```
          CITY-STATE                  X (20,0,20)          CITY-STATE
          ZIPCODE                     X (6,0,6)            ZIPCODE
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 ORDER            AUTO      ORDER                100
       ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
          ORDER-NO             *      X (8,0,8)            ORDER-NO
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 PARTS            MAST      PARTS                100
       ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
          PART-NUMBER          *      X (8,0,8)            PART-NUMBER
          DESCRIPTION                 X (20,0,20)          DESCRIPTION
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 ORDERHEAD        DETL      ORDERHEAD            100
       ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
          ORDER-NO             *      X (8,0,8)            ORDER-NO
                                    CHAIN MASTER SET: ORDER
          CUST-NO              *      9 (4,0,4)            CUST-NO
                                    CHAIN MASTER SET: CUSTOMER
          ORDER-STATUS                X (2,0,2)            ORDER-STATUS
          ORDER-DATE                  X (6,0,6)            ORDER-DATE
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 ORDERLINE        DETL      ORDERLINE            100
       ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
          ORDER-NO             *      X (8,0,8)            ORDER-NO
                                    CHAIN MASTER SET: ORDER
          LINE-NO                     9 (2,0,2)            LINE-NO
          PART-NUMBER          *      X (8,0,8)            PART-NUMBER
                                    CHAIN MASTER SET: PARTS
          QUANTITY                    I (6,0,4)            QUANTITY
FILE(ALIAS):     TYPE:     FILE(PRIMARY):       CAPACITY:
 INVENTORY        DETL      INVENTORY            100
       ELEMENT(ALIAS):            PROPERTIES:         ELEMENT(PRIMARY):
          PART-NUMBER          *      X (8,0,8)            PART-NUMBER
                                    CHAIN MASTER SET: PARTS
          LOCATION                    X (4,0,4)            LOCATION
          QUANTITY                    I (6,0,4)            QUANTITY
> show file
                    FILE kcust
```

```
PRIMARY/SECONDARY (P/S)?>
FILE                 TYPE: RESPONSIBILITY:
 KCUST                KSAM
        ELEMENT(ALIAS):            PROPERTIES:        ELEMENT(PRIMARY):
          CUST-NO             !      9 (4,0,4)           CUST-NO
          NAME                *      X (20,0,20)         NAME
> show file
                    FILE shortage
PRIMARY/SECONDARY (P/S)?>
FILE                 TYPE: RESPONSIBILITY:
 SHORTAGE             MPEF
        ELEMENT(ALIAS):            PROPERTIES:        ELEMENT(PRIMARY):
          CUST-NO                    9 (4,0,4)           CUST-NO
          PART-NUMBER                X (8,0,8)           PART-NUMBER
          ORDER-NO                   X (8,0,8)           ORDER-NO
          ORDER-DATE                 X (6,0,6)           ORDER-DATE
          LINE-NO                    9 (2,0,2)           LINE-NO
          QUANTITY                   I (6,0,4)           QUANTITY
          BACK-ORDER                 I (6,0,4)           BACK-ORDER
> show file
                    FILE batchinv
PRIMARY/SECONDARY (P/S)?>
FILE                 TYPE: RESPONSIBILITY:
 BATCHINV             MPEF
        ELEMENT(ALIAS):            PROPERTIES:        ELEMENT(PRIMARY):
          PART-NUMBER                X (8,0,8)           PART-NUMBER
          LOCATION                   X (4,0,4)           LOCATION
          QUANTITY                   I (6,0,4)           QUANTITY
```

Figure G-1. The Database and Files Used Throughout this Document



H Application Forms Formats

MAINMENU

```
mainmenu                    Customer module

                            f1 = add customer

                            f2 = update customer

                            f3 = report customer

                            f8 = exit
```

VCUSTOMER

```
vcustomer                   customer data

                    number  [    ]

                    name    [                    ]

                    address [                    ]

                city,state  [                    ]

                  zipcode   [    ]
```

VCUSTNO

```
          vcustno


                        enter customer number [    ]
```

VCUSTOMERRD

```
                      [    ]   [                    ]
                               [                    ]
                               [                    ]
                               [      ]
```

VCUSTOMERRH

```
                                        customer  address
```

VORDERHEAD

```
       vorderhead                    order data

order number [       ]   customer [     ]    status  [  ]  date [   ]
```

VORDERLINE

```
       line number [  ]  part-number [        ]  quantity [       ]
```