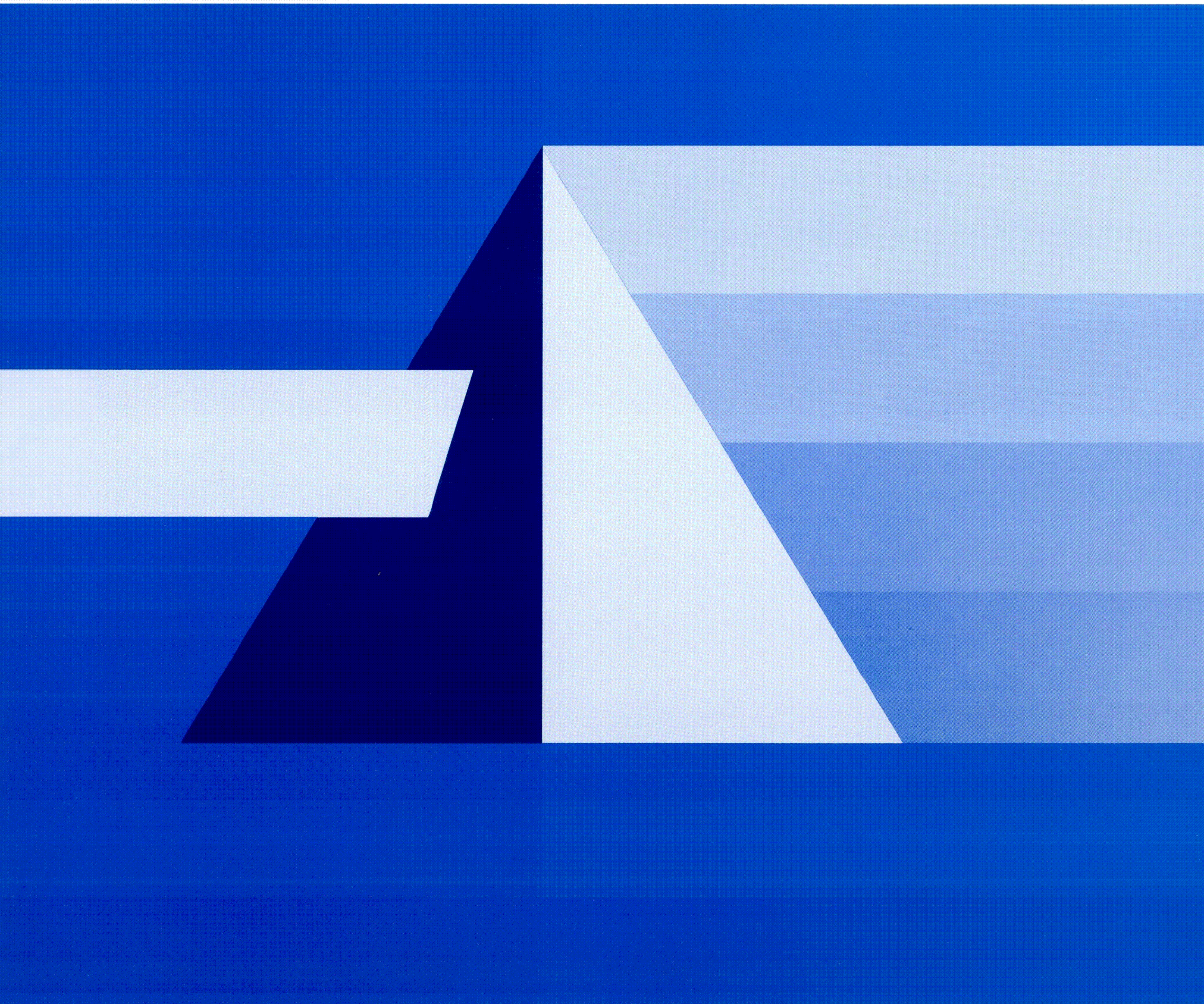
HP 3000 Computer Systems

HEWLETT PACKARD

Transact/3000 Reference Manual



HP 3000 Computer System

TRANSACT/3000 Reference Manual

HEWLETT PACKARD

19420 HOMESTEAD RD., CUPERTINO, CALIFORNIA 95014

Part No. 32247-90001 Printed in U.S.A. 12/82

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and the dates when pages were changed in updates to that edition. Within the manual, any page changed since the last edition has the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. When an update is incorporated in a subsequent reprinting of the manual, these bars are removed.

Second Edition	Dec 1982

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	Dec 1981	32247A.00
Second Edition	Dec 1982	32247A.03

PREFACE

This manual is a reference for programming in the Transact Programming Language. It assumes a working knowledge of computer programming and the HP 3000 computer system, including the subsystems IMAGE/3000 and VPLUS/3000. The manual contains the following sections:

Section 1. INTRODUCTION TO TRANSACT PROGRAMMING LANGUAGE, describes the features and benefits of Transact and illustrates Transact coding.

Section 2. TECHNICAL OVERVIEW OF TRANSACT/3000, describes the Transact/3000 Compiler, the Transact/3000 Transaction Processor, and Dictionary/3000.

Section 3. TRANSACT/3000 PROGRAMS, describes in detail how to write a Transact program. It includes information on statements, command sequences, comments, delimiters, and data items.

Section 4. TRANSACT DATA STORAGE REGISTERS, describes the areas of data storage in the transaction processor, called "registers", and how they work.

Section 5. RUNNING TRANSACT/3000, tells how to compile and execute Transact programs and control execution at run-time. It also discusses automatic and programmer-controlled error handling.

Section 6. TRANSACT/3000 VERBS, provides detailed specifications for using the Transact verbs, which are presented alphabetically.

Section 7. TRANSACT TEST FACILITY, tells how to use the test facility, which is a major aid in program testing, integration, and optimization.

PREFACE (continued)

Appendix A explains the error messages issued during Transact program compilation.

Appendix B explains processor error messages issued when user, program, system, or internal exceptions occur.

Appendix C contains flow charts illustrating the file or data base procedures called when Transact verbs perform file or data base operations.

Appendix D lists the intrinsics allowed in a DEFINE(INTRINSIC) statement.

Appendix E provides guidelines for optimizing the run-time performance and efficiency of Transact applications.

The following manuals and courses are recommended for additional reference or for practice in using Transact/3000.

Reference Manuals

Part Number	Title
30000-90009	*MPE Commands Reference Manual*
30000-90079	*KSAM/3000 Reference Manual*
32215-90003	*IMAGE/3000 Reference Manual*
30000-90010	*MPE Intrinsics Reference Manual*
32209-90001	*VPLUS/3000 Reference Manual*
32244-90001	*Dictionary/3000 Reference Manual*
32245-90001	*Report/3000 User's Guide*
32246-90001	*Inform/3000 User's Guide*

Self-Paced Courses:

Part Number	Title
22842A	*Programming in Transact/3000*
22843A	*Using Dictionary/3000*

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
[ ]	An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: [A] User may select A or [B] B or neither.

{ }	When several elements are stacked within braces the user *must* select one of these elements.
	
	{A}
	Example: {B} User must select A or B or C.
	{C}

italics	Lowercase italics denote a parameter which must be replaced by a user-supplied variable. Example: CLOSE *file-name*
...	A horizontal ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted. Example: ...[:*item-name*...]...;
upper case	Words in upper case appearing in syntax or format statements must be entered exactly as shown. Example: EXIT;

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

LIST OF TABLES

THE TRANSACT LANGUAGE

SECTION I

The Transact Programming Language (Transact) is a high-level computer language. Programs written in Transact are compiled by the Transact/3000 Compiler. The code produced by compilation is then executed by the Transact/3000 Transaction Processor. Both the processor and the compiler can be used in conjunction with Dictionary/3000 to develop information processing systems quickly and easily.

Transact is used in a variety of applications, including manufacturing, finance, and service, and it is used in a variety of industries, including electronics, communications, banking, oil, forestry, and entertainment. Transact users are typically applications programmers who design and implement information management systems. The Transact language provides these users with a means to develop programs rapidly. In particular, it provides a high-level interface to data management and data entry subsystems, and a built-in command structure that allows testing of the end-user interface during the early stages of program development.

This section contains a brief description of Transact features and benefits, followed by examples of Transact coding and how it works.

Transact: FEATURES AND BENEFITS

Transact's features make programming easy and fast, thereby increasing your effectiveness as a programmer. Because of the sophisticated nature of the language, the amount of Transact coding is significantly less than that required by other languages:

- What ordinarily requires many lines of code can be accomplished by one Transact statement.
- Data need not be defined in a Transact program because the compiler and processor can use the data dictionary for definitions.
- Low level intrinsic calls that interface with IMAGE data bases, KSAM and MPE files, and VPLUS/3000 forms files need not be coded in Transact.

The transaction processor, which actually executes compiled Transact programs, greatly enhances the efficiency of Transact and adds to end-user control of program execution. The processor has many built-in capabilities that reduce coding requirements:

- It handles data validation, display layout, and error procedures automatically.
- It enables the end user to control program execution by using special data entry characters and command qualifiers.

In addition, Transact programs can interface with HP 3000 subsystems including the intrinsics library, as well as programs coded in COBOL, FORTRAN, and other languages.

EXAMPLES OF Transact CODING

The following examples illustrate some of the features of the Transact Programming Language. The first two examples use *Command Sequences* to prompt the user for data; in the first example, the entered data is written to an IMAGE data set; in the second example, the user enters a key value that is used to select an entry from an IMAGE data set and display it at the terminal. The third example uses Transact's VPLUS interface to display a form, accept the data entered through that form, and then write it to an IMAGE data set.

Using a Command Sequence

Consider the following example of Transact code, which prompts the user for new customer information and then adds the information to a data set called CUST-MAST:

```
   $$ADD:
     $CUSTOMER:

       PROMPT CUST-NO ("Enter Customer Number"):
              CUST-NAME ("Enter Customer Name"):
              CUST-ADDR ("Enter Customer Address"):
              CUST-CITY ("Enter Customer City"):
              CUST-STATE ("Enter Customer State"):
              CUST-ZIP ("Enter Customer Zip");

       PUT CUST-MAST,
                 LIST=(CUST-NO:CUST-ZIP);
```

In a program that uses command sequences such as the above, Transact issues a prompt character (>) to which the user can respond with the command "ADD CUSTOMER":

```
> ADD CUSTOMER
```

Transact then executes the code in the ADD CUSTOMER command sequence shown in the example. That is, it prompts the user for customer information using the prompts specified in the PROMPT statement. The user responds with information about the customer:

```
Enter Customer Number> 30335
Enter Customer Name> XYZ Co.
Enter Customer Address> 33 Greenway, Seattle, WA, 98305
```

The PUT statement adds the new record to the data set CUST-MAST, and then prompts the user for another command. Notice that the user entered four responses separated by commas in response to the ENTER CUSTOMER ADDRESS> prompt. The processor automatically relates each response to its prompt and saves the user from having to wait for three extra prompts.

NOTE: The user must press a carriage return following each response line.

Using the Data Management Interface

The following example prompts the user for a key value and then uses this value to retrieve a customer entry from the data set CUST-MAST. The same statement that retrieves the entry also displays it at the user's terminal.

```
$$DISPLAY:
  $CUSTOMER:

     PROMPT(PATH) CUST-NO ("Enter Customer Number");
          CHECK=CUST-MAST;    <<Check that customer is in data base>>

     LIST CUST-NAME:
          CUST-ADDR:
          CUST-CITY:
          CUST-STATE:
          CUST-ZIP;

     FORMAT CUST-NO,     col 1,  head=" Customer:":
            CUST-NAME,   col 11, nohead:
            CUST-ADDR,   col 30, head=" Address:":
            CUST-CITY,   col 49, nohead:
            CUST-STATE,  col 62, nohead:
            CUST-ZIP,    col 65, nohead;

     OUTPUT CUST-MAST,
          LIST=(CUST-NO:CUST-ZIP);
```

Transact executes this code, when the end-user responds to the command prompt with "DISPLAY CUSTOMER":

```
     > DISPLAY CUSTOMER
```

The PROMPT statement asks the user to enter a customer number. This number is used to locate a particular entry in the data set CUST-MAST. If the entry is found, the OUTPUT statement displays all the values in the entry from CUST-NO through CUST-ZIP formatted according to the preceding FORMAT statement. If the entry is not found in CUST-MAST, Transact issues an error message and repeats the prompt for a customer number.

Assuming the data entered in the preceding example, the output looks like this:

```
Customer:                     Address:
30335    XYZ Co.              33 Greenway        Seattle       WA 98305
```

Using the VPLUS Interface

Assuming a VPLUS form called ADDFORM with six fields for customer information, the following code displays the form and retrieves the customer information entered by the end-user. If this information is not already in the data set, it writes the customer information to the data set CUST-MAST, and returns for another customer.

```
ADD:

    RESET(STACK) LIST;              << clear list register,         >>
    LIST CUST-NO:                   << and set it up for ADD        >>
         CUST-NAME:
         CUST-ADDR:
         CUST-CITY:
         CUST-STATE:
         CUST-ZIP;

ADD-CUSTOMER:

    GET(FORM) ADDFORM,
         INIT,
         LIST=(CUST-NO:CUST-ZIP),
         WINDOW=("Please enter a new customer"),
         F7=START-OF-PROGRAM,       << f7 key to restart program     >>
         F8=END-OF-PROGRAM,         << f8 key to end program         >>
         AUTOREAD;                  <<accept keys f1 thru f6 as ENTER>>

PUT-CUSTOMER:

    SET(KEY) LIST(CUST-NO);
    FIND CUST-MAST, LIST=();        << test if customer in base      >>
    IF STATUS <> 0 THEN             << customer already in base      >>
      DO
         MOVE (MESSAGE) =
           "Customer already exists, please press ENTER to continue.";
         GO TO ERROR-MESSAGE;
      DOEND;

    PUT CUST-MAST,                  << add customer to base          >>
          LIST=(CUST-NO:CUST-ZIP),
          ERROR=PUT-ERROR(*);       << process any PUT errors        >>

    GO TO ADD-CUSTOMER;

ERROR-MESSAGE:

    UPDATE(FORM) *,                 << issue error message and       >>
         LIST=(),                   << wait for user to press ENTER >>
         WAIT=F0,
         WINDOW=((MESSAGE)),
         F0=ADD-CUSTOMER;
```

When Transact executes this code, it places the terminal in block mode, displays the form ADDFORM with a message in the window and any FORMSPEC initialization. It then waits for the user to press the ENTER key, or any undefined function key. When the user presses ENTER, or any key except f7 or f8, Transact transfers user-entered data from the form to the data register.

It then checks whether this customer already exists, and if not, writes the customer information to the data set; if the customer is in the data set already, it displays a diagnostic message in the window area of the form, and waits for the user to press the ENTER key. When the AUTOREAD option is included, Transact accepts any function key not specified in an Fn option as an ENTER key.

Note that the list register is reset with RESET(STACK) LIST before setting up the list register for ADD. This is not necessary when using a command sequence (see previous examples) because Transact automatically resets the list register at the start of each new command sequence.

NOTE: When VPLUS forms are used for the end-user interface, there is no need for the command sequence structure shown in the previous examples. Since VPLUS operates in block mode, the user presses the ENTER key to enter an entire block of data instead of pressing carriage return to enter the response to a prompt.

TECHNICAL OVERVIEW

SECTION II

The Transact Programming Language is used in conjunction with the Transact/3000 Compiler, the Transact/3000 Transaction Processor, and, optionally, with Dictionary/3000. These components work together and with other HP 3000 system components and data storage facilities to process Transact programs. Figure 2-1 shows how these components interact and how the processor works with other HP 3000 systems and data storage facilities.

This section contains information on the following:

- Transact/3000 Compiler
- Transact/3000 Transaction Processor
- Dictionary/3000
- Interface with Other Systems and Data Storage Facilities

Section 5 tells how to use these components to run Transact programs. Specifically, it tells how to use the compiler and the processor and how to control Transact programs at run time.

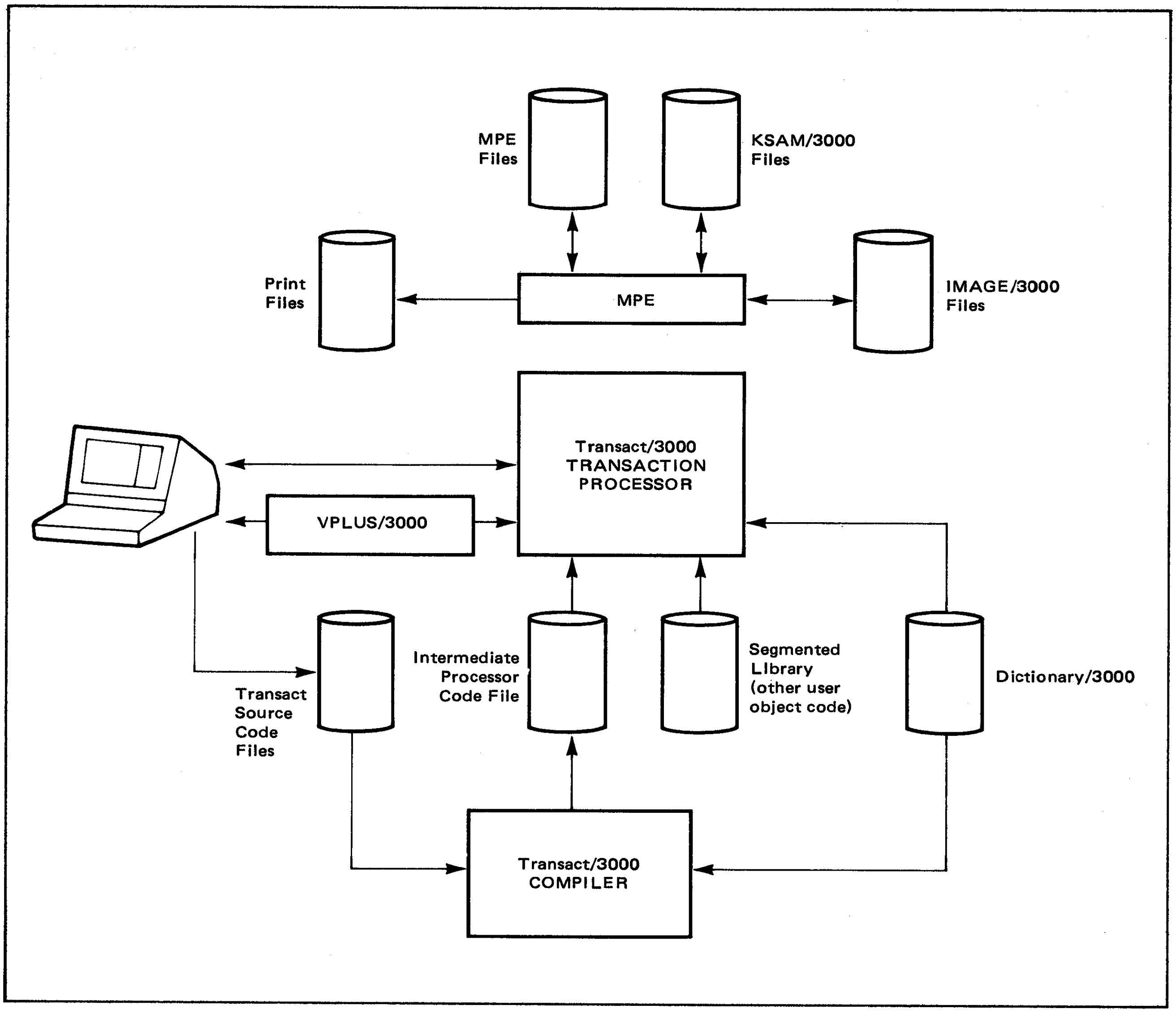

MPE Files
KSAM/3000 Files
Print Files
MPE
IMAGE/3000 Files
Transact/3000 TRANSACTION PROCESSOR
VPLUS/3000
Transact Source Code Files
Intermediate Processor Code File
Segmented Library (other user object code)
Dictionary/3000
Transact/3000 COMPILER

Figure 2-1. How Transact Works

THE TRANSACT COMPILER

As Figure 2-1 shows, the Transact/3000 compiler reads the Transact source code and generates a code file from it. The code file contains data tables and high-level instructions called "intermediate processor code". The Transact/3000 processor then executes this intermediate processor (object) code.

The compiler accesses the Dictionary/3000 data dictionary to resolve data definitions that are not defined in the program. Although you need not use the dictionary to define your data, using it means you need to define much less data within the program than in a traditional programming language.

You can control the source of input and the destination of output for the compiler. Options under your control include the listing of source code and the listing of data item definitions.

THE TRANSACT PROCESSOR

When the Transact processor executes the intermediate processor code, it is effectively executing your program.

The processor has these special features and benefits:

- Built-in capabilities eliminate many coding requirements that would otherwise be necessary. The processor handles aspects of data specification and validation as well as display layout and other supportive actions. These actions would normally need to be programmed. If you want to manage such actions yourself, you may override the default actions of the processor.

- Error-handling techniques simplify Transact programming and help to ensure effective processing. When the processor discovers an error, it automatically returns control to the program instruction where the error most logically occurred, thus saving you from having to code error routines. You can, however, override this automatic error handling.

- Several command modifiers can be used at execution time to enhance or modify the program procedures that you set up. For example, you can direct a display to the line printer, rather than to the terminal. You can also request that information be sorted before it is displayed. You need not program these options.

- Data items can be resolved at run time. The processor uses the data dictionary to resolve the main attributes of any data items whose definitions the compiler was not able to resolve. This allows you to code and compile programs before you define all data items.

DATA DICTIONARY

Using Dictionary/3000, you can define data items, data bases, and forms files used in Transact programs. Thus, it is not always necessary to define data items within the program itself. The data dictionary provides a central location for data definitions and attributes; it also allows you to change existing definitions and attributes for easy and dynamic data base maintenance. Dictionary/3000 does not supply the data itself, which must come from MPE or KSAM files, IMAGE data bases, or the user.

Figure 2-2 shows how data definitions are compiled into a Transact intermediate processor code file through either the source program or the dictionary.

The compiler looks for any undefined data items in the dictionary. The compiler can resolve data item definitions and VPLUS form definitions through the dictionary. If it cannot find the items or forms in the dictionary, it issues a warning message and then produces the intermediate processor code file.

When the Transact processor executes this intermediate processor code, it, too, looks in the dictionary for undefined items. These items can be those not satisfied during compilation or items defined to be satisfied at run time by a DEFINE(ITEM) *item-name* statement. If the processor cannot find the items in the dictionary, it issues an error message and terminates processing.

The processor can resolve VPLUS form definitions only at compile time. At compile time, all item attributes can be resolved from their dictionary definitions. At run time, the processor *can* resolve such basic item attributes as type, size, decimal length, and storage length; it does not, however, get such secondary attributes as heading or entry text and edit nasks.

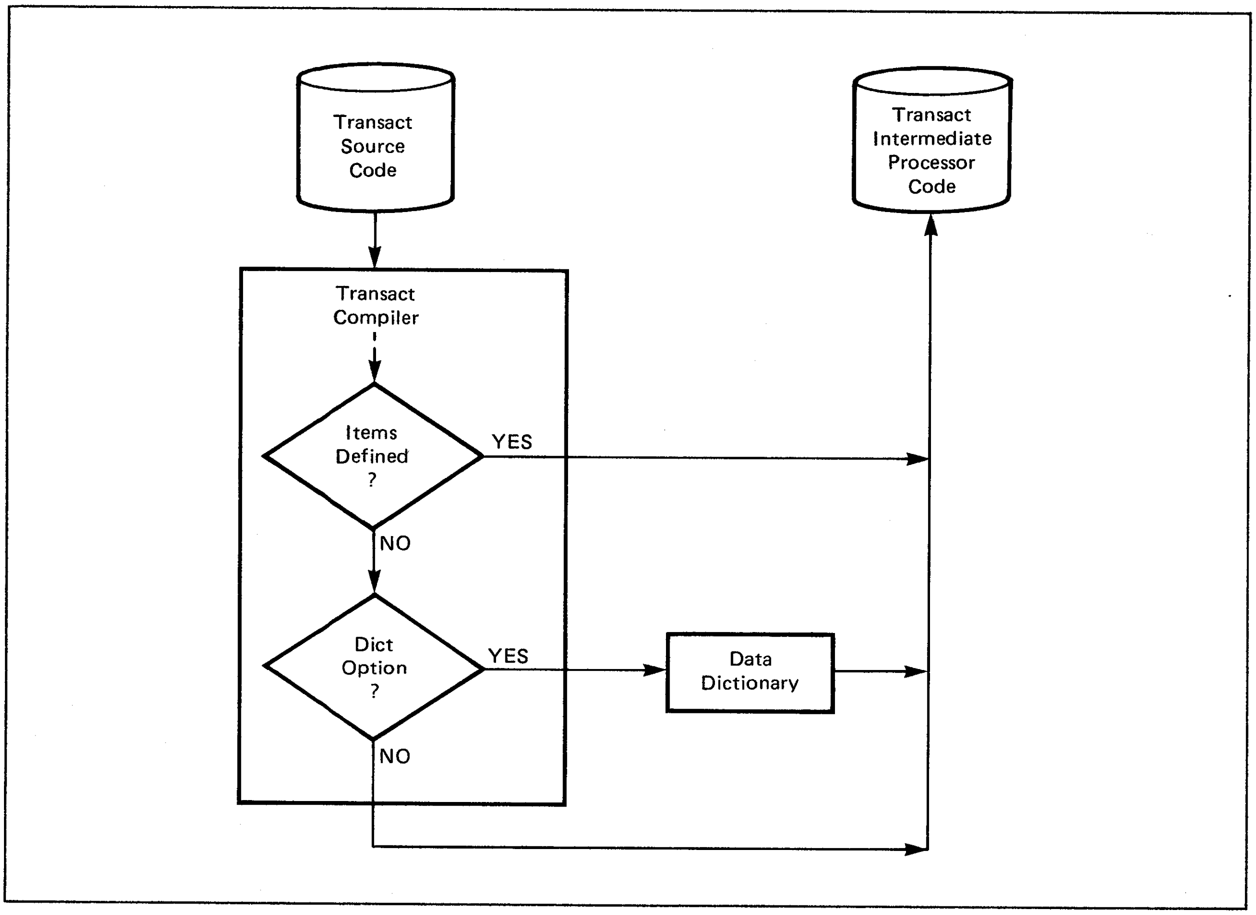


Figure 2-2. Data Definition

INTERFACE WITH OTHER SYSTEMS

As Figure 2-1 indicates, Transact interfaces with other systems and allows you to use various HP 3000 data storage facilities. For example, you can

- Enter and display data via prompts,
- Enter and display data on a terminal screen formatted by VPLUS/3000,
- Call system intrinsics and other compiled procedures that have been loaded into a segmented library file, and
- Use IMAGE, KSAM, and MPE files to store and access data. Transact verbs that enter, update, and delete data allow any access type needed by the file: chained, serial, direct, or indexed.

Data Management Interface

Transact provides data management facilities that allow you to use MPE files, KSAM files, and IMAGE data bases without making a single intrinsic call. The interface to these three subsystems is built upon a common set of verbs and a common set of special purpose registers.

The same verbs are used to manage the interface between Transact and the data storage subsystems, IMAGE, KSAM, and MPE. Modifiers associated with these verbs specify particular functions. For example FIND(CHAIN) retrieves all entries with a particular key value from either a KSAM file or an IMAGE detail data set. FIND(SERIAL) retrieves all entries in serial order from an MPE or KSAM file or any IMAGE data set.

The flexibility provided by the verb modifiers is further enhanced by the special registers provided with Transact (see Section 4). For instance, the key register contains the key for keyed selection; the match register contains criteria for selecting particular records or entries; the update register specifies not only the item to update but the new value; and the status register contains values used in error handling.

Error handling is automatic unless you choose to override it. With automatic error handling, the status register is set to the number of selected records for the file or data base being accessed, or to a subsystem error number if an error occurs. Further, when an error occurs, Transact returns the program to the state it was in before the data base transaction started.

Although you can fully define MPE or KSAM files and IMAGE data bases in a dictionary, you must also name each file or data base used by your program in the SYSTEM statement of the program. However, you need not name the data sets within a data base; Transact resolves the data set definitions from the data base root file. For a data base, you can also specify a password and/or an open mode in the SYSTEM statement; for a file, you may specify options similar to the file definition options in a FILE or BUILD command.

If desired, you can use the Transact PROC statement to call file system or IMAGE procedures directly, as well as your own SPL, PASCAL, COBOL, or FORTRAN procedures. When a PROC statement executes the called procedure, any open file or data base remains open and information is transferred across the call. One use for the PROC statement is to perform data set or data item locks on an IMAGE data base; Transact normally locks the entire data base whenever a modification is requested in a shared environment. Another use for the PROC statement is to delimit logical transactions through the IMAGE DBBEGIN and DBEND procedures.

VPLUS Interface

Transact uses a subset of the data management verbs (GET, PUT, SET, UPDATE) to access and control VPLUS forms. Without making calls directly to VPLUS intrinsics, you can retrieve data from forms, move data to forms, control forms sequence, manage function keys, and send messages to the window.

Often, many different functions are performed with a single statement. For instance, GET(FORM) opens the terminal and forms file, gets and displays a form, reads data entered by the user, performs any edits specified through FORMSPEC, highlights any field with errors and sends an error message to the window, transfers the data to the program, checks the data against the data definitions in the program, again performs error processing if necessary, and finally, performs any finish phase operations specified by FORMSPEC.

Normally, Transact programs operate with the terminal in character mode. Using any VPLUS verb (GET, PUT, SET, or UPDATE with the FORM modifier) places the terminal in block mode. Transact automatically switches back to character mode for any operation, such as a DISPLAY statement, that requires character mode.

The VPLUS interface supports function key labels on the 262x terminals when such labels are defined in FORMSPEC. It does does not support the split screen feature of 262X terminals, nor does it support Datacapture Terminals.

VPLUS forms files may be defined in a dictionary and/or in the SYSTEM statement of a Transact program. If not defined in the dictionary, each form and field must be specified in the SYSTEM statement; fields must be specified in screen order. Even if forms files are defined in the data dictionary, specifying individual forms in the SYSTEM statement reduces stack size.

Transact redefines fields in the forms according to its own data definitions, maintaining the correspondence between items in the program and fields in the forms through the screen order of the fields. Note that this means Transact does not retain the data independence provided by the VPLUS field numbers, nor does it retain field definitions specified in FORMSPEC.

If the automatic features of the VPLUS interface do not solve a particular application problem, you can call any of the VPLUS procedures directly with Transact's PROC verb. Any referenced forms file will remain open and the terminal will remain in block mode.

TRANSACT/3000 PROGRAMS

SECTION III

Programming in the Transact Programming Language requires an understanding of Transact program structure, which includes the following basic elements:

- The SYSTEM statement,
- The DEFINE(ITEM) statement,
- Command sequences,
- Statements,
- Comments,
- Delimiters, and
- Data Items.

This section includes a discussion of each of these elements.

Figure 3-1 illustrates some typical Transact code. It will be referenced throughout this section. The callouts on the illustration highlight important points.

```
  SYSTEM CSTINF,                                    <---------SYSTEM
     BASE = CUST(,3),                                       statement
     SIGNON = "CUSTOMER INFORMATION SYSTEM V1.0";

  DEFINE(ITEM) TDATE X(6); <<TODAY'S DATE>>  <----DEFINE(ITEM)
                                                            statement
  $$ADD:  <--command
  $$A:  <--short form
          subcommand                 comment
          /                          /
     $CUSTOMER:  <<ADD NEW CUSTOMER TO DATA BASE>>
     $C:
verb--> PROMPT CUST-NO ("ENTER CUSTOMER NUMBER"):     \
               CUST-NAME ("ENTER CUSTOMER NAME"):     |
               CUST-ADDR ("ENTER CUSTOMER ADDRESS"); |<--command
                  \                  \                |   sequence
                   data item         user prompt     |
verb--> PUT CUST-MAST;                                /
               \
                data set

     $PAYMENT:  <<ADD PAYMENT TO A/R DATA SET>>
     $P:
        PROMPT CUST-NO ("ENTER CUSTOMER NUMBER"),<----delimiter
                    CHECK=CUST-MAST:
               PDATE ("ENTER PAYMENT DATE"):<---------delimiter
               INV-NO ("ENTER INVOICE NUMBER"):
               AMOUNT ("ENTER AMOUNT OF PAYMENT");<---delimiter
        LIST TDATE, DATE;
        PUT AR-DETAIL; \
                       option
  $$UPDATE:
  $$U:
     $ADDRESS:  <<CHANGE CUSTOMER'S ADDRESS>>
     $A:
               verb modifier
                /
        PROMPT(KEY) CUST-NO ("ENTER CUSTOMER NUMBER");
        PROMPT CUST-ADDR ("ENTER CUSTOMER ADDRESS");
        UPDATE CUST-MAST, LIST=(CUST-ADDR);
                             \
  $$DELETE:                   option
  $$D:
     $CUSTOMER:  <<DELETE OLD CUSTOMER FROM DATA BASE>>
     $C:
        PROMPT(KEY) CUST-NO ("ENTER CUSTOMER NUMBER");
        DELETE CUST-MAST;

  END CSTINF;
```

Figure 3-1. Sample Transact Program

SYSTEM STATEMENT

The SYSTEM statement names the Transact program and any IMAGE data bases, MPE or KSAM files, or VPLUS forms that are used by the program. It can also override default space allocations used by the processor. The SYSTEM statement must be the first statement in any Transact program.

The SYSTEM statement in Figure 3-1 names a program, CSTINF, that maintains customer information and accounts receivable information kept in an IMAGE data base called CUST.

The format for the SYSTEM statement is specified in section 6.

DEFINE(ITEM) STATEMENT

DEFINE(ITEM) statements are used to define items that are not defined in your dictionary, or to redefine items that are defined in your dictionary. If you use a dictionary, items not defined in the dictionary may include temporary variables or any data items that you must explicitly redefine for your program. If you are not using a dictionary, then you must explicitly define every data item in your program in a DEFINE(ITEM) statement.

The DEFINE(ITEM) statement in Figure 3-1 defines a temporary variable used in the CSTINF program. The other data items used in Figure 3-1 are defined in the dictionary.

Although they may appear anywhere in a program, it is good practice to place any needed DEFINE(ITEM) statements immediately following the SYSTEM statement. DEFINE(ITEM) statements that follow the SYSTEM statement define data global to the Transact program. You can define data that is local to a program segment, however, by including DEFINE(ITEM) statements in that segment. Program segmentation is discussed in section 5.

Section 6 contains specifications for DEFINE(ITEM) statements.

COMMAND SEQUENCES

You may structure the body of a Transact program around command sequences specifically designed for a particular interactive interface to the program. A command sequence consists of the statements between a command or a subcommand and the next command, subcommand, or END statement, whichever comes first. One command sequence in Figure 3-1 begins with the statement following the subcommand $C and ends with the statement preceding $PAYMENT. The statements after $P and before $$UPDATE are also considered a command sequence. Command sequences divide the Transact program into functional parts that make logical sense to you and that are meaningful to the end user.

One or more functions in a Transact program can be contained in a command sequence. Each sequence is headed by a command label such as $$ADD or $$UPDATE and possibly one or more subcommand labels such as $CUSTOMER. Command and/or subcommand labels are followed by statements. Each sequence ends with another command label or an END statement.

The remainder of this section describes how the processor executes command sequences, and it discusses command and subcommand labels. It also tells how to request user-entered passwords for commands and subcommands.

NOTE: Although Transact is particularly well suited to writing programs for interactive access using command sequences, it is equally well suited to standard programming that does not make use of command sequences. In particular, programs that use VPLUS forms for data entry might not use the command sequence structure. Also, you may use Transact for standard batch processing with neither command sequences nor VPLUS forms.

Processing Command Sequences

When the Transact processor executes a program, it starts by executing any statements between the SYSTEM statement and the first command label of the root segment. If there is no command label in the root segment, the processor executes statements between the SYSTEM statement and the end of the root segment. When the processor encounters the first command label, it issues the prompting character ">" to ask the end user to enter a command. The end user must respond to this prompt with a command name defined in the program followed by a subcommand name, if there is one. The response that the end user gives determines which command sequence is executed. The end user may also respond with one of the processor-defined commands, such as the command EXIT to exit from Transact.

Before each command sequence is executed, the processor resets all the registers used for data storage and other data management functions, although it does not actually clear any data. Registers are discussed in section 4.

When the sample program in Figure 3-1 is executed, the end user might enter any of the following in response to the prompting character:

```
> ADD CUSTOMER

> ADD PAYMENT

> UPDATE ADDRESS

> DELETE CUSTOMER
```

Command and Subcommand Labels

A command label is preceded by "$$" and a subcommand label is preceded by "$". Both are followed by colons, as in

```
$$command:
   $subcommand:
```

Either label can contain from 1 to 16 characters, not including the leading "$$" or "$". It must begin with an alphanumeric character. The remaining characters may consist of any characters except $, ; : = < > ( ) " or a blank. All command and subcommand labels are global to the program and may be referenced from any program segment (see "Program Segmentation" in section 5.)

You may choose to use short forms for commands and subcommands. These short forms are illustrated for each command and subcommand in Figure 3-1.

A command label must have at least one character following the "$$", for example, "$$A:". A subcommand, however, can have a null value, as in "$:". The following code shows a null subcommand:

```
$$CHANGE:
  $ADDRESS:
  $:
```

The null subcommand in this example allows the statements following it to be executed whether the end user enters

```
> CHANGE ADDRESS
```

or merely

```
> CHANGE
```

User-entered Passwords for Commands and Subcommands

You may require that a system user enter a password in order to execute a command/subcommand sequence. A password can be a 1-8 character string of any combination of alphanumeric or special characters. Passwords must be specified exactly as they were defined. Thus, if a password was defined with all uppercase characters, then it must be specified with all uppercase characters in your program and entered by the end-user with all uppercase characters.

To request passwords, use the following syntax:

```
$$command("password"):
   $subcommand("password"):
```

Consider the following code:

```
$$ADD("PQX2"):
   $CUSTOMER:
```

When the end user enters the command

```
> ADD CUSTOMER
```

the processor requests the password:

```
COMMAND PASSWORD>
```

In order to execute the statements associated with the command ADD CUSTOMER, the end user must enter the correct password, PQX2:

```
COMMAND PASSWORD> PQX2
```

Note that the password is not "pqx2". Passwords must be exact.

Subcommands as well as commands can require passwords:

```
$$ADD("PQX2"):
   $CUSTOMER("MKC"):
```

When the end user enters the command:

```
> ADD CUSTOMER
```

the processor requests both passwords:

```
COMMAND PASSWORD> PQX2
SEQUENCE PASSWORD> MKC
```

If the end user enters an invalid command password, the processor responds with:

```
INVALID COMMAND PASSWORD.
```

If the end user enters an invalid password for a subcommand (or sequence), the processor responds with:

```
INVALID SEQUENCE PASSWORD.
```

In either case, the processor issues a prompt for another command.

STATEMENTS

Statements perform the data processing functions of a Transact program.

The general format for a Transact statement is:

[*label*:] *verb*[(*modifier*)] [*target*][,*option-list*];

These statement parts are described below, followed by a discussion of compound statements and statement formatting. Other statement parts, including relational and arithmetic operators, are listed with the verbs to which they apply. A statement is always terminated by a semicolon.

Labels

Statement labels help control program flow. They identify the point to which a conditional or unconditional statement should branch.

A statement label may be up to 32 characters long, and it must begin with an alphabetic character. It is followed by a colon and one or more Transact statements.

The following code illustrates three statement labels (START-FIND, TEST1, and PRINT) and points to a fourth (GRAND-TOTAL):

```
START-FIND:
   FIND(CHAIN) DET,
       LIST = (A:H),
       PERFORM = TEST1;
   PERFORM GRAND-TOTAL;
   END;

TEST1:
   IF (A)="AUGUST" THEN
      PERFORM PRINT;
   RETURN;

PRINT:
   LET (SUB) = (SUB) + (AMOUNT);
   DISPLAY;
   RETURN;
```

Verbs

Transact verbs are the heart of Transact statements. They are the action words for any procedure. Verbs in the above example include FIND, IF, LET, DISPLAY, PERFORM, RETURN, and END. Verbs in Figure 3-1 include PROMPT, PUT, LIST, UPDATE, and DELETE. Verbs are listed alphabetically and their specifications are described in detail in section 6.

Modifiers

Modifiers are an integral part of the verb that change or enhance its action. Some modifiers specify how values entered by the user will be used. Other modifiers describe a file access method.

In figure 3-1, the verb PROMPT(KEY) with the modifier KEY has a different function than PROMPT with no modifier. See section 6 for further information on the modifiers for each verb.

Modifiers are always enclosed within parentheses and must NOT be separated from the preceding verb by a space. For example:

```
FIND(CHAIN) DET;     <---correct

FIND (CHAIN) DET;    <---NOT correct
```

Target

The target identifies the program variable upon which the verb action is performed. It can also identify the file or data base for a file operation. Targets used in Figure 3-1 include names of data sets and data items, such as the data set CUST-MAST and the data item CUST-ADDR.

Option-List

A list of one or more options, separated by commas, can be specified with certain verbs to enhance their action. Some options tell how information should be formatted; others suppress regular processor operations. Examples of option-list options in Figure 3-1 are "DATE" and "LIST=(CUST-ADDR)".

The verbs that allow options also have a target; options always follow the verb's target separated by a comma. Some verbs allow you to specify more than one target/option-list combination by separating them with a colon, as follows:

```
verb(modifier)
     target1, option-list1:
     target2, option-list2:
      .
      .
     targetn, option-listn;
```

The verbs that allow such multiple target/option lists include DEFINE, DISPLAY, DATA, LIST, and PROMPT; multiple target/option lists are not allowed with data base or file access verbs.

Compound Statements

You may combine several Transact statements to form a compound statement, either unconditional or conditional. All compound statements are bracketed with a DO...DOEND pair of statements.

Compound statements may be nested. In that case, only the last DOEND has the ";" terminator. (Program delimiters are summarized in this section.)

The following is an example of an unconditional compound statement:

```
DO
   PROMPT(MATCH) CUST-NO;
   LIST NAME:
        ADDRESS:
        CITY:
        ZIP;
   OUTPUT MASTER, LIST=(CUST-NO:ZIP);
        LIST=(CUST-NO:ZIP);
DOEND;
```

The following is an example of a conditional compound statement:

```
IF (A) = (B) THEN
  DO
     LET (A) = (A) * (D);
     LET (B) = (B) * (X);
  DOEND
ELSE
  DO
     LET (A) = (A) * (C);
     LET (B) = (B) * (Z);
  DOEND;
```

Note that the first DOEND does not have the semicolon (;) terminator; the terminator is used to end the entire compound statement. Individual statements within the DO...DOEND pairs are, of course, terminated with a semicolon.

Statement Formatting

A Transact source program contains program text in 72-column records, not including line numbers. Program text is entered in free format. Good programming practice, however, suggests that you use a paragraph and indentation structure.

In general, you can read and modify code more easily if you break the code into separate lines for labels, verbs, and options, and use indentation freely.

You can break lines of code in any place except the middle of a word. Thus, the following two statements have the same effect:

```
MOVE (A)=(B);          and          MOVE (A)=
                                         (B);
```

Note that the second line can start anywhere and that no continuation indicator is required. Words, however, cannot be split. The following statements are illegal:

```
MO                     and          FIND
VE (A)=(B);                          (CHAIN)
```

(verbs cannot be split) *(verb(modifier) is a single word)*

A string within quotes can be split between lines:

```
DISPLAY "THIS IS A"
" TEST",
```

COMMENTS

Comments document a program but do not affect program execution; that is, a comment appears in the source code listing but does not generate any code. Comments may appear anywhere in the statement line and are enclosed with the "<<" and ">>" characters, as follows:

<<[*comment*]>>

The following is an example of comments used in Transact coding:

```
MOVENAME:
  MOVE (OUT-NAME) = (IN-NAME);   << Move input field to output field >>
```

Figure 3-1 also includes several comments.

DELIMITERS

Transact programs can contain five explicit delimiters, plus the space. The Transact delimiters are listed below with their functions.

Delimiter	Function
;	Semicolon - terminates a statement.
:	Colon - separates target/option phrases within statements; or serves as a terminator for a label, a command, or a subcommand; or specifies a range in LIST=options.
,	Comma - separates options within a statement.
=	Equal sign - when used in option-list, denotes the value an item should take; or serves as an arithmetic operator in MOVE and LET statements; or serves as a relational operator in condition clauses; or specifies the label to which a program should branch.
( )	Parentheses - enclose a modifier; or enclose an item name to reference its value; or enclose certain PROC statement parameters; other uses are noted in verb specification in section 6.
blank	Blank space - required as a delimiter between a verb or verb(modifier) and its target; must never appear between a verb and its modifier; otherwise, blanks are ignored.

Figure 3-1 contains examples of Transact delimiters.

DATA ITEMS

Data items defined in the dictionary can be obtained either at compile time by the Transact compiler or at execution time by the processor. They can also be defined in a Transact program by means of a DEFINE(ITEM) statement. The DEFINE(ITEM) statement must specify the name and type of the data. This section tells about data items and data item types; it also describes the concept of parent items and child items as well as the concept of compound items.

Data Item Names

The first character of a data item name must be alphanumeric. Subsequent characters may be either alphabetic (A through Z), digits (0 through 9), or any ASCII character that is not one of the following characters: , ; : = < > ( ) " or blank. The name can be from 1 to 16 characters long.

Data items in Figure 3-1 include CUST-NO, CUST-NAME, CUST-ADDR, and others.

When you are referring to the specific value of a data item that is in the data register, you must enclose the name in parentheses. (Registers are discussed in detail in section 4.) When you are referring to the name of the item, that is, its location in the list register, you do not enclose the name in parentheses.

Notice the difference between

```
LIST CUST-NO;
```

which reserves space for CUST-NO in the list register, and

```
LET (CUST-NO)= 123;
```

which manipulates the value of CUST-NO.

Data Item Types

Data items defined in a DEFINE(ITEM) statement or through Dictionary/3000 can be one of ten types. The following table lists the ten item types and their corresponding DEFINE(ITEM) code.

Item Type	DEFINE(ITEM) Code
Alphanumeric string	X
Uppercase alphanumeric string	U
Numeric ASCII string (leading zeroes stripped)	9
Zoned decimal (COBOL format)	Z
Packed decimal (COBOL comp-3)	P
Integer number	I
Integer number (COBOL comp)	J
Logical value (absolute binary)	K
Real, floating point, commercial notation	R
Real, floating point, scientific notation	E

You can specify that values must be positive only, by following the type with a "+". Positive only values never require an extra character to display the sign.

Data Item Sizes

Data item size is specified as the number of characters or digits you want in each data item. Transact determines how much storage space is required for that number of characters or digits based on the data item type. You may override the default storage space by specifying an exact storage size. You may also specify the number of decimal digits, that is, the number of digits you want to follow the decimal point in a numeric item. Ensure that space is allocated for the decimal point when you are computing item sizes.

When items are displayed, Transact generally requires the same number of display characters as the item size. If the item is not positive only, a character is added for the sign, even if the value to be displayed happens to be positive.

It is important to know exactly how Transact allocates storage for items used with IMAGE or VPLUS. For example, Transact does not require that data be stored as whole words, whereas IMAGE does; and Transact adds a display character to signed numeric items, whereas VPLUS does not.

Table 3-1 shows the storage allocated by Transact and the number of characters required for display, based on the data item type and size. It also gives the corresponding COBOL specification as an aid to understanding the Transact data types.

Table 3-1. Data Item Size

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
X or U	ASCII character string; 1 storage byte per specified character.	Same as storage.	DISPLAY PIC X
9	ASCII numeric string; 1 storage byte per specified digit.	Same as storage.	DISPLAY PIC 9
Z	*Zoned decimal number; 1 storage byte per digit, including sign, if any, which is combined with last digit: Z+(10) = 10 bytes Z(10) = 10 bytes	1 character per digit, plus 1 character for the sign, unless item is positive only: Z+(10) = 10 chars Z(10) = 11 chars	DISPLAY PIC 9 (Z+) DISPLAY PIC S9 (Z)

* Zoned decimal items are stored as a string of ASCII numeric digits. If the item is defined as Z, the rightmost digit is always overpunched with a sign indicator, a character that represents both the sign and the rightmost digit:

Low-Order Digit	Last Character if Positive	Last Character if Negative
0	{	}
1	A	J
2	B	K
.	.	.
.	.	.
.	.	.
9	I	R

If the item is defined as Z+ (implying positive only), no overpunch occurs and the rightmost digit is unchanged. Z+ is stored like type 9.

Table 3-1. Data Item Size (Continued)

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
P	Packed decimal digit; 1 nibble (1/2 byte) per digit, plus 1 nibble for the sign: P+(10) = 6 bytes P(10) = 6 bytes P(11) = 6 bytes (Sign is stored even if item is positive only)	1 character per digit, plus 1 character for a sign, unless item is positive only: P+(10) = 10 chars P(10) = 11 chars P(10,2) = 11 chars	COMP-3
I	Binary integer; storage length depends on item size:	1 character per digit, plus 1 character for a sign, unless item is positive only:	
	*I(1) to I(4) =2 bytes		COMP S9 to S9(4)
	*I(5) to I(9) =4 bytes	I+(5) = 5 chars I(5) = 6 chars I(5,2) = 6 chars	COMP S9(5) to S9(9)
	I(10) to I(18)=8 bytes		COMP S9(10) to S9(18)
	I(19) to I(27)=12 bytes		(none)
J	Identical to I; (use for consistency with IMAGE type J).		

* You can force 5-digit I-type values in the range 10,000 through 32,767 to be stored in 2 bytes by specifying storage length as 2: I(5,0,2). Similarly, you can force 10-digit values in the range 1,000,000,000 through 2,147,483,647 to be stored as 4 bytes: I(10,0,4).

Table 3-1. Data Item Size (Continued)

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
K	SPL logical value; storage length depends on item size: K(1) - K(4) = 2 bytes K(5) - K(9) = 4 bytes K(10) - K(18) = 8 bytes K(19) - K(27) =12 bytes	1 character per digit (K type items are always positive): K(10) = 10 chars K(10,2) = 10 chars	(none)
R	SPL Real or Long value: R(1) thru R(6) = 4 bytes R(7) and above = 8 bytes	1 character per digit, plus 1 character for a sign, unless item is positive only: R+(5) = 5 chars R(5) = 6 chars R(5,2) = 6 chars NOTE: Exponent is not displayed.	(none)
E	SPL Real or Long value; stored exactly like R. Constant values may not be entered in E format; constant values entered in other formats into E-type items are displayed in E-type format.	Displayed in format: *n.nnE+nn* 1 character per digit, plus 1 character each for the mantissa sign (unless item is positive), the decimal point, the E, and the exponent sign, plus 2 characters for the exponent: E(5) = 11 chars E+(5) = 10 chars E(5,2) = 10 chars	(none)

Data Types and VPLUS

Items are displayed on and entered from VPLUS forms in the external display format. (Refer to Table 3-1 for the display storage requirements.) It is important to remember that VPLUS does not add a character for the sign to its numeric data types, whereas Transact does. For example, if you want to display a 5-digit numeric item in a VPLUS field defined with a maximum size of 5 characters, you must define it in Transact as positive only. A VPLUS item with a size of 5 digits allows a maximum of 5 characters but a Transact item defined as I(5) requires 6 display characters.

Data Types and IMAGE

There are several differences between the IMAGE data types and Transact data types. The main difference is that IMAGE requires all items to be defined as whole words. In order to maintain consistency, you can define an item in Transact with an odd number of bytes, but specify that the item be stored in whole words. For example, you can define an item in Transact as 9(5,0,6) in order to specify 5 digits, stored as 6 bytes.

This example illustrates the second difference between IMAGE and Transact data types. IMAGE does not have a numeric ASCII type 9. This difference does not cause problems; Transact automatically converts a type 9 item to type X before transferring it to IMAGE. When data is transferred into a Transact type 9 item, Transact checks to make sure the data is numeric.

Data Types and Dictionary/3000

There is an exact correspondence between data item definitions in Transact and Dictionary/3000. Thus, when a Transact program uses an item defined in a dictionary, it is as if it were defined in the program's DEFINE(ITEM) statement. All item attributes can be resolved from the dictionary when Transact compiles with the default DICT option; and all item attributes, except for heading or entry text, edit masks, and sub-items, can be resolved from the dictionary at run-time.

You must use caution when using dictionary definitions of parent/child relations, compound/sub-item relations, and aliases. You must specifically define an alias relation in the DEFINE(ITEM) statement of your Transact program; any alias relations defined in the dictionary are ignored by Transact. The Transact processor recognizes parent/child and compound/sub-item relations defined in the dictionary, but you can only reserve space in the list register for the parent or compound item. (For details, refer to the DEFINE(ITEM) discussion in section 6, and to the discussions below of Parent and Child Items, Compound Items, and Alias Items.)

Parent Items and Child Items

A single data item can contain other data items, called child items. A data item containing child items is called a parent item. For example, a data item containing a date may be composed of three child items: month, day, and year, in any order you choose. A child item may itself be a parent item containing child items. In this case, it would be both a child item and a parent item.

You define the relation of a child to its parent by including, in the child item's definition, the parent item name and the position of the child item within the parent item. Child items need not be of the same type as parent items. A parent item need not be completely redefined by its child items. For example, a parent item that is 10 characters long may have a single child item that is 4 characters long starting in the 2nd character position of the parent item. (Refer to the DEFINE(ITEM) description in section 6 for details on how to define parent and child items.

Only the parent item name can be added to the list register, not the child item names. Child item names may, however, be used in a PROMPT or DATA statement to prompt the user for these values. Child items may also be specified in the LIST= options of statements that access VPLUS forms. The Transact processor understands that these item names are part of the parent item, and transfers data to the data registers accordingly. Transact makes the connection between parent and child items through the DEFINE(ITEM) or dictionary definition of their relation. This parent/child relation can only be resolved from the dictionary at compile time, not at run-time.

The child items can be the elements of a one- or multi-dimensional array, which is the parent item. The LET(OFFSET) verb modifier combination, described in section 6, helps manipulate such arrays.

Compound Items

Compound items are data items that are divided into smaller items called sub-items that are the same in all attributes, that is, size, type and number of decimal places. Compound items can be thought of as arrays. They, too, are defined in the DEFINE(ITEM) statement, if not already defined in the dictionary.

A sub-item is referenced by an offset into the compound item, not by an item name. Thus, only the compound item name can be added directly to the list register, or referenced in a LIST= option; since a sub-item has no name, it cannot be referenced by name.

Alias Items

Any item may be assigned an *alias-name* where the alias is another name for the defined item. Generally, you would use an alias in a Transact program where the dictionary definition of an item has the same definition but a different name in an IMAGE data set. The primary definition in the dictionary can be associated with one or more alias names to identify items in data sets that have different names. The primary name is always used in the Transact program.

You must define all alias relations with a DEFINE(ITEM) statement in your program; Transact ignores alias definitions in the dictionary.

TRANSACT DATA STORAGE REGISTERS

SECTION IV

Data storage registers, which are illustrated in Figure 4-1, are storage areas within the processor. The processor allocates space in these registers only when it is needed. You must ensure that space allocation is made in the appropriate registers and at appropriate points in the logic flow of the program. Because the order in which data is put into the registers is significant, you should always be aware of what is happening in the registers.

This section describes the data storage registers and includes a description of how they work.

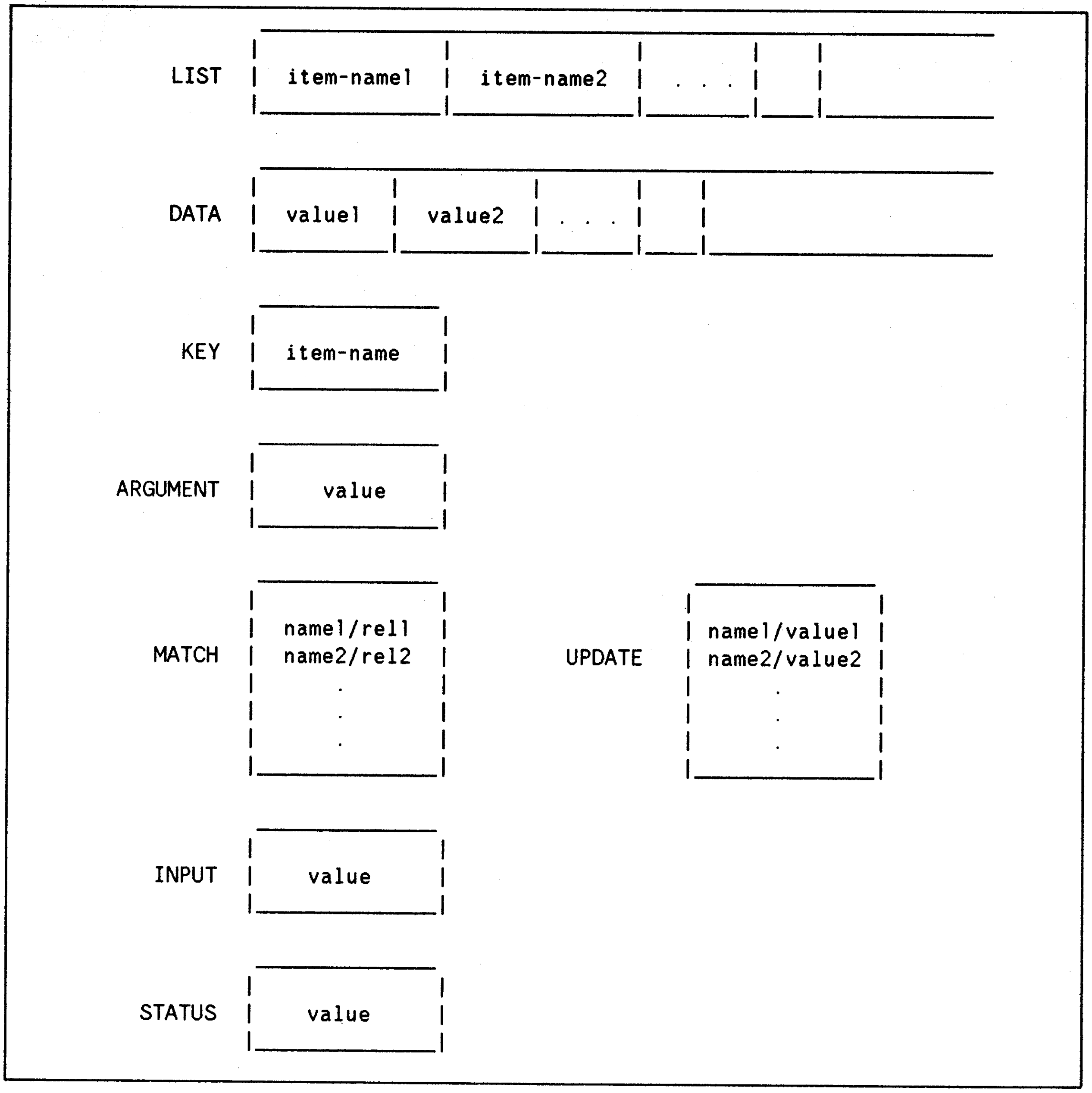


Figure 4-1. Data Storage Registers

LIST AND DATA REGISTERS

The list and data registers are the mechanism for data storage internal to the processor. If you are familiar with the concept of "stacks" and how they work, it may help to know that the list and data registers are manipulated as stacks.

The data register is the storage area for the values of data items. Any data item you wish to manipulate must have its value in the data register. The data register holds values (data) only; it does not hold any information about the values, such as what value is associated with what data item.

The list register is a map of the data register. It holds the names of data items. For every data item name in the list register, there is a fixed amount of space (determined by the data item's attributes) allocated for that data item in the data register. This storage space in the data register holds the value for the data item. The list register itself contains only the data item names, not their values.

The order of item names in the list register determines the order of the corresponding data item values in the data register. Item names in the list register, and the corresponding space in the data register, are allocated in the order they are specified in the program. For example, if the first data item name added to the list register is NAME, identifying a six-byte character string, then the first six bytes of the data register are allocated to hold the value of NAME. If NAME is followed in the list register by an item called ADDRESS that identifies a 20-byte character string, the value of ADDRESS requires 20 bytes of storage space in the data register following the value of NAME.

```
          _____________________________________________
         |    |       |
List     |NAME|ADDRESS|
Register |____|_______________________________________
         \     \
          _____________________________________________
         |      |                     |
Data     |Miller|15 West Cliff Drive  |
Register |______|_____________________|________________

         <---bottom                              top---->
```

The data item names are added to the list register starting at one end, the bottom, filling towards the other end, the top. The most recently added data item name is at the top of the currently used space in the list register, and its value is at the top of the currently allocated space in the data register. Remember that you can think of these registers as stacks.

Managing the List and Data Registers

At the start of program execution, the list register is empty and the contents of the data register are undefined. When the list register is empty, you cannot access the data register. During the course of program execution, you add data item names to the list register, thereby defining the data item space. Every item added to the list register must have been previously defined either in a DEFINE(ITEM) statement or the dictionary. Note that child item names may not be added to the list register, only the parent item names.

Allocating space in the data register does not move the data into the register. Transact provides means to transfer data to the data register either interactively from a terminal through prompts or a VPLUS screen, or programmatically from files or data sets.

In order to keep your data storage requirements to a minimum, you should release the data register space for your data items when you are through using them. One way to release this storage space is to let Transact do it for you. If you are using a command structure Transact resets the list register whenever a command sequence executes. If you are not using a command structure, you should manage your data storage directly with Transact statements.

When data items are removed from the list register, they are removed from top to bottom; that is the last item added is the first item removed. The values, however, corresponding to items removed from the list register still exist in the data register. You can access these values again by adding their item names back into the list register in the correct sequence.

KEY AND ARGUMENT REGISTERS

The Transact processor uses the key and argument registers to perform keyed access to KSAM files or IMAGE data sets. You must use these registers in order to perform keyed access to such files; you do not need these registers, however, to access MPE files or for serial access to IMAGE data sets or KSAM files.

Both registers are write-only registers. That is, you can assign a data item name to the key register and a value to the argument register, but you cannot read either register, nor can you test their contents. The processor uses the contents of these registers for file and data set access; and a program can pass their values to an external procedure.

A unique pair of key and argument registers is made available with each level of nesting of the PERFORM= option of the data management verbs. As many as 10 levels can be declared.

Key Register

The key register contains a single data item name that identifies a key item in a KSAM file or a search item in an IMAGE data set. The name you place in the key register is used by the processor to perform a keyed access to an existing record. The key register is not used to add a new record or entry.

The key register is needed only when the key name must be specified. It is needed to locate a particular key in a KSAM file, and it is needed to locate the chain head in an IMAGE detail data set. The key register is not needed to access key items in IMAGE manual or automatic master sets; there is only one key (search) item in a master data set and that item is known to IMAGE.

Argument Register

The argument register contains a single value, the value of the key item in the key register. The Transact processor uses this value to locate any records in a KSAM file or an IMAGE data set with that key value. If you try to perform a keyed access without setting up the key and argument registers, Transact issues an error message.

The argument register is needed when an actual key value is used to access a file or data set. If the key is known (as in an IMAGE master set), you need not set up the key register, but you must still set up the argument register, unless you want to access all the entries.

To illustrate, suppose you have an IMAGE detail data set from which you want to retrieve all product numbers with the value A105. You can put the search item name (PROD-NO) in the key register and the value (A105) in the argument register.

```
            __________
key        |          |
register   | PROD-NO  |
           |__________|

            __________
argument   |          |
register   | A105     |
           |__________|
```

You can then use an appropriate Transact statement to retrieve any entries that contain a product number with the value "A105". Transact performs all the necessary IMAGE calls.

MATCH REGISTER

The match register contains the selection criteria for data retrieval operations. It holds a list of data item names and selection criteria for each data item name in the list. The match criteria determine which records are selected when a retrieval is performed from a data set or file. Only those records that meet the criteria are retrieved.

In order to use the match criteria, the match items must be in the list register and also must be retrieved by the data management statement that uses the match criteria. You must, therefore, not only add match items to the match register, but also add each item to the list register and include each item in a LIST= option of the data management statement. If a match item is not specified in the LIST= option, the data management statement ignores the match criteria associated with that item.

As many match criteria as you want can be specified. You may assign different criteria to the same item or to different items, or specify the same criteria for different items. By default, a Boolean AND connects selection criteria gathered from different PROMPT(MATCH) or DATA(MATCH) statements. A Boolean AND also connects selection criteria from multiple SET(MATCH) statements unless the statements use the same item name and specify equality as the connector; such statements are joined by a Boolean OR. End users can specifically override these defaults by their responses to a PROMPT(MATCH) or a DATA(MATCH) prompt (see "Responding to a Match Prompt" in section 5).

For example, consider the following match register which contains four separate match criteria:

Match Register

```
 ____________________________________________________________________
|                                                                    |
|  UNIT-PR         QTY-ORDERED           QTY-ORDERED          CREDIT |
| less than  AND  greater than   AND    less than      AND    equals |
|   500              10000               1000000                A    |
|____________________________________________________________________|
```

UPDATE REGISTER

The update register holds a list of update specifications, each consisting of a data item name and a new value for that item. These name/value pairs may be used to update records in an MPE or KSAM file or an IMAGE data set. The update register is used with the REPLACE verb to update one or more records.

The update register operates on data retrieved with data management verbs. The retrieved data generally satisfies other criteria set up in the key register or in the match register. The update register contains new values for data items in the selected entries. When REPLACE executes, it retrieves each selected entry and places its current values in the data register. It then replaces any values in the data register that have a corresponding value in the update register. If a data item is not named in the update register, its value in the data register is not changed. REPLACE then writes the updated entry back to the file or data set.

For example, suppose you want to change the credit rating for all customers whose rating is currently "A" to "A1". You can set up the match register to contain the criterion CREDIT = "A" and then set up the update register with the new value for CREDIT.

```
Match Register:              Update Register:

 ______________               ______________
|              |             |              |
|   CREDIT     |             |   CREDIT     |
|     =        |             |    "A1"      |
|    "A"       |             |              |
|______________|             |______________|
```

NOTE: You can update records without using the update register simply by changing the values in the data register. You do not use the update register with the UPDATE verb, and you normally would not use it with REPLACE to update multiple entries with different values. The update register is particularly useful for making the same change to multiple entries.

INPUT REGISTER

The input register contains a character string entered by an end user in response to a prompt generated by the INPUT verb. Typically, the contents of the input register are tested with an IF verb for a yes or no condition. Because the processor upshifts all responses, it is not necessary to test for "YES" and "yes", for example. The contents of the input register cannot be assigned to any data item or any other register.

STATUS REGISTER

The status register is used to hold status information about the last operation performed. The contents of the status register differ depending on whether Transact uses the register for its automatic error handling, or you control error handling programmatically by specifying the STATUS option with various verbs. (Refer to "Automatic Error Handling" in section 5 for a full discussion of the status register contents.)

In either case, you can test the contents of the status register with an IF statement; you can also assign the contents of the status register to a variable for subsequent display or testing.

Transact's automatic error handling is a powerful feature of the the processor and, in general, you should allow the status register to be used for this purpose. If you do choose to override automatic error handling with the STATUS option, you are responsible for doing all your own error handling. You should also be aware that the STATUS option affects the operation of the data management verbs. In general, using this option makes iterative verbs singular and requires the use of the PATH verb to specify a path for subsequent keyed retrievals. It also suppresses the rewind operation on a data set prior to a serial read.

Refer to section 5 for a full discussion of the status register and how it operates with both automatic and programmer-controlled error handling.

HOW REGISTERS WORK

This section summarizes the use of registers with Transact verbs. It also illustrates how registers work using code extracted from a Transact program.

Verbs and Registers

The LIST, DATA, PROMPT, and INPUT verbs cause data to be placed into the various registers. Following, is an overview of how these four verbs work with the registers.

- LIST causes an item to be placed in the list register and appropriate space to be allocated in the data register.
  - For use with VPLUS, items may be in any position in the register.
  - For use with IMAGE data base access, items must be consecutive but in any order.
  - For use with KSAM or MPE files, items must be consecutive and in the same order as in the records.
- DATA places values in the data register in space already allocated. These values come from user input, because DATA causes a prompt.
- PROMPT places the item name in the list register and places the value (supplied by the user) in the data register.
- INPUT places a character string (supplied by the user) into the input register.

How LIST, DATA, and PROMPT act is specified by their modifier. Table 4-1 shows how verbs and modifiers work together to affect registers.

In addition to the verbs listed above, all the data base and file access verbs (except PUT) and the assignment verbs LET, SET, and MOVE add values to the data register. The data access verbs get the data from files or data bases; with LET, SET, and MOVE, the data is assigned in the program.

Table 4-1. Verb/Modifier/Register Summary

Verb / Modifier	PROMPT	LIST	DATA	INPUT
none	List Data	List	Data	Input (1)
PATH	List Data Key Argument	List Key	Data Argument	-
KEY	Key Argument	Key	Argument Key (2)	-
MATCH	List Data Match	List Match	Data Match	-
UPDATE	List Data Update	List Update	Data Update	-
SET	List (1) Data (1)	-	Data (1)	-
ITEM	-	-	Data (3)	-

(1) Only if the user enters a value
(2) If key register is empty
(3) For the given item

For example, PROMPT affects the list and data registers only, whereas PROMPT(PATH) affects the list, data, key, and argument registers. If you only want to add items to the list register, use LIST with no modifier; if you only want to add an item to the key register, use LIST(KEY).

Sample of Transact Coding

The following code extracted from a Transact program shows how registers work. This discussion includes

- Illustrations showing the files used and the records from those files, and
- The Transact code and the corresponding register activity.

The code is shown in total first and then broken down by statement.

```
LIST CUST-NAME:
     CUST-ADDRESS;
PROMPT(PATH) CUST-NO;
GET CUSTOMERS,
     LIST=(CUST-NAME:CUST-ADDRESS);
PROMPT(PATH) PART-NO;
PROMPT QTY-ORDERED;
LIST COST;
LIST UNIT-PRICE:
     PART-DESC:
     QTY-ONHAND;
GET PARTS,
     LIST=(UNIT-PRICE:QTY-ONHAND);
IF (QTY-ORDERED) > (QTY-ONHAND) THEN
   DISPLAY "Only": QTY-ONHAND, NOHEAD: "in stock"
   ELSE
     DO
       LET (QTY-ONHAND)=(QTY-ONHAND) - (QTY-ORDERED);
       UPDATE PARTS, LIST=(QTY-ONHAND);
       LET (COST) = (UNIT-PRICE) * (QTY-ORDERED);
       PUT ORDERS, LIST=(CUST-NO:COST);
     DOEND;
```

The data base referenced contains the three data sets shown below (PARTS, CUSTOMERS, and ORDERS). The items in each data set are also listed below.

```
PARTS
  M
 ____      PART-NO, UNIT-PRICE, PART-DESC, QTY-ONHAND
 \  /
  \/


CUSTOMERS
  M
 ____      CUST-NO, CUST-NAME, CUST-ADDRESS
 \  /
  \/


ORDERS
  D
_____      PART-NO, QTY-ORDERED, COST, CUST-NO
\   /
 \_/
```

The following illustrations show how specific statements affect specific registers.

```
LIST CUST-NAME:
     CUST-ADDRESS;
```

CUST-NAME and CUST-ADDRESS are placed in the list register, and space is reserved for their values in the data register.

LIST	CUST-NAME
DATA	
KEY	
ARG	

LIST	CUST-NAME	CUST-ADDRESS
DATA		
KEY		
ARG		

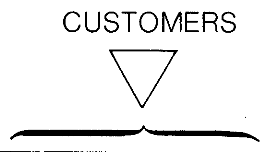
```
PROMPT(PATH) CUST-NO;
```

Transact prompts the user for CUST-NO, and places the item name CUST-NO in the list and key registers. It places the user's response in the data and argument registers.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO
DATA			345
KEY	CUST-NO		
ARG	345		

```
GET CUSTOMERS,
    LIST=(CUST-NAME:CUST-ADDRESS);
```

When Transact retrieves the appropriate record from the CUSTOMERS data set using the key and argument values, it places the values for CUST-NAME and CUST-ADDRESS into the data register.



LIST	CUST-NAME	CUST-ADDRESS	CUST-NO
DATA	ABC CO	13 CANAL ST.	345
KEY	CUST-NO		
ARG	345		

```
PROMPT(PATH) PART-NO;
```

Transact prompts the user for PART-NO, places the item name PART-NO into the list and key registers, overwriting any value already in the key register. It then places the value entered by the user into the data and argument registers, overwriting the previous values in those registers.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO
DATA	ABC CO	13 CANAL ST.	345	1234
KEY	PART-NO			
ARG	1234			

```
PROMPT QTY-ORDERED;
```

Transact prompts the user for QTY-ORDERED, and places the item name QTY-ORDERED in the list register. It then places the value entered by the user into the data register.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED
DATA	ABC CO	13 CANAL ST.	345	1234	3
KEY	PART-NO				
ARG	1234				

```
LIST COST;
```

Transact places COST in the list register, and reserves space for its value in the data register.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST
DATA	ABC CO	13 CANAL ST.	345	1234	3	
KEY	PART-NO					
ARG	1234					

```
LIST UNIT-PRICE:
     PART-DESC:
     QTY-ONHAND;
```

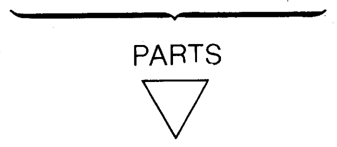
UNIT-PRICE, PART-DESC, and QTY-ONHAND are placed in the list register, and space is reserved for their values in the data register.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST	UNIT-PRICE	PART-DESC	QTY-ONHAND
DATA	ABC CO	13 CANAL ST.	345	1234	3				
KEY	PART-NO								
ARG	1234								

```
GET PARTS,
     LIST=(UNIT-PRICE:QTY-ONHAND);
```

When the appropriate record is retrieved from the PARTS data set using the key and argument values, values are placed in the data register for UNIT-PRICE, PART-DESC, and QTY-ONHAND. Note that PART-DESC need not be specified here, because it is in the range between UNIT-PRICE and QTY-ONHAND.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST	UNIT-PRICE	PART-DESC	QTY-ONHAND
DATA	ABC CO	13 CANAL ST.	345	1234	3		998	FRAMMIS	5
KEY	PART-NO								
ARG	1234								



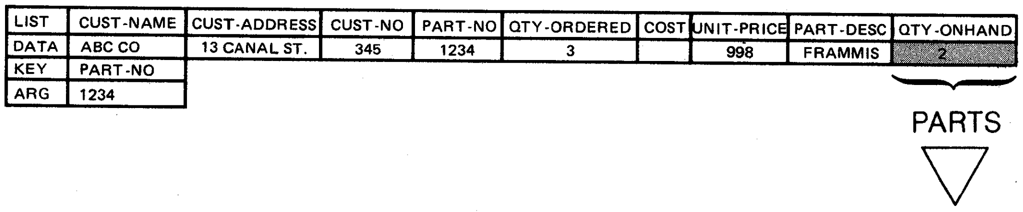
```
IF (QTY-ORDERED) > (QTY-ONHAND) THEN
   DISPLAY "Only": QTY-ONHAND, NOHEAD: "in stock";
   ELSE
     DO
       LET (QTY-ONHAND) = (QTY-ONHAND) - (QTY-ORDERED);
```

This statement computes a new QTY-ONHAND value and places it in the data register.

LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST	UNIT-PRICE	PART-DESC	QTY-ONHAND
DATA	ABC CO	13 CANAL ST.	345	1234	3		998	FRAMMIS	2
KEY	PART-NO								
ARG	1234								

```
UPDATE PARTS, LIST=(QTY-ONHAND);
```

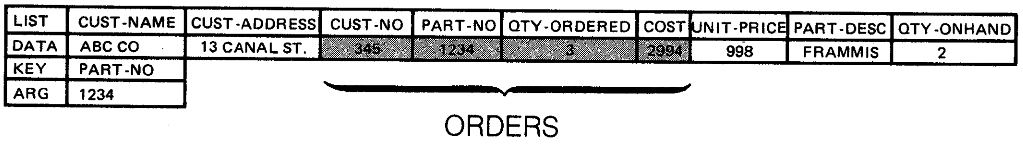
Update the PARTS data set with the new QTY-ONHAND for the part whose entry was the last one accessed by the previous GET statement.



LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST	UNIT-PRICE	PART-DESC	QTY-ONHAND
DATA	ABC CO	13 CANAL ST.	345	1234	3		998	FRAMMIS	2
KEY	PART-NO								
ARG	1234								

```
  LET (COST) = (UNIT-PRICE) * (QTY-ORDERED);
  PUT ORDERS, LIST=(CUST-NO:COST);
DOEND;
```

Compute the cost and place it in the data register. Update the ORDERS data set with the values from CUST-NO through COST.



LIST	CUST-NAME	CUST-ADDRESS	CUST-NO	PART-NO	QTY-ORDERED	COST	UNIT-PRICE	PART-DESC	QTY-ONHAND
DATA	ABC CO	13 CANAL ST.	345	1234	3	2994	998	FRAMMIS	2
KEY	PART-NO								
ARG	1234								

RUNNING TRANSACT/3000

SECTION V

This section explains how to run Transact, including

- How to compile and execute Transact programs,
- How to control Transact programs at run time,
- How automatic error handling works, and
- How to control processing using the STATUS option.

TRANSACT PROGRAM COMPILATION

This section tells how to compile Transact programs and lists the control options you can choose. It also illustrates a compiler listing, tells how you can control compiler listings, discusses program segmentation, and describes how to control input sources to and output destinations from the compiler.

Compiling Transact Programs

You create Transact source programs using the HP EDITOR subsystem or any suitable text management system. The source code file can be either numbered or unnumbered. You request the Transact compiler to compile the source code with the following command:

```
:RUN TRANCOMP.PUB.SYS
```

When you are running interactively and responding to prompts at a terminal, the compiler prompts for the name of the file containing the Transact source code:

SOURCE FILE>	Enter the file name under which you saved the source code.
LIST FILE>	Enter a carriage return to direct the listing to your terminal ($STDLIST). You can direct the listing to the line printer by responding with LP; or you can suppress the listing altogether by responding with NULL. These are the more common responses. See the discussion of "Controlling Output Destinations from the Compiler" for other possible responses.

The compiler then asks you to specify which control options are to be applied to the compilation:

CONTROL> Enter one or more of the following options, separated by commas in response to this prompt. You can precede any of the options by "NO" to reverse its effect.

The default options are marked by an asterisk (*).

*LIST	Generates a listing of the compiled source code.
*DICT	References the data dictionary to resolve item definitions.
*CODE	Creates the intermediate processor code file that can be executed by the Transact processor. The code file is created only if no errors occur during compilation (See option XERR).
*ERRS	Lists compilation errors on $STDLIST, even if you direct a listing elsewhere.

DEFN	Produces a listing of item definitions as part of the compiler list output. You can use this option to determine how much storage space the Transact compiler has allocated to each data item defined in your source code or in the dictionary.
OBJT	Produces a listing of the intermediate processor (object) code.
OPTI	Optimizes the tables in the code file so that the data segment stack is reduced at execution time. This option is meaningful only if any data items are defined with the OPT option of DEFINE(ITEM) to suppress the item's textual name. Note that the OPTI option should not be used if the data item names are needed for prompt strings, display item headings, and LIST= constructs. Appendix E provides additional information on this option in conjunction with data stack optimization.
OPTS	Optimizes segmented Transact programs only. When you include this option, the processor does not check for local segment items in the list, match, and update registers when loading a new segment. Since such checks are essential for debugging programs under development, this option should only be used after a program is fully tested and ready for production. Although OPTS speeds segment transfers, the program may malfunction or terminate abnormally if a local item is left in a register.
STAT	Generates statistics on data stack usage. These values are useful in deciding how program structural and/or coding differences would improve the run-time performance of your program. Appendix E provides additional information on this option in conjunction with data stack optimization.
XERR	Creates a code file even if errors are encountered in the compilation (See option CODE).
XREF	Generates a listing to provide a cross-reference to locations of label definitions and their references.

Two RUN command options may be used to bypass the Transact compiler prompts. These are the PARM= and INFO= options, which are specified in the compiler invocation statement.

The PARM= option has parameters that identify your source file and/or your list file:

1	This parameter indicates that TRANTEXT is the formal-file-designator for your source file. If it is specified, the SOURCE FILE> prompt does not appear.
2	This parameter indicates that TRANLIST is the formal-file-designator for your list file. If it is specified, the LIST FILE> prompt does not appear.
3	This parameter indicates that both TRANTEXT and TRANLIST are formal-file-designators. If used, neither the SOURCE FILE> nor the LIST FILE> prompt appears.

The following invocation produces two listings at the line printer after the source statements in APPL01 are processed:

```
FILE TRANTEXT=APPL01
FILE TRANLIST;DEV=LP,,2
RUN TRANCOMP PUB.SYS; PARM=3; INFO="DEFN, XREF"
```

The INFO= option accepts parameters identical with the options used to respond to the CONTROL> prompt. As the above example illustrates, the parameter is enclosed in quotation marks. If only blanks are included in the quotation marks, the default compiler options take effect. If the INFO= option is used, the CONTROL> prompt does not appear.

You can direct the compiler to a file for answers to its prompts. See "Controlling Input Sources to the Compiler." You can also compile a program by streaming it as a batch job. To do this, set up the stream file to contain the following MPE commands:

```
:STREAM
:!JOB jobname
:!RUN TRANCOMP.PUB.SYS
:filename
:list-destination
:control-options
:!EOJ
```

Compiler Listing

Figure 5-1 shows the listing of the source program produced by the compiler using all four default control options. The three columns of figures on the left hand side of the page are annotated in the figure, and described below it.

Figure 5-2 illustrates the output from a compilation that used the STATistics option. The statistics are especially useful for helping you optimize your application's run-time data stack utilization. Appendix E defines the statistics fields in detail, maps them to actual stack components, and suggests ways to use test modes, program structure, and various coding options to minimize stack space.

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

         Line Number
        /      Internal Location
       /      /    Nesting Level
      /      /    /
    1.000   /    / SYSTEM COMPIL;
    2.000  0000  /  IF (A) = (B)
    3.000  0000 1       THEN DO
    4.000  0000 1       DISPLAY "DUPLICATE ENTRY";
    5.000  0005 1       IF (A) = (C)
    6.000  0005 2           THEN IF (D) < 50
    7.000  0008 2               THEN MOVE (A) = (D);
    8.000  0013 1       DOEND;
    9.000  0013     END;


CODE FILE STATUS: NEW


0 COMPILATION ERRORS
PROCESSOR TIME=00:00:01
ELAPSED TIME=00:00:03
```

Figure 5-1. Compiler Listing

Line Number	Line number from the source listing
Internal Location	Internal location reference number of the statement on the associated text line. These numbers are useful when TEST mode is used during execution (See Section 7).
Nesting Level	Nesting level indicator that is incremented by one when the compiler encounters the start of a compound statement or a new level and decremented by one when the end of such a compound statement or level is reached.

```
COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

*****COMPILE TIME STATISTICS****   __
     STACK=      23368               |--> words on stack used during
     TABLE=      14482             __|    compilation

*******RUN TIME STATISTICS******   __
     PCODE=          0               |--> words on stack for instruction
     SCODE=       3764             __|    code data

***PARTIAL TABLE REG. SUMMARY***   __
      BASE=    1,   10               |
      FILE=   38,  544               |
       SET=   12,  176               |
      PROC=    0,    0               |
     $$CMD=   11,   65               |
      $CMD=    0,    0               |
      ITEM=   82, 1047               |
     STRNG=  195, 2192               |--> words on stack for managing such
     CNTRL=  116,  916               |    entities as data bases, files,
          ------------               |    strings, procedures, commands, and
             455, 4950               |    Transact register manipulations
                                     |
****FINAL TABLE REG. SUMMARY****     |
WORK  AREA=   30,  100               |
          ------------               |
TABLE REG.=       5050               |
TABLE INDX=        485               |
TABLE LEN.=        485             __|

*****RUN TIME STACK SUMMARY*****   __
DATA  REG.=        200               |
TABLE REG.=       5050               |
TABLE INDX=        485               |
TABLE LEN.=        485               |
ROOT  SEG.=       3765               |--> summary of Transact processor
ITEM  REG.=         30               |    stack use
DATA INDEX=         30               |
DATA  LEN.=         30               |
          ------------               |
                 10075             __|
CODE FILE STATUS: REPLACED

0 COMPILATION ERRORS
PROCESSOR TIME=00:01:43
ELAPSED TIME=00:02:15
```

Figure 5-2. Compiler Statistics

Controlling the Compiler Output

You may place any of the following commands between any two statements in the source program to control the compiled output:

!COPYRIGHT ("*text-string*")	Causes the compiler to place the specified *"text-string"* in the first record of the code file as a copyright notice. The text string may be up to 500 characters long. This command may only be specified once; normally, it should follow the SYSTEM statement.
!INCLUDE(*file-name*)	Causes the compiler to include the Transact statements from a specified source file (*file-name*) that is not the source file being compiled. The *file-name* statements are included at the point in the listing where !INCLUDE appears, and are compiled with the main source file. *file-name* may be a fully qualified name with file group and account. Up to 5 files may be nested with !INCLUDE commands.
!LIST	Write subsequent source statements to the list file. If LIST is specified in response to the CONTROL> prompt, !LIST has no effect.
!NOLIST	Suppress the listing of subsequent source statements. If NOLIST is specified in response to the CONTROL> prompt, !NOLIST has no effect.
!PAGE	Causes the compiler to skip to the top of the next page on the listing
!SEGMENT [("*text-string*")]	Causes the compiler to segment the program and the resulting code file at this point in the source file. The compiler displays the specified "*text-string*" on TRANOUT when it processes the !SEGMENT command. The text string may be up to 500 characters long. The following discussion of segmentation tells why and how to segment programs.

Since these commands are not language statements, do not terminate them with a semicolon.

Program Segmentation

The Transact/3000 compiler produces compact code. This code is placed on the process stack at execution time and therefore affects the size of the stack. Even though the Transact code is compact, large programs may produce so much executable code that the process stack becomes uncomfortably large for the operating environment. Some programs produce a code file so large that the process stack cannot contain the code.

One way to solve this problem is to segment your program. Transact allows you to segment your program into as many as 126 separate segments.

If you choose to divide your program into segments, these segments can be overlaid in the processor stack in memory. In addition to the root segment (segment 0) which is always in memory, only the currently executing segment needs to be on the memory stack. When control transfers to another segment, the new segment can overlay the segment currently in memory. This technique allows the processor to execute within a smaller stack size than that needed by an entire program.

You divide a program into segments by including the !SEGMENT compiler command in your code wherever you want a new segment to start. You can place this command between any two Transact statements. However, you should exercise judgement about where you segment your program. For example, you should not segment within a loop construct. And, when a FIND or OUTPUT statement, for example, requires a PERFORM block, the statement and the PERFORM block should be within the same segment. Program control cannot automatically cross segment boundaries.

One way to control the use of segments is with command labels. When an end user enters a command, control transfers to the associated command label. As far as the end user is concerned, it does not matter in which segment a command label is coded; when the user specifies a particular command label identifying a particular sequence, the Transact processor makes sure the segment containing that sequence is loaded into memory, if it is not there already.

Another way to control the use of segments is to use a GO TO or PERFORM statement to transfer control to a program control label in a different segment. In order to transfer control to a program label in another segment, you must specifically define that label as an entry point. Entry point labels are necessary for transfers into any segment except your main program segment (segment 0, the "root" segment). You define a label as an entry point with a DEFINE(ENTRY) statement. Labels so defined are global to your program; that is, they can be referenced from outside the segment in which they appear. Labels defined within a segment are local to that segment.

The following information describes exactly how segmentation affects data items and command or program labels.

- All command and subcommand labels are global to the program in which they are declared. That is, you can reference them from any segment, and they must be unique within the entire program.
- All program control labels and data items declared before the first !SEGMENT command are global to the program and may be referenced from any point.
- Any program control label or data item declared after a !SEGMENT command is local to that segment. An item of the same name may be declared in another segment and its separate definition is insured.

If you use the compile option DEFN in a segmented program, the compiler produces a list of the effective ITEM definitions at the end of each segment.

Normally, Transact checks the list, match, and update registers when it loads a new segment to make sure they do not contain items local to another segment. However, if you compile your program with the compile option OPTS, Transact does not check the registers for local items. If items local to one segment remain in these registers when another segment is executed, they may cause your program to malfunction or even abort.

In addition to the specific considerations discussed above, you should always consider the following general rules when segmenting your programs:

- Stay in one segment for as long as possible; and, when you leave a segment, stay out for as long as possible;
- Try to define segments of uniform size since stack space is allocated for the largest segment
- Put routines used by many segments in the main (root) segment since it always resides in memory along with whatever other segments happen to be loaded. However, try to minimize the size of this segment as well.

Controlling Input Sources to the Compiler

TRANIN is the formal-file-designator for responses to prompts issued by the compiler. The default setting for TRANIN is $STDINX. You may, however, change that by means of a file equation. A file equation is specified with the MPE FILE command, which is further described in the MPE COMMANDS reference manual. The compiler then reads input from that file until it encounters an end-of-file condition. If it reaches end-of-file before all prompts are answered, it returns to $STDINX. (If TRANIN is an EDITOR file, it must be unnumbered.)

TRANTEXT is the formal-file-designator for the source code file. Like TRANIN, it can be file-equated to the name of another file.

Controlling Output Destinations from the Compiler

TRANLIST is the formal-file-designator for the destination of compiler listings when LP is the response to the LIST FILE> prompt. The default setting for TRANLIST is DEV=LP. You may, however, change that by means of a file equation. A file equation or the destination default is activated when you respond to the LIST FILE> prompt with LP.

TRANOUT is the formal-file-designator for output from the compiler that, by default, is sent to the standard list device. (The default setting for $STDLIST is your terminal in session mode, the line printer for a batch job.) You can use a file equation to specify a device other than $STDLIST for TRANOUT. If you do this, the compiler prompts, such as SOURCE FILE>, appear on that device, as do the compiler listing and any requested statistics or item definitions. (Note that TRANOUT also controls processor output, including the SYSTEM NAME> prompt.)

If you simply want to redirect your compiler listing, not other compiler output, you can respond to the LIST FILE> prompt with any of the following responses:

- A carriage return or $STDLIST directs the compiler listing to the terminal in a session, to the line printer in a batch job (TRANOUT).
- LP directs the compiler listing to TRANLIST, which is the line printer unless a :FILE command has specified another device for TRANLIST.
- NULL directs the compiler to display errors on the terminal (in a session) or to the line printer (in a job) if ERRS is specified, but to suppress other parts of the listing.
- $NULL directs the listing to a null file, in effect suppressing the listing (the NULL response is preferred.)
- The name of a file directs the listing to a new disc file. If a file of the same name already exists, the compiler asks if you want to purge the existing file.
- A file name preceded by an "*" directs the compiler to back reference a file equation.

TRANCODE is the name of the code file opened and used by the compiler. The default maximum size of this file is 1023 records. If the error message "BINARY FILE FULL" is issued during compilation, use an MPE FILE command to increase the maximum TRANCODE file size. For example, to increase the size to 2000 records, use the following FILE command:

```
:FILE TRANCODE;DISC=2000
```

TRANSACT PROGRAM EXECUTION

This section describes how to execute Transact programs and tells how to control input to and output from the Transact/3000 Transaction Processor.

Executing Transact Programs

Transact programs are executed by running the Transact processor with the following MPE command:

```
:RUN TRANSACT.PUB.SYS
```

After an acknowledgement message, Transact issues the following prompt:

```
SYSTEM NAME>
```

You should respond with the name of the program as specified in the SYSTEM statement of the program you want to execute. In addition to this required response, you may specify one or more optional responses separated by commas. These optional responses specify the mode with which you want to open a data base, and the test mode in which you want to execute, followed optionally by the locations where you want testing to begin and/or end. The syntax of a full response to the SYSTEM NAME> prompt is:

program-name [,*mode* [,*test-mode* [,*start* [,*end*]]]]

where

program-name	is the name of the program as it appears in the SYSTEM statement in the source program. (Required)
mode	is the mode to be used in opening any data bases specified in the program. The mode consists of a single digit indicating one of the open modes specified for DBOPEN in the HP3000 IMAGE reference manual. If you do not specify a mode here or in the SYSTEM statement of your program, Transact opens the data bases in mode 1. Mode 1 allows concurrent modifications to be made to a data base. Any mode specified in the SYSTEM statement of the program takes precedence over a mode specified here.
test-mode	The test mode you want to use to debug your program. Test modes are indicated by a one or two-digit number. (The exact meaning of each test mode is explained in section 7.)

start The location where you want testing to begin; this is the *internal location* number of a line of processor code (see figure 5-1), optionally preceded by a segment number if it is in a segment other than segment 0:

segment number.start.

end The location where you want testing to end; specify as the *internal location* number of a line of processor code, optionally preceded by a segment number if *end* is in a segment other than segment 0:

segment number.end.

For example, suppose you want to open any data bases named in your program in mode 3, and you want to execute in test mode 24 between internal locations 0 and 8, respond to SYSTEM NAME> as follows:

```
SYSTEM NAME> MYPROG,3,24,0,8
```

If the processor cannot find an intermediate processor code file associated with the program name ("IPxxxxxx", where "xxxxxx" is the program name), then it generates an error message and re-issues the SYSTEM NAME> prompt. If you respond with a carriage return to the original or re-issued prompt, then control returns to the MPE operating system.

To bypass answering the SYSTEM NAME> prompt, you can use the INFO= option. This option enables you to specify a system name when you invoke the processor:

```
:RUN TRANSACT PUB.SYS; INFO="APPL01"
```

Note that the INFO= parameters are enclosed in quotation marks. When the INFO= option is used, the SYSTEM NAME> prompt does not appear.

After it locates the code file, the processor generates the following prompt if IMAGE data bases have been defined in the SYSTEM statement and no password supplied:

PASSWORD FOR *databasename*>

You must enter the correct password to open any data bases so specified. If the password is invalid, then you are prompted again for the correct password. If you enter a carriage return in response to the second prompt, control returns to the SYSTEM NAME> prompt and you can request another program or specify other modes. Make sure you enter the password exactly as it is defined; if it is defined with all uppercase letters, enter it with all uppercase letters.

Once your program is executing, you can redisplay the SYSTEM NAME> prompt by pressing the CNTL-Y keys (see CNTL-Y description below) to stop execution and get the > prompt. When the > prompt appears, type INITIALIZE in order to redisplay SYSTEM NAME>. You can then specify another program name, or a new data base mode or test mode.

NOTE: Unlike the programs developed and executed under MPE control, a Transact program can only be executed by running the Transact processor. You cannot execute a Transact program with the MPE RUN command, nor can it be executed through an MPE User Defined Command (UDC).

Controlling Input Sources to the Processor

TRANIN is the formal-file-designator for responses to prompts issued by the processor. The default setting for TRANIN is $STDINX. You may, however, change that by means of a file equation. The processor then reads input from the specified file or device until it encounters an end-of-file condition. If it reaches end-of-file before all prompts are answered, it returns to $STDINX.

TRANSORT is the name of the sort file opened and used by the processor. The default size of this file is 10,000 records divided into 30 extents. If a larger or smaller sort file is desired, then use a file equation to change the size. For example, to reduce the sort file size to 5,000 records, use the following FILE command:

```
:FILE TRANSORT; DISC=5000
```

Controlling Output Destinations from the Processor

TRANLIST is the formal-file-designator for the destination of processor output that is normally sent to the line printer. The default setting for TRANLIST is DEV=LP. You may, however, change the list device by means of a file equation. The file equation or the destination default is activated by the PRINT option to a command or by a SET(OPTION) PRINT statement.

TRANOUT is the formal-file-designator for output from the processor that is normally sent to your terminal in a session or to the line printer in a job ($STDLIST). You can direct such output to another file or device by specifying TRANOUT in a file equation. If you do this, the SYSTEM NAME> prompt, as well as other processor output is sent to the specified file or device. (Note that TRANOUT is also the file designator for output from the compiler.)

TRANVPLS is the name of the file used by the processor to open the VPLUS terminal. If VPLUS forms are to be directed to a device other than your terminal during program testing, use a file equation to specify a particular terminal. For example, suppose your terminal is logical device 20 and you want the VPLUS forms displayed on another terminal, logical device 40, use the following file equation:

```
:FILE TRANVPLS; DEV=40
```

TRANDUMP is the formal-file-designator for the destination of test mode output if you specify a negative test mode in response to the SYSTEM NAME> prompt. Normally, test mode output is sent to your terminal in a session, to the line printer in a job (TRANOUT). If you want test mode output to be sent to another device, you can specify TRANDUMP in a file equation. This is particularly useful when you are using test mode with a program that uses VPLUS and you do not have another terminal handy for the VPLUS forms.

For example, you can direct test mode output to the line printer as follows:

```
     SYSTEM NAME> VTEST,,-34    <---negative test mode directs
                                      test output to TRANDUMP
```

Another method is to direct the test mode output to a disc file by equating TRANDUMP with this file. For example, you can send your test mode output to a file TEST with the following commands:

```
   :BUILD TEST; REC=-80,,F,ASCII
   :FILE TRANDUMP=TEST
   :RUN TRANSACT.PUB.SYS

    SYSTEM NAME> VTEST,,-34     <---test output goes to file TEST
```

Test mode output from the program VTEST is saved in the file TEST, which can be examined or listed using a text editor after your program completes.

A third method is to defer test mode output by setting the output priority to 1. For example:

```
     :FILE TRANDUMP; DEV=,1     <---priority 1 defers test mode output
     :RUN TRANSACT.PUB.SYS

      SYSTEM NAME> VTEST,,-34
```

After your program executes, you can run SPOOK.PUB.SYS to examine the test mode information saved in a spool file.

RUN-TIME CONTROL OF PROGRAM EXECUTION

The processor provides several capabilities that allow the end user to control the execution of a Transact program. This section describes these capabilities and includes the following sections:

- Built-in Processor Commands
- Command Qualifiers
- Special Characters and Keys That Control Execution
- Responses to a MATCH Prompt

Built-in Processor Commands

Certain commands are built into the processor and are available to the user if the program uses a command structure. These commands influence the execution of the processor and include the following:

COMMAND [*command-name*]	Lists all the commands, or lists all the subcommands associated with the specified command-name, in the currently loaded program
EXIT	Generates an exit from the processor
INITIALIZE	Generates an exit from the current program and initiates the loading of a new program You are prompted with SYSTEM NAME> when you enter INITIALIZE.
RESUME	Causes a process to be resumed that was interrupted by a CNTL-Y. (CNTL-Y is explained later in this section under "Special Characters that Control Program Execution".)
TEST[*,mode* [*,range*]]	Causes the processor to execute in test mode for the specified range; if no mode is specified, turns of test mode.

If you define a command in your program with the same name as these built-in processor commands, the program-defined command takes precedence.

Command Qualifiers

Transact program commands, such as ADD CUSTOMER or UPDATE ADDRESS can be qualified by use of command qualifiers. The qualifiers that are recognized by the processor include

FIELD	Indicates the prompted-for field length on 264X series terminals
PRINT	Directs output to the line printer instead of to the user's terminal
REPEAT	Repeats a command sequence until a termination character of "]" is entered from the terminal or from the job stream
SORT	Sorts any data generated by an output verb within the command sequence
TPRINT	Directs a line-printer-formatted display to the user's terminal

For example, when the command string "DISPLAY COMPANY" causes a number of companies to be listed on the terminal, then the end-user can enter the command:

```
PRINT SORT DISPLAY COMPANY
```

This command produces a sorted list of companies on the line printer.

The processor can also accept match selection criteria if the command sequence contains a PROMPT(MATCH) or a DATA(MATCH) statement. For example:

```
PRINT SORT DISPLAY COMPANY = DE^
```

This command produces a sorted list on the line printer of all company names beginning with the letters "DE". (Refer to "Match Specification Characters," below, for an explanation of the character "^".)

The REPEAT option could be added to this command string:

```
REPEAT PRINT SORT DISPLAY COMPANY = DE^, G^
```

The command now produces a sorted list on the line printer of all company names beginning with "DE" and a sorted list of all those beginning with "G".

The REPEAT option can be useful with commands that perform data entry. The command REPEAT ADD TIME-SHEET causes the command sequence to repeat until the user enters a terminating character.

Using FIELD accomplishes the same purpose as SET(OPTION) FIELD="><". It causes the field length of a prompted-for data item to be displayed. For example

```
NAME>                              <
 or
COMPANY>                   <
```

Special Characters and Keys That Control Execution

Several categories of special characters and keys lend powerful programmer and user control to Transact program execution. These characters and keys include:

- CNTL-Y
- Data entry control characters
- Match specification characters
- Field delimiters
- Special keys for use with V/3000 forms

CNTL-Y. The processor recognizes CNTL-Y entered from the user terminal as an operation break. It causes control to return to the Transact command interpreter to await the next user command.

To generate a CNTL-Y, press the CNTL and Y keys simultaneously.

You may choose to use the CNTL-Y feature to halt program execution temporarily in order to enter a TEST or COMMAND command. After using either of these commands, you can continue execution by entering the command, RESUME. This feature is especially useful during program debugging. For example, you can enter the command TEST followed by a test-mode parameter when the program is temporarily halted. When you resume execution, the program executes in the specified test mode. (Refer to section 7 for a description of the test facility).

DATA ENTRY CONTROL CHARACTERS. Several special characters have a predetermined meaning to the processor. They should not be used in any other way as a response to a data entry prompt. They include the following:

]	Terminates the current operation. Control passes to the next higher processing level, which may be the command level.

]] Terminates the current operation. Control passes to command level.

! Generates null responses for all subsequent prompts when entered as a response to an item prompt. It generates null responses for all subsequent sub-item prompts within a compound item when entered as a response to a compound item prompt.

In a command sequence, the effect of the ! response is terminated by the end of the command sequence; if the prompt is not in a command sequence, the ! response remains in effect for all subsequent prompts up to the beginning of a command sequence, if any. The effect of the ! response is also terminated if control passes again through the statement to which the end user responded with !. And, Transact terminates the effect of the ! when it performs automatic error handling.

MATCH SPECIFICATION CHARACTERS. Several special characters help to set up match specifications and are used in response to prompts issued by PROMPT(MATCH) and DATA(MATCH) statements. They are further described in the section entitled "Responding to a Match Prompt." Because of their special meaning, they should only be used for these purposes in character strings. They include the following:

^ Indicates a partial word selection criterion for alphanumeric string data items.

- If "^" is the last character of the entry, then the selection is based on a search for data base or item values that start with the preceding character string.

  For example, when the user enters "DE^" in response to a prompt generated by a PROMPT(MATCH) statement, all values starting with the characters "DE" in a subsequent data base or file operation are selected.

- If "^" is the first character of the entry and it does not occur at the end of the string, then values are selected that end with the input string.

  For example, "^DE" would retrieve all item values that end with the characters "DE".

- If the "^" character appears in any other position in the entry, values are selected that have any character in this position.

  For example, an entry of "^EF^G^" will cause a selection of all values having "EF" in the second and third positions and "G" in the fifth position.

^^ Indicates another partial word selection criterion for alphanumeric string data items. When the user enters "^^" as the trailing characters in an entry, then the selection is based on a search for data base or file item values that contain the preceding character string anywhere within them. For example, an entry of "DE^^" causes a selection of all item values that contain "DE" in any location.

FIELD DELIMITERS. Two characters are used as field delimiters for data entry. They cannot be used as part of an input string unless the field delimiter characters have been suppressed or modified by the SET(DELIMITER) statement. These field delimiters are the comma (,) and the equal sign(=).

If you want to use these characters as is, not as delimiters, you can do one of two things: You can enclose text or responses containing these delimiters within quotes, or you can use a SET(DELIMITER) statement to change the processor's default delimiters to some other character.

Blanks are not normally treated as delimiters; leading and trailing blanks are stripped from responses unless they are enclosed in quotes. You can also use the BLANKS option with data entry verbs (DATA, INPUT, and PROMPT) to allow leading blanks to be be included in a response.

Whatever the delimiter, delimiters can be very useful for responding to prompts. When the user knows the prompt sequence for a particular operation, then he or she does not have to wait for prompts, but can enter a string of data fields separated by delimiters. The processor takes the appropriate action. For example, assuming the default delimiter, suppose an end user responds as follows to the command prompt:

```
>ADD TIME-SHEET = SMITH,77,3,2,V10400,100,....
```

In this example, the processor recognizes the "," as a delimiter, and associates each response with the sequence of prompts that would normally be issued by the ADD TIME-SHEET command.

SPECIAL KEYS FOR USE WITH VPLUS FORMS. Certain special keys may be used during the processing of VPLUS forms sequences:

ENTER When used in a GET(FORM) operation: Normal edit processing as defined in the VPLUS form definition is executed and the data is transferred to the data register. Control passes to the next statement in the program.

When used in a PUT(FORM) operation with a WAIT= option: Control passes to the next statement in the program.

f1-f7 Control passes to the next statement in the sequence.

f8	Control returns to command level unless there are no commands to execute, in which case the Exit/Restart prompt is issued.

This is the default action caused by these keys; this action may be overridden by using the FKEY= or the Fn= options with verbs that use the FORM modifier.

Responding to a MATCH Prompt

The MATCH modifier, available with the PROMPT, DATA, and LIST verbs, provides a powerful mechanism for specifying record selection criteria.

The response to a prompt issued by the PROMPT or DATA verb using the MATCH modifier is set up in the match register. Then the processor uses it in subsequent file or data set accesses. It provides a mechanism by which to specify at run time which records to access.

The response to the prompt may take the following general format:

```
{[relation] value1}             {[relation] value2}
{                  } [connector {                  } ]...
{value1 TO value2 }             {value3 TO value4 }
```

Where:

relation	relational operators for a condition that is other than equal; use one of the following: NE not equal to LT less than LE less than or equal to GT greater than GE greater than or equal to
value	A numeric value or a partial string specification. A string with embedded blanks must be enclosed in quotation marks.
TO	Specifies a range of values bounded by the value preceding and the value following TO.
connector	A logical connector, one of the following: AND Specifies that the record accessed must contain both the value before this operator and the value after this operator OR Specifies that the record accessed must contain either the value before or the value after this operator

The precedence of these connectors is

AND then OR

To illustrate this syntax, assume the program contains the following PROMPT(MATCH) statement:

```
   PROMPT(MATCH) ITEM1 ("Enter match criteria for ITEM1");
```

When executed, this statement adds the item name, ITEM1, to the list register and issues the specified prompt. It then sets up the the match register with criteria entered by the user. For example:

```
   Enter match criteria for ITEM1> GE 500 AND LE 1000
```

This response sets up the match register with two criteria, as shown below:

```
 ______________________________________________________________________
|                                                                      |
|          ITEM1                                      ITEM1            |
|( equal to  OR  greater than )  AND   ( less than  OR   equal to )    |
|   500             500                    1000             1000       |
|______________________________________________________________________|
```

To further illustrate, the following examples are all legal responses to prompts issued by DATA(MATCH) or PROMPT(MATCH) statements:

```
     > 20 TO 30
```

This response sets up the match register to accept any value for the match item that is between 20 and 30 inclusive. Note that the following response gives identical results:

```
     > GE 20 AND LE 30
```

The following response sets up the match register to accept either the value "LAX" or "CGY":

```
     > LAX OR CGY
```

This next response sets up the match register to accept values beginning with "REG" or values beginning with "SAS" and containing the value "CITY" in any position:

```
     > REG^ OR SAS^ AND CITY^^
```

If you want to include one of the standard delimiters, comma or equals, within a value, you must enclose the value in quotes; or you must specify an another delimiter with a SET(DELIMITER) statement. For example, you could respond with:

```
> "San Diego, California"
```

This response ensures that the comma is included in the match register specification.

These examples illustrate how you can set up the match register with responses to DATA(MATCH) or PROMPT(MATCH) statements. You can also set up the match register in your program with SET(MATCH) statements. Using SET(MATCH), you can set up only one selection specification at a time, and you must also make sure the values used in the match criteria are already in the data register. For example, the following four statements place the same criteria in the match register as the response "A OR B" to the prompt issued by a DATA(MATCH) CREDIT statement.

```
LET (CREDIT) = A;
SET(MATCH) LIST (CREDIT);
LET (CREDIT) = B;
SET(MATCH) LIST (CREDIT);
```

AUTOMATIC ERROR HANDLING

The Transact/3000 Processor automatically traps various types of errors encountered during the execution of a program and takes certain predetermined actions. The processor traps errors during data entry and during data base or file operations.

Data Entry Errors

The processor validates a value entered as a response to a data entry prompt according to attributes defined for the data item in the dictionary or the Transact program, that is, data type, field size, decimal field length, integer field length. If it detects an error in the validation procedure, then it issues an appropriate error message on the terminal and re-issues the data entry prompt.

Data Base or File Operation Errors

The processor assumes that a data set or file error has been caused by the user specifying an incorrect value for a key item or other incorrect user input. (Other types of software error conditions should be eliminated before the program is put into production mode.) If the processor detects an error, then it generates an error message and returns program control to an appropriate statement preceding the data set or file operation.

The return location can be the start of the command sequence. In this case, the program reissues the command prompt so the user can start over with a command. The return location could also be to a data entry prompt. For instance, if an error occurs on the second of two data base or file operation verbs, and there is a data entry prompt between the two, the return locations is the statement immediately following the first data base or file operation.

The intention of the logic that determines the return location is to restart at a program point that allows a corrected value to be entered, one that will not cause the error to recur.

If you want to return to a location of your choice where you can process the error, you can use the "ERROR=label" option on the associated file or data set operation statement.

You can test the status register contents with an IF statement within your own error routines at a label specified by the ERROR= option. You can display the contents of the status register by first assigning it to a data item. The data item should be type I(4) to hold the maximum status value. For example:

```
    DEFINE(ITEM) STAT I(4);
    LIST STAT;
    LET (STAT) = STATUS;
    DISPLAY STAT;
```

Transact does not take the ERROR= option when no entries are found with a multiple access verb (DELETE, FIND, OUTPUT, or REPLACE).

Table 5-1 shows the contents of the status register following a data base or file access statement.

Table 5-1. Status Register Following Operations of Data Management Verbs when STATUS Option Not Used

Verb	Status Register Value	
	Operation Successful	Operation Not Successful
DELETE FIND OUTPUT REPLACE	number of entries or records selected (not necessarily number retrieved)	0 = no entries or records found* -1 = no master entry (FIND(CHAIN) and FIND(RCHAIN) only) otherwise undefined
GET PUT UPDATE	1 = one entry or record found	-1 = entry not found otherwise undefined
FILE(READ)	number of bytes read	-1 = end of file otherwise undefined
PATH	number of records in IMAGE detail data set chain	0 = no detail set chain -1 = no master entry otherwise undefined

FILE(CLOSE)
FILE(CONTROL)
FILE(SORT)
FILE(UPDATE)
FILE(WRITE)
	0 = successful operation	undefined

* Entry not found does not activate the ERROR= option

USING THE STATUS OPTION

You can disable several aspects of the processor's automatic processing by using the STATUS option. Use of the STATUS option sets the status register. You can then test the contents of the status register (by using an IF statement) before deciding what further processing should be done. The STATUS option has a different effect depending on whether the statement in which it appears performs data entry or accesses a data base or file.

Note that you can assign a value to the status register with a LET statement. Thus, you can reset status to zero with the following statement:

```
LET STATUS = 0;
```

Data Entry Errors

The status register normally contains the number of characters entered in response to the data entry verbs DATA, INPUT, and PROMPT. When the user enters "]" or "]]" and the verb does not have a STATUS option, an escape to the next processing level is generated as discussed above under "Data Entry Control Characters". The STATUS option suppresses the escape and allows you to test the contents of the register before continuing processing.

Table 5-2 shows the contents of the status register when a data entry verb is used with and without the STATUS option.

Table 5-2. STATUS with Data Entry Verbs

User Entry	Status Register with no STATUS Option	Status Register with the STATUS Option
<CR>	0	0
ABC	3	3
blanks	-3	-3
]	escape	-1
]]	escape	-2

When the STATUS option is used with the CHECK or CHECKNOT option and the user enters a blank, a carriage return, "]", or "]]", then neither CHECK nor CHECKNOT is performed.

The processor validates data for data entry verbs whether or not the STATUS option is used.

Data or File Operation Errors

When you specify the STATUS option with data base and file operation verbs, the automatic error handling described above is suppressed. Instead, you must determine further processing according to the contents of the status register. When STATUS is specified, the effect of the operation is described by the value in the status register:

Status Register Value	Meaning
0	The operation was successful.
-1	An end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or KSAM file system error documentation corresponding to the value.

In addition, STATUS has the following effects:

- It causes accesses and deletions that are normally multiple (iterative) to be single. This affects the iterative verbs: DELETE, FIND, OUTPUT, and REPLACE.
- It suppresses location of the chain head when DELETE, FIND, GET, REPLACE, or OUTPUT is used with the CHAIN modifier. Before using these verbs with the CHAIN modifier, you must locate the chain head with the PATH verb.
- It suppresses the normal rewind performed on a data set or file when DELETE, FIND, GET, REPLACE, or OUTPUT is used with a SERIAL modifier. You should force a rewind by closing the file or data set before using any of these verbs with the SERIAL modifier.

Table 5-3 summarizes the effect of STATUS with data base and file operations verbs.

Table 5-3. STATUS Option with Data Base and File Operation Verbs

Verb	No Automatic Error Handling or Recovery	No Close or Find Before the Operation (CHAIN and SERIAL modifiers)	Multiple Action Suppressed
CLOSE	X		
DELETE	X	X	X
FIND	X	X	X
GET	X	X	
OUTPUT	X	X	X
PATH	X		
PUT	X		
REPLACE	X	X	X
UPDATE	X		

Further information about the STATUS option with these verbs is contained in the verb reference section, Section 6.

TRANSACT/3000 VERBS

SECTION VI

This section contains detailed specifications for using Transact verbs. The verb specifications are arranged in alphabetic order for easy reference. Each specification contains a single-phrase description of the verb's functions. The verb's syntax is enclosed in a box, followed by a general description of the syntax and how the verb is used.

The syntax for most of the verbs is described in terms of "Statement Parts". The specifications for each statement part are provided in detail.

Some verbs, however, have modifiers that change both the syntax and the function of the verb. These verbs are described in terms of "Syntax Options". Each syntax option description consists of the syntax for that option followed by a description of the statement parts for that particular syntax option. Information common to the verb regardless of the particular syntax option precedes the description of the individual syntax options. Verbs with syntax options include DATA, DEFINE, LET, LIST, PROMPT, RESET, and SET.

Examples are provided wherever applicable. The examples may be included within the syntax descriptions, or they may follow the entire verb description.

Table 6-1 groups Transact verbs by primary function and then tells specifically what each verb does.

Table 6-1. Transact Verbs by Function

DECLARATIVE VERBS

Verb	Modifier	Function
DEFINE	ENTRY	Defines program control labels as entry points into program segments.
	INTRINSIC	Declares MPE and subsystem intrinsics for subsequent reference by PROC verb. Refer to Appendix D.
	ITEM	Defines items not defined in dictionary.
ITEM	None	Defines items not defined in dictionary; DEFINE(ITEM) preferred.
SYSTEM	None	Defines data bases, files, or forms used in program; establishes the program environment.

Table 6-1. Transact Verbs by Function (cont'd)

DATA ENTRY AND RETRIEVAL VERBS

Verb	Modifier	Function
DATA	None	Prompts for value; places value in data register.
	ITEM	Prompts for item name; locates name in list register, replaces existing value in data register with value entered in response to verb's second prompt.
	KEY	Prompts for value; places entered value in argument register; does not affect the data register or the key register.
	MATCH	Prompts for value; places entered value in data register, and sets up match criteria in match register based on user response (see section 5 "Responses to a MATCH Prompt").
	PATH	Prompts for value; places entered value in data and argument registers for subsequent keyed access to KSAM file or IMAGE data set.
	SET	Prompts for value; places value in data register if user enters value other than carriage return; leaves existing value in data register if user presses return.
	UPDATE	Prompts for value; places entered value in data register, and item name and value in update register for subsequent use with REPLACE verb to update KSAM or MPE file or IMAGE data set.
DISPLAY	None	Generates display of values from data register.
FORMAT	None	Formats data specified by a subsequent OUTPUT or unformatted DISPLAY verb.
GET	FORM	Displays VPLUS form, retrieves data from form and places retrieved data in data register (see Data Base and File Operation Verbs for other GET functions).

Table 6-1. Transact Verbs by Function (cont'd)

DATA ENTRY AND RETRIEVAL VERBS (cont'd)

Verb	Modifier	Function
INPUT	None	Prompts for value; places value entered by user in input register for subsequent test.
LIST	None	Adds item name to list register.
	KEY	Adds item name to key register only.
	MATCH	Adds item name to list and match registers; uses existing data in data register as match criteria.
	PATH	Adds item name to list and key registers.
	UPDATE	Adds item name to list and update registers; uses existing data in data register as update value.
OUTPUT	None	(See explanation below under "Data Base and File Retrieval Verbs").
PROMPT	None	Prompts for value; adds specified item name to list register, value entered by user to data register.
	KEY	Prompts for value; adds specified item name to key register, value entered by user to argument register.
	MATCH	Prompts for value; adds specified item name to list register; adds value entered by user to data register; and sets up match criteria in match register based on user response (see section 5 "Responses to MATCH prompt").
	PATH	Prompts for value; adds specified item name to list and key registers, value entered by user to data and argument registers.

Table 6-1. Transact Verbs by Function (cont'd)

DATA ENTRY AND RETRIEVAL VERBS (cont'd)

Verb	Modifier	Function
PROMPT	SET	Prompts for value; adds specified item name to list register, and value entered by user to data register if user response is other than a carriage return.
	UPDATE	Prompts for value; adds specified item name to list and update registers, value entered by user to data and update registers.
PUT	FORM	Displays VPLUS form and moves data from the data register to the form (see Data Management and File Operation Verbs for other PUT functions).
SET	FORM	Transfers data from data register to VPLUS buffer for subsequent forms file operations (see Assignment Verbs for other SET functions).
UPDATE	FORM	Transfers data from the data register to the currently displayed form (see Data Management and File Operation Verbs for other UPDATE functions).

Table 6-1. Transact Verbs by Function (Cont'd)

DATA BASE AND FILE OPERATION VERBS

Verb	Modifier	Function
CLOSE	None	Closes an MPE or KSAM file or an IMAGE data set or data base.
DELETE	None CHAIN CURRENT DIRECT PRIMARY RCHAIN RSERIAL SERIAL	Deletes one or more records from KSAM file or IMAGE data set; modifiers determine type of access. DELETE does not delete records from MPE files.
FILE	*CLOSE CONTROL OPEN *READ SORT *UPDATE *WRITE	Operates on MPE files; modifier determines type of operation. * indicates other Transact verb is preferred; see FILE verb discussion for preferred method.
FIND	None CHAIN CURRENT DIRECT PRIMARY RCHAIN RSERIAL SERIAL	Retrieves multiple records from an MPE or KSAM file or multiple entries from an IMAGE data set, and places retrieved data in data register; modifier determines type of access.

Table 6-1. Transact Verbs by Function (cont'd)

DATA BASE AND FILE OPERATION VERBS (cont'd)

Verb	Modifier	Function
GET	None CHAIN CURRENT DIRECT PRIMARY RCHAIN RSERIAL SERIAL	Retrieves a single record from an MPE or KSAM file or a single entry from an IMAGE data set, and places retrieved data in the data register; modifier determines type of access.
	KEY	Locates key value in IMAGE master data set, but transfers no data.
	FORM	Displays VPLUS form and retrieves data entered in form; places retrieved data in data register.
OUTPUT	None CHAIN CURRENT DIRECT PRIMARY RCHAIN RSERIAL SERIAL	Retrieves multiple records from an MPE or KSAM file or multiple entries from an IMAGE data set, and displays the retrieved data; display is formatted according to preceding FORMAT statement, if any; modifier determines type of access.
PATH	None	Establishes a chained access path to an IMAGE data set or a KSAM file; may not be used with MPE files.
PUT	None	Moves data from data register to a record in an MPE or KSAM file, or to an entry in an IMAGE data set.
	FORM	Displays VPLUS form and transfers data from VPLUS buffer to form.

Table 6-1. Transact Verbs by Function (cont'd)

DATA BASE AND FILE OPERATION VERBS (cont'd)

Verb	Modifier	Function
REPLACE	None CHAIN CURRENT DIRECT PRIMARY RCHAIN RSERIAL SERIAL	Updates values in an MPE or KSAM file or in IMAGE data set; uses update register for new values, which may include key values; modifier determines type of access.
UPDATE	None	Updates non-key values in an MPE or KSAM file or in an IMAGE data set.
	FORM	Transfers data from VPLUS buffer to currently displayed form.

Table 6-1. Transact Verbs by Function (Cont'd)

PROGRAM CONTROL VERBS

Verb	Modifier	Function
CALL	None	Transfers control to another Transact program, a REPORT program, or an INFORM program.
END	None	Ends a command sequence, a level, or a program.
EXIT	None	Generates exit from Transact program to MPE control, or from called Transact program to calling Transact program.
GO TO	None	Transfers control to a labelled statement.
IF	None	Executes a simple or compound statement if conditional test is true; optionally executes another simple or compound statement if conditional test is false.
LEVEL	None	Defines processing levels within a program.
PERFORM	None	Transfers control to a labelled statement; use RETURN verb to return control to the statement following PERFORM.
PROC	None	Calls an MPE system intrinsic or procedure in an SL file.
REPEAT	None	Executes a simple or compound statement until a condition is true.
RETURN	None	Used with PERFORM to return control to statement following PERFORM.
WHILE	None	Repeatedly tests a condition clause and executes a simple or compound statement while test is true.

Table 6-1. Transact Verbs by Function (cont'd)

ASSIGNMENT VERBS

Verb	Modifier	Function
LET	None	Assigns result of an arithmetic operation or array manipulation to data register or process control cell.
MOVE	None	Moves data within the data register, or moves a character string or status information to the data register; does not check data type.
RESET	DELIMITER	Resets values of delimiters.
	OPTION	Resets command options.
	STACK	Resets stack pointer in list register.
SET	COMMAND	Performs specified processor or user-defined commands.
	DELIMITER	Sets delimiter to value other than default comma (,) or equals (=).
	FORM	Transfers data to VPLUS form buffer from data register for subsequent forms file operation.
	KEY	Sets value for key and argument registers.
	MATCH	Adds name in list register and value in data register to match register for subsequent data set or file operations.
	OPTION	Sets command options or overrides default execution parameters.
	STACK	Moves stack pointer in list register.
	UPDATE	Adds name in list register and value in data register to update register for subsequent file or data set operation using REPLACE.

Transfers execution to another Transact program or to a REPORT or INFORM program

```
*****************************************************************
*                                                               *
*  CALL file-name[([password][,mode])]                          *
*       [,option-list];                                         *
*                                                               *
*****************************************************************
```

CALL passes control to another Transact program, a REPORT program, or an INFORM program. The called program operates as if it were the main program, but it shares all or part of the calling program's data register space. The called program returns to the calling program with an EXIT statement. Return is to the statement following the CALL statement in the calling program.

When a CALL from a main program is executed, any open files or data sets remain open across the call. When a CALL from a called system to another system is executed, files opened by *this* calling system do *not* remain open for use by the system it calls.

While a called program is executing, both the calling program and the called program are in the memory stack and share the data register.

STATEMENT PARTS

file-name The name of one of the following:

- Another Transact program (as specified in a SYSTEM statement).
- A REPORT program (as specified in a REPORT statement).
- An INFORM program.

If *file-name* names a report, REPORT or INFORM must be specified in the *option-list*.

file-name may also be specified as *(item-name)*, where *item-name* is the name of an item that contains the name of the program or report to be executed.

file-name can be fully qualified as:

file-name.group.account

password	A password for access to the data base used by the called program. This parameter is optional, required only if the called program does not specify a data base password in its SYSTEM statement; Transact prompts for password at run time if not specified here.

password may be specified as:

"*text-string*"	The data base password.
item-name	The name of an item containing the data base password.

It is possible to supply the called program with more than one password. This can be accomplished by defining a compound item of type X or U, where the size of each element in the compound is 8 characters. If a list of passwords is passed to the called program, the first password on the list is used to open the first data base specified in the SYSTEM statement, the second password on the list is used to open the second data base specified, and so on.

mode	The mode in which the data base used by the called program is to be opened. This parameter is optional, and may be specified here if SYSTEM statement in called program does not specify mode in which to open a data base; if mode is specified in the SYSTEM statement of the called program, that mode overrides the mode specified here. Default=1

mode may be specified as:

digit	Number 1 to 8.
item-name	Name of item containing *mode* value.

It is possible to specify a list of modes to be passed to the called program. This is done by passing a compound item of type I(2). The mode list may be passed only if a password list is also passed. Like the password list, the mode list is used to open each of the data bases specified in the SYSTEM statement with a different mode.

option-list One or more of the following options separated by commas:

DATA=*item-name* The location in the data register where the called program may begin using space. This space includes the location of the specified item. If *item-name* is an "*", the called program cannot use any space already used by the calling program. If DATA= is omitted, CALL resets the list register before transferring to the called program.

SIZE=*number* The number of words of data register space that the called program can use. If DATA=*item-name* is also specified, space starts at the location assigned to *item-name*. This space cannot be larger than the number of unused words in the data register and must start on a word boundary.

NOTE: When Transact CALLs a subprogram, the data register space allocated to the subprogram is determined by the DATA= and SIZE= parameters of the CALL statement, *not* the DATA= option of the SYSTEM statement in the called program. The total size of the data register, however, is determined by the DATA= option of the main program's SYSTEM statement.

SWAP A request to write part of the caller's stack space out to a temporary MPE file before the CALL is made. When control is transferred back to the calling program, the MPE file is read back and the stack is restored.

Use of the SWAP option increases the number of nested calls that can be made before stack space is exhausted. There is some overhead, however, associated with using the SWAP option. Therefore it should be used only if available stack space is very limited.

INFORM A request to run the INFORM report specified by *file-name*. None of the INFORM menus are displayed. If needed, a data base password is prompted for. After the INFORM report is complete, control returns to the statement following the call.

REPORT A request to run the REPORT report specified by *file-name*. If needed, a data base password is prompted for. After the report is complete, control returns to the statement following the call.

EXAMPLES

```
CALL INVMGT ("X43",7),
     DATA = ORDER,
     SIZE = 1000;
```

calls the INVMGT program, provides a password for opening any data bases used by INVMGT and allows the data base to be opened in mode 7 for exclusive read access. INVMGT may use data register space beginning at the item named ORDER, and it may use 1000 words of space.

```
DATA(MATCH) SYSNAME("Enter name of application to run :");
SET(KEY) LIST(USER);
GET(CHAIN) PASSWORD-DSET, LIST(SYSNAME, PASSWORD);
CALL (SYSNAME) (PASSWORD, 5),
     DATA=*;
```

The user is prompted for the name of the application to run. Then the password needed to access the data base is retrieved from the PASSWORD-DSET detail data set.

```
DEFINE(ITEM) PASSWORD-LIST 2 X(8) :
             MODE-LIST 2 I(2) :
             MODE-ITEM I(2) = MODE-LIST(1);
MOVE (PASSWORD-LIST) = "PASS1   PASS2   ";
LET (MODE-ITEM) = 1;
LET OFFSET(MODE-ITEM) = 2;
LET (MODE-ITEM) = 5;
CALL ORDPROC (PASSWORD-LIST,MODE-LIST), DATA=*;
```

This example shows how multiple passwords and multiple modes can be passed to a called program.

Closes an MPE or KSAM file or an IMAGE data set or data base

```
**********************************************************************
*                                                                    *
*  CLOSE file-name[,option-list];                                    *
*                                                                    *
**********************************************************************
```

CLOSE closes and rewinds an MPE or KSAM file or an IMAGE data set, or closes the entire data base. Except to rewind or set a file or data set to its beginning, you need not use CLOSE. Transact automatically closes all files and data sets at the end of a command sequence and at the end of a program.

You typically use CLOSE to set a file or data set to its beginning when you are planning to use the STATUS option with a data base access verb that performs serial access; these verbs are FIND(SERIAL), GET(SERIAL), DELETE(SERIAL) or OUTPUT(SERIAL). You would also use CLOSE before a FILE(SORT) statement.

Two special forms of the CLOSE statement may be used to close print files before you exit from Transact:

● CLOSE $FORMLIST	Closes the spool file used by the VPRINTFORM intrinsic of VPLUS.
● CLOSE $PRINT	Closes the print file TRANLIST. This statement is useful for directing output to the printer using SET(OPTION) PRINT without terminating your program.

STATEMENT PARTS

file-name The file or data set to be closed. If the data set is not in the home base as defined in the SYSTEM statement, you must specify the base name in parentheses as follows:

set-name(base-name)

You may close an entire IMAGE data base by specifying *file-name* as a data base with the following format:

@[(*base-name*)]

To close the home base, omit *base-name*; to close any other base, specify a *base-name*.

option-list	One or more of the following options separated by commas:
ERROR=*label* ([*item-name*])	Suppress the default error return that the processor normally takes. Instead, the program branches to the statement identified by *label*, and the stack pointer for the list register is set to the data item *item-name*. The processor generates an error at execution time if the item cannot be found in the list register. If you do not specify an item name, as in ERROR=*label*();, the list register is cleared. If you use an * instead of *item-name*, as in ERROR=*label*(*);, then the list register is not touched. For more information, see "Automatic Error Handling" in section 5.
NOMSG	Suppress the standard error message produced by the processor as a result of a file or data base error.
STATUS	Suppress the processor action defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option. When STATUS is specified, the effect of a CLOSE statement is described by the value in the status register:

Status Register Value	Meaning
0	The CLOSE operation was successful.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation that corresponds to the value.

You can use the STATUS option with CLOSE to do exit processing on an error. For example:

```
CLOSE KSAM-FILE,
      STATUS;
IF STATUS <> 0 THEN
   GO TO ERROR-CLEANUP;
```

EXAMPLES

```
CLOSE ACCREC,
  ERROR = FIX (CUST-NAME);
```

This statement closes the file ACCREC. If an error occurs, it passes control to the statement labelled FIX and sets the list register to CUST-NAME.

DATA

Prompts for a value and changes the appropriate location in the data, argument, match, and/or update registers

```
*****************************************************************************
*                                                                           *
* DATA[(modifier)] [item-name] [("prompt-string")][,option-list]            *
*                  [:item-name...]...;                                      *
*                                                                           *
*****************************************************************************
```

DATA prompts the user for a value and, depending on the syntax option chosen, places the value in one or more registers. The registers affected depend on the verb modifier. Available modifiers are:

● none	Place value in data register (see Syntax Option 1).
● ITEM	Prompt for item name and if found, place value in data register (see Syntax Option 2).
● KEY	Place value in argument register (see Syntax Option 3).
● MATCH	Place value in data register; set up match criteria in match register (see Syntax Option 4).
● PATH	Place value in data register and in argument register (see Syntax Option 5).
● SET	Place value in data register unless user presses carriage return (see Syntax Option 6).
● UPDATE	Place value in data register; place item name and value in update register (see Syntax Option 7).

The user enters a value in response to a prompt-string or to the item-name. At execution time the processor validates the input value as to type, length, and other characteristics defined in the Data Dictionary or by a DEFINE(ITEM) statement. It validates the data before the register is modified. If the processor detects an error, then it displays an appropriate error message and reissues the prompt.

You normally use the DATA verb to change the value for a data item that has already been specified in the list register. DATA searches the list register from the top of the stack to the bottom to find the requested item-name. If there are multiple occurrences of the same item in the list register, it uses the last one placed on the list.

STATEMENT PARTS

modifier	Changes or enhances the action of DATA; often indicates the register to which the input value should be added or the register whose value should be changed (see "Syntax Options", below).
item-name	The name of the data item in the list register whose value should be added or changed in the appropriate register.
*	The item at the top of the list register; that is, the one referenced by the last LIST or PROMPT statement unless explicitly changed by a previous SET or RESET command.
prompt-string	The string that prompts the terminal user for the input value; if not specified, the user is prompted by the item name or by an entry text specified in the DEFINE(ITEM) statement or in the dictionary, if one exists.
option-list	A field specifying how the data should be formatted and/or other checks to be performed on the entered value. Include one or more of the following options (separated by commas) unless you use the ITEM modifier (Syntax Option 2):

BLANKS	Do not suppress leading blanks supplied in the input value; leading and trailing blanks are normally stripped.
CHECK= *set-name*	Check input value against the IMAGE master set *set-name* to ensure that the value already exists. If the condition is not met at execution time, the processor displays an appropriate error message and re-issues the prompt. (Note: you cannot use CHECK= with a KSAM or MPE file, nor can you use it in a DATA(MATCH) statement.)
CHECKNOT= *set-name*	Check input value against the IMAGE master set *set-name* to ensure that the value does not already exist. If the option condition is not met at execution time, then the processor issues an appropriate error message and re-issues the prompt. (Note: you cannot use CHECKNOT= with a KSAM or MPE file, nor can you use it in a DATA(MATCH) statement.)
NOECHO	Do not echo the input value to the terminal.
NULL	Fill item with ASCII null characters (binary zeros) instead of blanks.
RIGHT	Right-justify the input value within the register field.

STATUS	Suppress normal processing of "]" and "]]", which cause an escape to a higher processing or command level. Instead, set the status register to -1 if "]" is pressed, and to -2 if "]]" is pressed. If the user enters one or more blanks, then the status register contains -3. (The status register normally contains the number of characters entered in response to a prompt.) The STATUS option allows you to control subsequent processing by testing the contents of the register with an IF statement. If the CHECK or CHECKNOT option is also used, then "]", "]]", a carriage-return, or one or more blanks suppress the DATA operation and control passes to the next statement.

SYNTAX OPTIONS

```
(1) DATA {item-name} [("prompt-string)"][,option-list];
         {    *    }
```

DATA with no modifier places the value entered as a response to *prompt-string* in the data register. It is added in an area associated with the current data item if "*" is used or with *item-name* if it is specified.

```
(2) DATA(ITEM) "prompt-string" [,REPEAT];
```

DATA(ITEM) issues a prompt (*prompt-string*) to request an item name. When the user enters an item name in response to this prompt, the processor looks for this item in the list register. If the item name cannot be found, it displays an error message and re-issues the prompt. If the item name is in the list register, this item name is issued as a second prompt to which the user responds with a value. If the entered value passes all edit checks, it is placed in the data register area associated with the item name. Otherwise, the user is prompted for another value. If the user responds with a "]", the processor re-issues the *prompt-string* prompt. If the user responds with "]]", the processor returns to command mode.

The ITEM modifier is typically used to update or correct one or more values in the data register.

If you use the REPEAT option, then the operation is repeated until a termination character (]) or a null response (carriage return) is entered in response to the *prompt-string* prompt.

```
(3) DATA(KEY) {item-name} [("prompt-string")][,option-list]
              {    *    }
              [:item-name...]...;
```

DATA(KEY) places the value entered as a response to *prompt-string* in the argument register. If *item-name* is specified, this name is used as the prompt for user input, unless this name is overridden by a *prompt-string*. If "*" is specified, then the current name in the the key register is used as the prompt for user input. The key register is changed by this verb only if it is empty. If the key register is not empty, this verb does not change the item name already there.

```
(4) DATA(MATCH) {item-name}[("prompt-string")][,option-list]
                {    *    }
                [:item-name...]...;
```

DATA(MATCH) places the value entered as a response to *item-name* or "*prompt-string*" in the data register. It places the value in the data register in an area associated with the current data item if the "*" is used or with *item-name* if it is specified. The item name and value are also placed in the match register as a selection criterion for subsequent data base or file operations.

You cannot specify either CHECK= or CHECKNOT= with DATA(MATCH).

User responses to the DATA(MATCH) prompt are further explained in "Responding to a Match Prompt" in section 5.

The MATCH option allows one or more of the option-list items allowed with all DATA options (see list above). You may also select one of the following, which specify that a match selection is to be performed on a basis other than equality.

MATCH *option-list*:

NE	Not equal to
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
LEADER	Matched item must begin with the input string; equivalent to the use of trailing "^" on input
SCAN	Matched item must contain the input string; equivalent to the use of trailing "^^" on input

TRAILER	Matched item must end with the input string; equivalent to the use of a leading "^" on input

```
(5) DATA(PATH) {item-name}[("prompt-string")][,option-list]
               {    *    }
               [:item-name...]...;
```

DATA(PATH) places the value entered as a response to *prompt-string* in the data register. It is placed in the data register in an area associated with the current data item if the "*" is used or with *item-name* if it is specified. The value is also placed in the argument register and the item name in the key register for subsequent keyed access to KSAM files or IMAGE data sets.

```
(6) DATA(SET) {item-name}[("prompt-string")][,option-list]
              {    *    }
              [:item-name...]...;
```

DATA(SET) places the value entered as a response to *item-name* or *prompt-string* in the data register. It is placed in the data register in an area associated with *item-name*, if it is used, or with the current item if "*" is used.

If the user responds to the prompt with a carriage return, then the existing value in the data register is not touched. Note that this differs from the other DATA statements which add blanks to the data register if the user responds with a carriage return.

If you use the CHECK= or CHECKNOT= options and the specified condition is not met, the item remains in the data register. In this case, you should reset the data register to the previous item to avoid creating an endless loop should the end user respond with a carriage return to the reissued prompt. Both CHECK= and CHECKNOT= look for the item in the IMAGE master set even if the user enters a carriage return.

The primary use of the SET modifier is to update values in the data register for existing items in the list register.

(7) DATA(UPDATE){*item-name*}[("*prompt-string*")][,*option-list*]

```
                 {    *    }
                 [:item-name...]...;
```

DATA(UPDATE) places the value entered as a response to *prompt-string* in the data register. It is placed in the data register in an area associated with the current data item if the "*" is used or with *item name* if it is specified. The item name and value are also placed in the update register for subsequent use with the REPLACE verb.

EXAMPLES

```
DATA(KEY) ACCT-NO ("Account number?"),
    CHECK=ACCOUNT-MASTER;
```

This example asks the user for an account number, which is placed in the argument register for subsequent access to the data set, ACCOUNT-MASTER . The value is checked first, however, to see if it already exists in ACCOUNT-MASTER. If it does not, then an error message is displayed and the prompt is re-issued.

```
DATA(SET) QUANTITY("New stock quantity?");
```

This example asks the user for a response. If the response is a carriage return, the data register is not changed. If a value is entered, the new value replaces the existing value in the data register space allocated to the item QUANTITY.

```
DATA ADDRESS ("Enter customer address"):
     CITY ("Enter city"):
     STATE (Enter 2-letter state code"):
     ZIP (Enter 5-digit zip code");
```

In response to the prompt for ADDRESS, the user can enter the entire address with each item separated by commas; or the user can enter one item of the address at a time. If the entire address is entered at once, the remaining item prompts are not issued.

For example, the following dialogue could occur:

```
Enter customer address> 312 Alba Road, San Jose, CA, 95050
```

Alternatively, if the user wants to wait for each prompt, the dialogue could be:

```
Enter customer address> 312 Alba Road
Enter city> San Jose
Enter 2-letter state code> CA
Enter 5-digit zip code> 95050
```

In either case, the entered data is moved to the data register locations associated with ADDRESS, CITY, STATE, and ZIP. If the user presses return in response to any single prompt, the associated area of the data register is cleared. If you want the return key to leave the existing data, you must use a DATA(SET) statement.

Specifies definitions of item names, names of MPE system intrinsics, or segmented program control labels to be used by the compiler

```
*******************************************************************
*                                                                 *
*  DEFINE(modifier) definition-list;                              *
*                                                                 *
*******************************************************************
```

The DEFINE statement is used to define items, entry points into program segments, or intrinsics called with the PROC statement. DEFINE statements are generally the first statements that follow the SYSTEM statement in a Transact program.

The function of the DEFINE statement depends on the modifier you choose, and for DEFINE(ITEM) on the particular syntax option. The allowed modifiers and the associated syntax options are:

● ENTRY	Define a program control label within a segment as global to the entire program (Syntax Option 1).
● INTRINSIC	Define an MPE system intrinsic to be called by the PROC verb (Syntax Option 2).
● ITEM	Define one or more item names (Syntax Option 3).
● ITEM	Define a synonym for an item name (Syntax Option 4).
● ITEM	Define a marker item, which is a position in the list register (Syntax Option 5).
● ITEM	Define an item name whose attributes are to be satisfied by the processor at execution time (Syntax Option 6).

The *modifier* and *definition-list* depend on the syntax option you choose.

SYNTAX OPTIONS

(1) DEFINE(ENTRY) *label*[:*label*]...;

The ENTRY modifier causes a statement label within a program segment to be global to the whole program so that statements in any segment can reference this label. You need not define entry point labels within the root segment (segment 0).

(2) DEFINE(INTRINSIC) *intrinsic-name*[:*intrinsic-name*]...;

The INTRINSIC modifier defines MPE system intrinsics that are called by the PROC verb. Declaring the intrinsic in this manner causes automatic linking rather than loading of the intrinsic at execution time, thus enabling significant load-time savings. System intrinsics are treated the same as user-written procedures.

Not all MPE intrinsics may be specified in a DEFINE(INTRINSIC) statement. (Refer to appendix D for a list of the intrinsics that are recognized by the Transact compiler.) If you include an intrinsic name that is not recognized by the compiler, a compile time message will be issued. If this occurs, remove the unrecognized intrinsic from the DEFINE(INTRINSIC) statement; the intrinsic will be loaded at run time.

No compile-time parameter verification is done for system intrinsics used in a Transact program *unless* they are declared using the DEFINE(INTRINSIC) statement.

(3) DEFINE(ITEM) *item-name* [*count*]
[*type*(*size*[,*decimal-length*[,*storage-length*]])]
[=*parent-name*[(*position*)]]
[,ALIAS=(*alias-reference*)]
[,COMPUTE=*arithmetic expression*]
[,EDIT="*edit-mask*"]
[,ENTRY="*entry-text*"]
[,HEAD="*heading-text*"]
[,OPT]
[:*item-name* ...]...;

This option defines an *item-name* not defined in the dictionary. It also redefines for this program only, items already defined in the dictionary. Any number of *item-names*, separated by colons (:) can be specified in a single DEFINE(ITEM) statement.

item-name	The name of a data item or system variable to which the definition applies. When it refers to a data item, *item-name* identifies an item that exists in a data base or file used by the Transact program or that is to be used as a temporary variable. This item may or may not be included in the dictionary. The first character must be alphanumeric, and the other characters may be alphabetic (A-Z, upper or lowercase), digits (0-9), or any ASCII characters except , ; : = < > ( ) " or a blank space. *item-name* can be up to 16 characters long.

Five system variables can be specified as an *item-name*: $CPU, $DATELINE, $PAGE, $TIME, and $TODAY. Note that only the EDIT= and HEAD= options are valid with these variables.

count	The number of occurrences of the item if it is a sub-item within a compound item. (All of the sub-items have the same attributes.) Example: `DEFINE(ITEM) SUB 24 X(30);` SUB is defined as a compound item that has 24 30-character sub-items.

type	The data type:
	X = any ASCII character
	U = uppercase alphanumeric string
	9 = numeric ASCII string (leading zeroes stripped)
	Z = zoned decimal (COBOL format)
	P = packed decimal (COBOL comp-3)
	I = integer number
	J = integer number (COBOL comp)
	K = logical value (absolute binary)
	R = real, or floating point, number
	E = real, scientific notation
	
	If *type* is followed by a "+", then the item is unsigned, and can have positive values only. Data entry values are validated as positive and, if the type is Z or P, positive unsigned value formats are generated.
	
	Items defined as type E are displayed in the format: *n.nnE+nn*, but cannot be entered in this format; they may be entered as integer or real numbers.
	
	(Refer to the discussion of Data Items in section 3 for details on these data types.)

size	The number of characters in an alphanumeric string or the number of digits, plus decimal point if any, in a numeric field. Transact adds a display character for the sign to the specified size of numeric items (types Z, P, I, J, R, and E) unless the item type is defined as positive only with a "+". You should be aware of this extra display character when transferring data to VPLUS numeric fields. (Refer to table 3-1 for the relation between the specified size, its storage allocation, and display requirements.)

If both *type* and *size* are omitted, the dictionary definition of the item is used.

decimal-length	The number of decimal places in a zoned, packed, integer or floating point number, if any. The maximum *decimal-length* is 1 less than the maximum *storage-length* of the item.
storage-length	The byte length of the storage area for the data item, which overrides the length calculated by the compiler from the type, size, and decimal length values.

Storage length of X and U type items is limited only by the size of the data register. The maximum size of the numeric item types 9, Z, P, I, J, and K is 27 digits or characters, unless a decimal is included in which case the maximum size is 28 characters or digits including the decimal point. For R and E types, the maximum recommended size is 22 characters and digits, to allow for 17 accurate digits in the mantissa, a decimal point, the sign of the exponent, the letter E, and 2 digits for the exponent.

parent-name Name of parent if you are defining a child item; redefines all or part of a parent item name defined elsewhere in the program or in the dictionary. (Similar to an equate in SPL or an equivalence in FORTRAN.)

The following is an example of redefinition of a parent item defined as "NAME".

```
DEFINE(ITEM) NAME X(32):
             FNAME X(10)=NAME(1):
             MIDINIT X(1)=NAME(11):
             LNAME X(21)=NAME(12);
```

position The byte position in the parent item that is the starting position of the child item. Begin counting at position 1. Default = 1.

In the following example, the child item YEAR starts in position 1 of the parent item DATE, MONTH starts in position 3, and DAY in position 5.

```
DEFINE(ITEM) DATE X(6):
             YEAR X(2)=DATE(1):
             MONTH X(2)=DATE(3):
             DAY X(2)=DATE(5);
```

ALIAS=(*alias-reference*)

Specify other names (aliases) by which *item-name* is known, where *alias-reference* has the form:

item-name1[(*file-list1*) [,*item-name2*[(*file-list2*)]]...]

The item defined as *item-name* is called *item-name1* in any of the files or data sets in *file-list1*, *item-name2* in any of the files in *file-list2*, and so forth. If *file-list1* is omitted, *item-name1* is the only *alias-reference* allowed. A file list may consist of file or data set names separated by commas. If a referenced data set is not in the home base specified in the SYSTEM statement, the base name must be specified as *set-name*(*base-name*).

Note that Transact does not retrieve alias definitions from the dictionary. You must define any aliases in a DEFINE(ITEM) statement in your program.

An alias ensures that when you reference *item-name* in your program, this name is associated with the other names by which the item is known in files or data sets. You always reference such an item by its primary name, not its alias.

The following example defines the item QTY-ORD, which is known in the file ORDERS as QUANTITY and in the file ORD-MAST as QUANT-ORD. Note that all aliases must have the same attributes as the primary item:

```
DEFINE(ITEM) QTY-ORD I(4),       <<use name QTY-ORD in program>>
   ALIAS=(QUANTITY(ORDERS),
          QUANT-ORD(ORD-MAST));
```

COMPUTE= *arithmetic-expression*

Arithmetic expression that specifies the computation to be performed before the item is used in a display statement or sort function. It may contain two or more variables separated by one or more arithmetic operators. Use the form required by the LET statement.

EDIT= "*edit-string*"

Default edit mask used for the item's value in any display (see the DISPLAY and FORMAT statements for an edit mask feature description).

ENTRY= "*entry-text*"

Text string used as the default prompt string for the item when used by the PROMPT and DATA statements.

HEAD= "*heading-text*"

Text string used as the default heading for the item in any display function.

OPT	OPT is used in combination with the compiler control option, OPTI. When OPT is specified for an item, the compiler does not store the item's textual name in the code file if the OPTI control option has been specified for a compile run. OPT used in conjunction with OPTI saves data segment stack space at execution time. (Refer to section 3 for a discussion of the OPTI compiler option.) It is your responsibility to ensure that the item's textual name is not required within the program. An item name is needed for a prompt string, display item heading, or for the LIST= option of verbs that access an IMAGE data base.

(4) DEFINE(ITEM) *item-name=item-name1*

This option defines a synonym for an item defined elsewhere in the program or in the dictionary.

item-name

A synonym for *item-name1* where *item-name1* is defined elsewhere in the program or in the dictionary. *item-name* assumes the definition of *item-name1*, but the processor always references *item-name1* in any file or data set operation.

Use this option to provide an alternate name for an item. The synonym *item-name* exists only while the program executes; it is not an item name in a file or data set, or the dictionary. For example:

```
DEFINE(ITEM) PROD-NO 9(10):
             PRODUCT-NUM=PROD-NO;
```

This statement defines the item PROD-NO as a type 9 10-digit item, and defines PRODUCT-NUM as a synonym for PROD-NO. The same item can now be called either PRODUCT-NUM or PROD-NO within the program.

(5) DEFINE(ITEM) *item-name* @[:*item-name* @]...;

This option defines a marker item. A marker item marks a point in the list register, but it reserves no space in the data register. The marker item must be defined with the DEFINE(ITEM) statement and placed in the list register with the LIST statement.

A marker item can be referenced by list pointer operations and list range options. Marker items are useful in conjunction with the SET modifier on the PROMPT verb. The PROMPT(SET) statement causes the contents of the list register to be defined at execution time.

The following sequence of Transact statements shows an appropriate use of the marker item:

```
DEFINE(ITEM) MARKER1 @: MARKER2 @;
LIST MARKER1;
PROMPT(SET)EMPL:DEPT:PHONE:ROOM:LOCATION;
LIST MARKER2;
UPDATE EMPLOYEES,LIST=(MARKER1:MARKER2);
```

The first statement defines marker 1 and marker 2. The second statement assigns space in the list register to marker 1. The third statement prompts for new information about employees. It is not known which and how much information will be entered. When data entry is complete, a second marker is assigned in the list register. Then the EMPLOYEES file is updated with all the information in the list and data registers between marker 1 and marker 2. (This example assumes that the current entry has been set up appropriately by a previous get of the EMPLOYEES data set.)

Generally, you know only the start and end positions of the data entered, but not how many entries will be made. By placing marker items in the list register using the LIST statement, you are able to pass a variable number of items to the EMPLOYEES file.

(6) DEFINE(ITEM) *item-name* * [:*item-name* *]...;

This option defines an item name whose attributes should be satisfied by the processor at execution time rather than by the compiler at compile time. Note that only the basic attributes can be resolved at execution time; these are count, type, size, decimal-length, and storage length, not such secondary attributes as heading text or entry text.

EXAMPLES

The following example shows how to define a key item for KSAM file access, assuming the key is a 10-character item starting in byte 3 of an 80-character record.

```
DEFINE(ITEM) RECORD    X(80):
      DEL-CODE  I(2) = RECORD(1):          <<reserve 1st word for delete code>>
      KEY       X(10)= RECORD(3);

MOVE (KEY) = "A123456789";                 <<assign value to key >>
SET(KEY) LIST(KEY);                        <<use key value to find chain head>>
FIND(CHAIN) KFILE,
     LIST=(RECORD);                        <<read entire record >>
```

In another example, a portion of a key is defined as a "generic key":

```
DEFINE(ITEM) RECORD    X(80):
     DEL-CODE  I(2)  = RECORD(1):
     KEY       X(10) = RECORD(3):
     GEN-KEY   X(2)  = RECORD(3);
```

The key search is similar to that shown above; use a GEN-KEY value to locate all records with key values starting with the same first two characters.

Deletes file or data set entries

```
*****************************************************************
*                                                               *
*  DELETE[(modifier)] file-name[,option-list];                  *
*                                                               *
*****************************************************************
```

DELETE specifies the deletion of one or more file entries. For multiple deletions, the entries to be deleted are determined by match criteria specified in the match register. If you do not specify match criteria for a multiple deletion, DELETE deletes all entries in a chain or in the entire file or data set, depending on the modifier.

DELETE cannot be used with MPE files.

STATEMENT PARTS

modifier	To specify type of access to the data set or file, choose one of the following modifiers:
none	Delete an entry from an IMAGE master set based on the key value in the argument register; this option does not use the match register.
CHAIN	Delete entries from an IMAGE detail set or a KSAM chain. The entries must meet any match criteria set up in the match register. The contents of the key and argument registers specify the chain in which the deletion is to occur. If no match criteria are specified, all entries are deleted. Match criteria must be included in a LIST= construct.
CURRENT	Delete the last entry that was accessed from the file or data set.
DIRECT	Delete the entry stored at the specified record number in an MPE or KSAM file, or an IMAGE detail or master data set. Before using this modifier, you must store the record number as a doubleword integer in the item specified by the RECNO=option.
PRIMARY	Delete the IMAGE master set entry stored at the primary address of a synonym chain. The primary address is located through the key value in the argument register.

RCHAIN	Delete entries from an IMAGE detail set or a KSAM chain in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.
RSERIAL	Delete entries from a file in the same manner as the SERIAL option, except in reverse order. For a KSAM file, this operation is identical to SERIAL.
SERIAL	Delete entries in serial mode from an MPE or KSAM file or from an IMAGE data set that meet any match criteria set up in the match register. If no match criteria are specified, all entries are deleted. If match criteria are specified, the match items must be included in a LIST= option.

file-name The file or data set to be accessed in the deletion. If the data set is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows:

set-name(*base-name*)

option-list One or more of the following options, separated by commas:

ERROR=*label* ([*item-name*]) Suppress the default error return that the processor normally takes. Instead, the program branches to the statement identified by *label*, and the stack pointer for the list register is set to the data item *item-name*. The processor generates an error at execution time if the item cannot be found in the list register.

If you omit *item-name*, as in ERROR=*label*();, the list register is cleared.

If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not touched.

For more information, see the section entitled "Automatic Error Handling" in section 5.

LIST= (*range-list*) The list of items from the list register to be used for the DELETE operation. If the LIST= option is omitted, all the items in the list register are used.

Only the items specified in a LIST= option have their match conditions applied if match conditions are set up in the match register. (The match register may be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.)

Each retrieved entry is placed in the area of the data register indicated by LIST= before any PERFORM= is executed, and then the delete is performed.

The options for *range-list* and the items they cause DELETE to access include the following:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current stack pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.
(*item-name1*, *item-name2*, ... *item-name*n)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM and MPE, items must be specified in the order of their occurrence in the record. This option incurs some system overhead.
()	A null item list. That is, delete the entry or entries, but do not retrieve any data.

LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked the whole time that DELETE executes. If LOCK is not specified, the file or data base is locked before each entry is retrieved, remains locked while the entry is processed by any PERFORM= statements, but is unlocked briefly before the next entry is retrieved.
NOCOUNT	Suppress the message normally generated by the processor to indicate the number of deleted entries.
NOMATCH	Ignore any match criteria set up in the match register.
NOMSG	Suppress the standard error message produced by the processor as a result of a file or data base error.

PERFORM=*label*	Execute the code following the specified label for every entry retrieved by the DELETE verb before the DELETE operation. The entries may be optionally selected by match criteria. This option allows operations to be performed on retrieved entries without your having to code loop-control logic. You may nest up to a maximum of 10 PERFORM options.
RECNO=*item-name*	With the DIRECT modifier: you must define *item-name* to contain the doubleword integer address of the record to be deleted. With other modifiers: Transact returns the record number of the deleted record in the doubleword integer *item-name*.
SINGLE	Delete only the first selected entry.
SOPT	Suppress the processor optimization of IMAGE calls. This option is intended to support a data base operation in a performed routine that is called recursively. It allows a different path of the same detail data set to be used at each recursive entry rather than optimizing to the same path. It also suppresses generation of an IMAGE call list of "*" after the first call is made.
STATUS	Suppress processor actions defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option. When STATUS is specified, the effect of a DELETE statement is described by the value in the status register:

Status Register Value	Meaning
0	The DELETE operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with DELETE:

- Normal multiple accesses/deletions become single,
- The normal rewind done by the DELETE is suppressed, so CLOSE should be used before DELETE(SERIAL).
- The normal find of the chain head by the DELETE is suppressed, so PATH should be used before DELETE(CHAIN).

In the following example, the programmer wants to be sure that an entry is not in MASTER-SET. Therefore, there are two acceptable conditions: either a status register value of zero (delete successful) or a status register value of 17 (IMAGE error 17 meaning record not found) is acceptable.

```
DELETE MASTER-SET,
      LIST=(KEY-ITEM),
      STATUS;
IF STATUS = 17, 0  THEN
      DISPLAY "ENTRY REMOVED"
ELSE
  DO
      DISPLAY "ERROR ON DELETE FROM "
      "MASTER-SET";
      GO TO ERROR-CLEANUP;
  DOEND;
```

EXAMPLES

```
PROMPT(MATCH) DEBT-LEVEL,LT;

DELETE(CHAIN) DEBT-DETL,
      LIST=(DEBT-LEVEL);
```

This example deletes all entries that contain a DEBT-LEVEL less than the number entered by the user. DEBT-LEVEL is required in the LIST parameter because DELETE reads each record in the chain into the data register area associated with DEBT-LEVEL in order to check the match condition before deleting the entry.

```
PROMPT(MATCH) ZIP ("DELETE ZIP CODE");

DELETE(RSERIAL) DETAIL-SET,
      SINGLE,
      LIST=(NAME:ZIP),
      PERFORM=LISTIT;
```

This example deletes only the last entry in the data set that matches the zip code entered by the user.

DISPLAY

Produces a display of values from the data register

```
**********************************************************************
*                                                                    *
*  DISPLAY[(TABLE)] [display-list]];                                 *
*                                                                    *
**********************************************************************
```

DISPLAY generates a display from values in the data register. The display can be formatted and enhanced by character strings specified in the *display-list*. If you do not specify a format, the display is formatted by any active FORMAT verb.

STATEMENT PARTS

none or TABLE	(No *display-list*.) The processor generates a display according to the specifications of an active FORMAT statement. If there is none, the following default formatting occurs:

- Values are displayed in the order in which they appear in the data register.
- A heading consisting of one of the following accompanies each value:
  - the heading specified by the HEAD= option in a DEFINE(ITEM) statement,
  - the heading taken from the dictionary, or
  - the associated data item name in the list register.
- Each value is displayed in a field whose length is the greater of the data item size or the heading length.
- A single blank character separates each value field. If a field cannot fit on the current display line, then the field begins on a new line.

TABLE	Headings are displayed only at the start of each new page in the information display. Without this modifier, headings are displayed each time the DISPLAY statement is executed.
display-list	The display list contains one or more display fields and their formatting parameters. Several fields can be displayed. The

fields and their formatting parameters are separated by commas; the field/format-parameter combinations are separated from each other by colons, as shown in the following general format:

```
display-field[,format-parameter]...
   [:display-field,[format-parameter]...]...;
```

If you omit *display-list*, the display is formatted as described under "none" and "TABLE".

display field The following options are possible for display fields:

- A reference to a data item name in the list register;
- A child item name whose parent item is in the list register; or
- A character string delimited by quotation marks.

If the requested item cannot be found in the list register, then the processor generates an error at execution time.

Five system variables can also be used as display fields:

$CPU	displays the cumulative amount of CPU time used by the Transact processor for the program, in milliseconds.
$DATELINE	displays the current date and time in the form Fri, Jul 16, 1983, 3:07 P.M.
$PAGE	displays the current page number.
$TIME	displays the current time; the default format is HH:MM AA (e.g., 03:07 PM).
$TODAY	displays the current date; the default format is MM/DD/YY (e.g., 07/16/83).

format-parameters One or more of the following formatting parameters can follow the display field name:

CCTL=*number* Issue a carriage control code of *number* (decimal representation) for the display line containing the associated display field. Carriage control codes are found in the *MPE Intrinsics Manual*.

CENTER Center a display field on a line. The entire field, including leading or trailing blanks, is centered.

COL=*number*	Start the display field in the absolute column position specified by *number*. The first column position is 1. If the display is already at a column position greater than the line width of the display device, the field is not displayed.
EDIT= "*edit-string*"	Characters that designate edit masks.

The following characters have special edit mask meanings for all *display fields* except system variables $TIME and $TODAY (all other characters are treated as insert characters):

^	Insert the character from the source data field into this position in the display field.
Z	Suppress leading zeros. Note that you must use an uppercase Z.
$	Use floating dollar signs.
*	Fill field with leading asterisks.
.	Align the implied decimal point as specified in the dictionary or in a DEFINE(ITEM) definition statement with the decimal point in the edit mask.
!	Ignore the implied decimal point and replace this character with a decimal point.

To denote a negative value with a trailing minus sign, use the minus sign as the final character of the edit string. To denote negative values with a trailing "DR" or "CR", add "CR" or "DR" to the edit string. Some edit-string examples:

Number	Edit String	Result
1234	$$,$$$!^^	$12.34
123456	$$,$$$!^^	$1,234.56
123456	***,**$!^^	*$1,234.56
000009	ZZZZ!^^	.09
-123456	$$,$$$!^^CR	$1,234.56CR
230479	^^/^^/^^	23/04/79

System variables (except $DATELINE) can also be edited. The edit mask characters just defined can be used for $CPU and $PAGE. Special editing characters are used for $TIME and $TODAY.

For $TIME, characters in the edit-mask string are processed as follows:

H	Displays hour with no leading blank or zero if hour < 10.
ZH	Displays hour with leading blank if hour < 10.
HH	Displays hour with leading zero if hour < 10.
24	Displays hour as expressed on a 24-hour clock; used as a prefix to H.
M	Displays minute with no leading blank or zero if minute < 10.
ZM	Displays minute with leading blank if minute < 10.
MM	Displays minute with leading zero if minute < 10.
S	Displays second with no leading blank or zero if second < 10.
ZS	Displays second with leading blank if second < 10.
SS	Displays second with leading zero if second < 10.
T	Displays tenth of a second.
A	Displays the next letter in the AM or PM sequence in uppercase.
a	Displays the next letter in the AM or PM sequence in lowercase.

Except for "a", all other $TIME edit mask characters must be in uppercase. All characters other than edit mask characters are inserted on a character by character basis.

Here are some examples of how edit masks change the format of the $TIME value 3:07:32 PM:

Edit Mask	Displayed Time
HH:MM:SS	03:07:32
24H:M:S	15:7:32
H:MM:SS a.a.	3:07:32 p.m.
ZH:ZM:SS AA	3: 7:32 PM

For $TODAY, characters in the edit mask string are processed as follows:

D	Displays day of the month with no leading blank or zero if day < 10.
ZD	Displays day of the month with leading blank if day < 10.
DD	Displays day of the month with leading zero if day of the month < 10.
DDD	Displays Julian day of year.
M	Displays month with no leading blank or zero if month < 10.
ZM	Displays month with leading blank if month < 10.
MM	Displays month with leading zero if month < 10.
nM	Displays the first n letters of month name in uppercase; if n > number of letters in month name, trailing blanks are not inserted.
nm	Displays the first n letters of month name in lowercase except for the first letter, which appears in uppercase.
YY	Displays last two digits in current year.
YYYY	Displays current year.
nW	Displays first n letters of day of week in uppercase; if n > length of the week name, no trailing blanks are inserted.
nw	Displays first n letters of day of week in lowercase except for the first letter, which appears in uppercase.

All edit string characters must be in uppercase, except for "m" and "w". All characters not defined as an edit string character are inserted on a character by character basis.

Various edit masks applied to the $TODAY date July 16, 1982, make it appear as follows:

Edit Mask	Displayed Date
3w. 3m DD, YYYY	Fri. Jul 16, 1982
DD 3M, YY	16 JUL, 82
M-DD-YY	7-16-82
MM/DD/YY	07/16/82
DDD, YYYY	197, 1982

HEAD= "*character-string*"	Use the *character-string* as a heading rather than the default, which is the heading from the dictionary, the heading from DEFINE(ITEM), or the item or system variable name.
JOIN[=*number*]	Place this number of spaces between the last non-blank character of the current line and the first character of the current display field. To concatenate the character strings, use JOIN=0. Default = 1.
LEFT	Left justify the data item value in the display field. This is the default specification.
LINE[=*number*]	Start the display field on a new line or on a line after a line skip count specified by *number*. If the print device being used can over-print and you want it to do so, you should specify "LINE=0". Default = 1.
LNG=*number*	Truncate the display field to this number of characters. If this option refers to a compound item, then that item is displayed within a display field length of *number*.
NEED=*number*	Print the current line at the top of the next page if there are fewer than the specified number of lines between the current line and the bottom of the page.
NOCRLF	Do not issue a carriage return and line feed for the display line containing the display field.
NOHEAD	Suppress the default heading for this item reference.

NOSIGN	A numeric display field is always positive and no sign position is required in the display field. If a negative value occurs, the display field contains a string of minus signs (-).
PAGE[=*number*]	Start the display field on a new page or on a page after a page skip count specified by *number*. Default = 1.
RIGHT	Right justify the data item value in the display field.
ROW=*number*	Place the display field at absolute line location *number*. The first line position is 1. If the display is already at a line position greater than *number*, then "LINE=1" is in effect.
SPACE[=*number*]	Place this number of spaces between the end of the previous display field and the start of the current display field. To concatenate fields, use SPACE=0. Default=1.
TITLE	Display the associated display field and any preceding display fields only at the start of each new page for which this statement applies.
TRUNCATE	Truncate this display field if it overflows the end of the display line; if field is a numeric type, display pound signs and do not truncate.
ZERO[E]S	Right justify a numeric data value in the display field and insert leading zeros.

EXAMPLES

Assuming the items NAME, ADDRESS, CITY, DISCOUNT, and CUR-BAL have been defined and also specified in a LIST statement, then the following code:

```
DISPLAY NAME, COL=5:
        ADDRESS, SPACE=3:
        CITY, SPACE=5:
        "DISCOUNT RATE IS", LINE=2, COL=5:
        DISCOUNT, NOHEAD:
        "%", JOIN=0:
        "CURRENT BALANCE IS", SPACE=10:
        CUR-BAL, EDIT="$,$$$,$$$.^^", NOHEAD;
```

results in the following display:

```
NAME           ADDRESS                       CITY
 SMITH R.       3304 ROCKY ROAD               COLORADO SPRINGS

DISCOUNT RATE IS 7.5%           CURRENT BALANCE IS $14,734.05
```

The following example illustrates the use of the TABLE modifier and the TITLE option.

```
DISPLAY(TABLE)
   "CUSTOMER LIST", COL=25, TITLE:
   CUST-NO, LINE=2:
   FIRST-NAME, SPACE=3:
   LAST-NAME, JOIN=3:
   STREET-ADDR, SPACE=3:
   CITY, SPACE=3:
   ZIP, SPACE=3;
```

This statement produces a display that prints the title "CUSTOMER LIST" at the start of each page as a result of the TITLE option, and only prints the item heads once on each page as a result of the TABLE modifier.

```
************************************************************************
*                                                                      *
*      {(LEVEL)          }                                             *
*  END[{ system-name     }];                                           *
*      {(SEQUENCE)       }                                             *
*                                                                      *
************************************************************************
```

The function of the END verb depends on the statement parts used.

STATEMENT PARTS

none	At the end of a command sequence: control returns to command level (the current command if the REPEAT qualifier is in effect) or to the beginning of a current level. At the end of a program: issues the message EXIT OR RESTART (E/R)? to which you can respond E to exit from the program or R to restart the program; neccessary only if program branches can cause more than one program end. In either case, the END statement resets the program registers.
LEVEL	The end of the current level. This causes control to fall through the level to the statement following the END(LEVEL) statement and resets the registers to their condition immediately before the level sequence began. If you do not use (LEVEL) in this option, Transact generates a loop after the first execution of the level. The loop begins at the top of the level. The registers are reset to whatever their values were at the beginning of the level. Information on levels is contained in the section that describes the LEVEL verb.
system-name	The end of the executing program (name specified in the SYSTEM statement); necessary if program is one of several included in a text file. The registers are reset.
SEQUENCE	The end of a command sequence; control passes unconditionally back to command level. The registers are reset.

EXAMPLES

```
$$ADD:
 $PROGRAM:
  PROMPT(PATH) PROG-NAME:
               VERSION:
               DESCRIPTION;
  PUT PROGRAMS,
      LIST=(PROG-NAME:DESCRIPTION);
  END;
```

END terminates the command sequence and clears the program registers.

```
SYSTEM PROG1;
 .
 .
 .
END PROG1;
```

This END statement terminates the program PROG1.

```
LEVEL;
 .
 <<process level code>>
 .
END;
```

This END statement terminates processing of the level, resets the program registers to their state before the LEVEL statement, and returns control to the LEVEL statement.

```
LEVEL;
 .
 <<process level code>>
 .
END(LEVEL);
NEXT:
```

This END statement terminates processing of the level, resets the program registers to their state before the LEVEL statement, and passes control to the next statement, in this case, the first statement following the label, NEXT.

Generates an exit from the Transact program to MPE or from a called Transact program to the calling Transact program

```
****************************************************************
*                                                              *
*  EXIT;                                                       *
*                                                              *
****************************************************************
```

EXIT causes control to return to the operating system from the processor if it was processing a main program; if the processor was processing a called program, control returns to the the calling program, where processing continues.

Unlike END, EXIT does not issue the EXIT OR RESTART (E/R)? prompt.

FILE

Reads, writes, updates, sorts, and otherwise operates on MPE files

```
***************************************************************
*                                                             *
*  FILE(modifier) file-name[,option-list];                    *
*                                                             *
***************************************************************
```

NOTE: Several FILE operations can be performed by other Transact verbs.

	For:	Use:
	FILE(CLOSE)	CLOSE
	FILE(READ)	GET or FIND
	FILE(UPDATE)	UPDATE
	FILE(WRITE)	PUT

The Transact verbs in the right column are more general; they apply to KSAM files and IMAGE data sets as well as to MPE files. They also provide more options, but they are not as efficient as the FILE verb for simple MPE file operations.

FILE specifies operations on any MPE file defined in the SYSTEM statement. The operations that FILE performs are determined by the following verb modifiers:

- CLOSE   Closes the specified file (see Syntax Option 1)
- CONTROL   Performs an FCONTROL operation (see Syntax Option 2)
- OPEN   Opens specified file (see Syntax Option 3)
- READ   Reads record from specified file (see Syntax Option 4)
- SORT   Sorts specified file (see Syntax Option 5)
- UPDATE   Replaces current record in specified file (see Syntax Option 6)
- WRITE   Writes record to specified file (see Syntax Option 7)

STATEMENT PARTS

modifier	For the meaning of particular modifiers, see the syntax options below.
file-name	The name of the file as defined in the SYSTEM statement, including the back-reference indicator (*) if applicable. A file is opened automatically the first time it is referenced.
option-list	The allowed options for *option-list* are unique to each syntax option.

SYNTAX OPTIONS

(1) FILE(CLOSE) *file-name*;

FILE(CLOSE) closes the file identified by *file-name*. If $PRINT is specified as the file name, the print file TRANLIST is closed.

(2) FILE(CONTROL) *file-name*,CODE=*number*[,PARM=*item-name*];

FILE(CONTROL) specifies that the FCONTROL operation designated by CODE=*number* is to be performed. The value of *number* must be an unsigned integer. (Refer to the FCONTROL intrinsic description in the *MPE Intrinsics Manual* for the meaning of *number*.)

Any value supplied or returned by the FILE(CONTROL) operation uses the data register field identified by PARM=*item-name*.

FILE(CONTROL) is the only statement that performs the FCONTROL functions on an MPE file.

(3) FILE(OPEN) *file-name*,LIST=(*item-name1*:*item-name2*);

FILE(OPEN) opens the file identified by *file-name*. It is required only with the FILE(SORT) operation. It structures the list register with *item-name1* through *item-name2* for the subsequent sort. This operation is required only if the file already exists and it is to be sorted by the system.

FILE(OPEN) is the only statement that opens an MPE file.

(4) FILE(READ) *file-name*,LIST=(*item-name1*:*item-name2*);

FILE(READ) reads a single record from the file identified by *file-name* and moves the record contents to the portion of the data register corresponding to *item-name1* through *item-name2* in the list register. At the completion of the operation, the status register contains either the number of characters read or -1 to indicate end-of-file.

(5) FILE(SORT) *file-name*

```
          { SORT=(item-name1:item-name2)                            }
          {                                                         };
          { SORT=(item-name1[(ASC)][,item-name2[(ASC)]]...)         }
                             [(DES)]            [(DES)]
```

FILE(SORT) executes the HP/3000 SORT utility to sort an existing file. The sort instruction can consist of (1) a range of items in the order that they are to be sorted (ascending order only), or (2) a list of items or sub-items in the order that they are to be sorted and a specification of ascending (default) or descending order.

Provided that the access mode of SORT is defined for the file, an end-of-file is automatically written into the file before the sort, and the file is rewound following the sort.

FILE(SORT) is the only command that causes a sort on an MPE file. FILE(SORT) requires a preceding FILE(OPEN) with a LIST= option if the file to be sorted has not previously been accessed in the program.

(6) FILE(UPDATE) *file-name*,LIST=(*item-name1*:*item-name2*);

FILE(UPDATE) replaces the current record in the file identified by *file-name*. The record contents are defined by *item-name1* through *item-name2* in the list register.

(7) FILE(WRITE) *file-name*,LIST=(*item-name1*:*item-name2*);

FILE(WRITE) writes a single record to the file identified by *file-name*. The record contents are defined by *item-name1* through *item-name2* in the list register. At the completion of the operation, the status register contains either the number of characters written or -1 to indicate end-of-file.

EXAMPLES

```
SYSTEM TEST,
       BASE=INVTRY,
       FILE=TAPE(WRITE(NEW),80,1,5000),...;
       .
       .
FILE(CONTROL) TAPE,
      CODE=7,
      PARM=LNUM;
       .
       .
```

The FILE(CONTROL) statement causes FCONTROL operation 7 to be performed; that is, it spaces the tape forward to the tapemark. The value it returns is placed in the data register field specified by LNUM. (Refer to *MPE Intrinsics Manual* for more information regarding FCONTROL.)

```
ITEM A X(10):
     B X(20):
     C X(15);
     .
     .
FILE(OPEN) DATAFILE,
     LIST=(A:C);
```

This example maps the data register for a subsequent FILE(SORT).

FIND

Performs multiple retrievals from a file or data set

```
*************************************************************
*                                                           *
*  FIND[(modifier)] file-name[,option-list];                *
*                                                           *
*************************************************************
```

FIND executes multiple retrievals from a file or data set and places retrieved data in the data register. It is usually used with a PERFORM= option to execute a block of statements that processes each record retrieved.

When using the match register to select records, each record is placed in the data register before it is tested for selection against the match register. At the end of a FIND, the area of the data register specified in the LIST= option contains the last record retrieved. This may not be the last record selected.

STATEMENT PARTS

modifier To indicate the type of access to the data set or file, choose one of the following modifiers:

none	Retrieve an entry from an IMAGE master data set based on the key value in the argument register; this option does not use the match register.
CHAIN	Retrieve entries from an IMAGE detail set or a KSAM chain. The entries must meet any match criteria set up in the match register in order to be selected. The contents of the key and argument registers specify the chain in which the retrieval is to occur. If no match criteria are specified, all entries are selected. Match criteria must be included in a LIST= construct.
CURRENT	Retrieve the last entry that was accessed from the file or data set.
DIRECT	Retrieve the entry stored at a specified record number from an MPE or KSAM file or an IMAGE data set. Before using this modifier, you must store the record number as a doubleword integer in the item referenced by the RECNO= option.
PRIMARY	Retrieve the IMAGE master set entry stored at the primary address of a synonym chain. The primary address is located through the key value contained in the argument register.

RCHAIN	Retrieve entries from an IMAGE detail set or a KSAM file chain in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.
RSERIAL	Retrieve entries from a file in the same manner as the SERIAL option, except in reverse order. For a KSAM or MPE file, this operation is identical to SERIAL.
SERIAL	Retrieve entries in serial mode from an MPE or KSAM file or an IMAGE data set that meet any match criteria set up in the match register. If no match criteria are specified, all entries are selected. If match criteria are specified, the match items must be included in a LIST= option of the FIND statement.

file-name The file or data set to be accessed in the retrieval operation. If the data set is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows:

set-name(base-name)

option-list One or more of the following options, separated by commas:

ERROR=*label* ([*item-name*]) Suppress the default error return the processor normally takes. Instead, the program branches to the statement identified by *label*, and the stack pointer for the list register is set to the data item *item-name*. The processor generates an error at execution time if the item cannot be found in the list register.

If you specify no *item-name*, as in ERROR=*label*();, the list register is cleared.

If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not touched.

For more information, see the section entitled "Automatic Error Handling," in section 5.

LIST= (*range-list*) The list of items from the list register to be used for the FIND operation. If the LIST= option is omitted, all the items in the list register are used.

Only the items specified in a LIST= option have their match conditions applied if match conditions are set up in the match register. (The match register may be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.)

Each retrieved entry is placed in the area of the data register indicated by LIST= and matching occurs before any FIND is executed.

The options for *range-list* and the items they cause FIND to access include the following:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current list pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.
(*item-name1*, *item-name2*, ... *item-name*n)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM and MPE, items must be specified in the order of their occurrence in the record. This option incurs some system overhead.
()	A null item list. That is, access the file or data set, but do not retrieve any data.

LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked the whole time that the FIND executes. If LOCK is not specified, the file or data base is locked before each entry is retrieved from the file or data set, remains locked while the entry is processed by any PERFORM= statements, but is unlocked briefly before the next entry is retrieved.
NOMATCH	Ignore any match criteria set up in the match register.
NOMSG	Suppress the standard error message produced by the processor as a result of a file or data base error.
PERFORM=*label*	Execute the code following the specified label for every entry retrieved by FIND. The entries may be optionally

selected by MATCH criteria, in which case control is transferred only for the selected entries.

This option allows operations to be performed on retrieved entries without your having to code loop-control logic.

You may nest up to 10 PERFORM= options.

RECNO=*item-name* With the DIRECT modifier: you must define *item-name* to contain the doubleword integer address of the record to be retrieved.

With other modifiers: Transact returns the record number of the retrieved item in *item-name*.

SINGLE Retrieve only the first selected entry.

SOPT Suppress the processor optimization of IMAGE calls. This option is primarily intended to support a data base operation in a performed routine that is called recursively. The option allows a different path of the same detail data set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of an IMAGE call list of "*" after the first call is made.

```
SORT=(item-name1 [(ASC)][,item-name2 [(ASC)]]...);
                 [(DES)]              [(DES)]
```

FIND sorts on the key items specified in the SORT= option. FIND sorts each occurrence of *item-name1* and, optionally, *item-name2*, and so forth. If the SORT= option does not include any item names, FIND sorts the items named in the LIST= construct. The key items in the SORT= option must also be included in the LIST= option; the items in the LIST= option are the record definition for the sort file.

The FIND statement only sorts if a PERFORM= option is also included, and it always performs the sort before processing the perform statements. The processing sequence for a sort is:

- first retrieve each selected record,
- then write each record to the sort file,
- sort the sort file by any specified items, and
- pass each record one by one to the perform statements.

You may specify ascending or descending sort order. The default is ascending order.

STATUS	Suppress processor actions defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of a FIND statement is described by the value in the status register:

Status Register Value	Meaning
0	The FIND operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with FIND:

- Normal multiple accesses become single.
- The normal rewind done by the FIND is suppressed, so CLOSE should be used before FIND(SERIAL).
- The normal find of the chain head is suppressed, so PATH should be used before FIND(CHAIN).

In the following example of FIND with the status option, normal processing of an error when a broken chain is found is suppressed. The STATUS option enables you to perform a routine to cancel operations until that point.

```
   SET(KEY) LIST(KEY-ITEM);
   PATH DETAIL-SET;
GET-NEXT:
   FIND(CHAIN) DETAIL-SET,STATUS,
   PERFORM=PROCESS-AN-ENTRY;
   IF STATUS=18 THEN       <<BROKEN CHAIN>>
      DO
        PERFORM UNDO-TRANSACTION;
        EXIT;
      DOEND;
   IF STATUS=15 THEN       <<END OF CHAIN>>
      END
   ELSE IF STATUS=0 THEN   <<SUCCESSFUL OPERATION>>
      GO TO GET-NEXT
      ELSE GO TO ERROR-CLEANUP;
```

Were you to avoid the STATUS option, you would set up a procedure to see if a specific entry exists in a chain. When you test the status register, you would get the number of records found.

```
  SET(KEY) LIST(KEY-ITEM);
  SET(MATCH) LIST(DATA-ITEM3);
  FIND(CHAIN) DETAIL-SET,
     LIST=(DATA-ITEM3),SINGLE;
  IF STATUS=0   <<then no entries found>>
        .
        .
        .
```

When the STATUS option is not in effect for a FIND(CHAIN) or FIND(RCHAIN) operation on a detail data set, the status register contains a -1 when the argument value is not in the master data set.

EXAMPLES

The following example uses a PERFORM= option to test data values in each retrieved entry. The routine TEST1 is performed on every record retrieved by FIND(CHAIN).

```
    FIND(CHAIN) DET,
         LIST=(A:H),
         PERFORM=TEST1;
    PERFORM GRAND-TOTAL;
    END;
 TEST1:
    IF (A) = "AUGUST" THEN
       PERFORM PRINT;
    RETURN;
 PRINT:
    LET (SUB) = (SUB) + (AMOUNT);
          .
          .
    DISPLAY ...;
    RETURN;
```

The next example sorts the entries in data set ORDER-DET in primary sequence by ORD-NO and in secondary sequence by PROD-NO. As it sorts, it passes the sorted entries to the PERFORM= statements at the label DISPLAY to be displayed in sorted order.

```
 SORT-FILE:
    LIST ORD-NO:
         PROD-NO:
         DESCRIPTION:
         QTY-ORD:
         SHIP-DATE:

    FIND(SERIAL) ORDER-DET,
       LIST=(ORD-NO:SHIP-DATE),
       SORT=(ORD-NO,PROD-NO),
       PERFORM=DISPLAY;
          .
          .
 DISPLAY:
    DISPLAY "Order List by Product Number", LINE=2:
            ORD-NO, NOHEAD, COL=5:
            PROD-NO, NOHEAD, COL=20:
            QTY-ORD, NOHEAD, COL=35:
            SHIP-DATE, NOHEAD, COL=50;
```

The following example illustrates a method for traversing a pair of IMAGE data sets organized in a tree structure. It uses a recursive routines; that is, the routine NEXT calls itself.

Assume the data base TREE has the following structure:

```
       TREE-MASTER
        __________
        \ PARENT /
         \      /
          \ A  /
           \  / \
            \/   \
                  \
                TREE-DETAIL
                ______________
                \PARENT X(4) /
                 \CHILD X(4)/
                  \        /
                   \______/
```

```
    LIST PARENT: CHILD;
    DATA PARENT;
    MOVE (CHILD) = (PARENT);   <<Initially parent and child must have  >>
                               << value entered by user                >>
    PERFORM NEXT;
    DISPLAY "Tree Traversal Complete";
    EXIT

  NEXT:
    MOVE (PARENT) = (CHILD)    <<child item at this level becomes      >>
                               <<parent at next level                  >>
    SET(KEY) LIST(PARENT);     <<PARENT is key to search for next level>>
    DISPLAY;

    FIND(CHAIN) TREE-DETAIL,   <<Find next level in tree and retrieve  >>
        LIST=(CHILD),          <<child (future parent), then call this >>
        PERFORM=NEXT,          <<routine again until there are no more >>
        SOPT;                  <<child chains.  SOPT is needed to allow>>
                               <<a different path at each level of the >>
                               <<recursion.                            >>

    DISPLAY;
  RETURN;
```

When you use a PERFORM= option in a FIND (or any other file access statement that allows this option), and execute other file access statements within the PERFORM= routine, Transact creates a chain of key/argument registers to keep track of which chain you are following. Each time the program returns from a PERFORM= routine, one set of key/argument values is removed.

For example:

```
    LIST PROD-NUM:
         PROD-CODE:
         DESCRIPTION;
    DATA(KEY) PROD-NUM;                  <<set up 1st key/argument pair >>
    FIND(CHAIN) PROD-DETAIL,
         LIST=(PROD-NUM:DESCRIPTION),
         SORT=(PROD-NUM,PROD-CODE),
         PERFORM=TESTIT;
  EXIT;

  TESTIT:
    DISPLAY "In TESTIT routine";
    DATA(KEY) PROD-NUM;                 <<set up 2nd key/argument pair >>
    FIND(CHAIN) PROD-DETAIL,
         LIST=(PROD-NUM:DESCRIPTION);
    DISPLAY;
  RETURN;
```

FORMAT

Specifies the format of information displayed by the OUTPUT verb or by an unformatted DISPLAY verb

```
****************************************************************
*                                                              *
*  FORMAT display-list;                                        *
*                                                              *
****************************************************************
```

FORMAT specifies the format of a display and the inclusion of any character strings to enhance the display. You use it in conjunction with the OUTPUT verb or an unformatted DISPLAY verb. Use the FORMAT/OUTPUT statement combination when you want to generate a display from more than one entry in a particular data set or file.

The FORMAT statement must precede the DISPLAY or OUTPUT statement it formats. A FORMAT statement in PERFORM procedure associated with an OUTPUT statement does not format that OUTPUT, though it may format another OUTPUT or DISPLAY statement within the PERFORM= procedure.

The specifications in a FORMAT statement are used by the next OUTPUT statement or by the next unformatted DISPLAY statement. The FORMAT specifications cannot be reused unless program control passes through that FORMAT statement again. Format specifications are reset to default values after each FORMAT statement is used by the OUTPUT or DISPLAY statement.

The default format is:

- Display values in order they appear in data register.
- Accompany each value with a heading consisting of:
  - the heading specified for that value in a HEAD= option of a DEFINE(ITEM) statement,
  - the heading taken from the dictionary definition of the item, or
  - the associated data item name in the list register.
- Each value is displayed in a field whose length is either the data item size or the heading length, whichever is longer.
- A single blank character separates each value field. If a field cannot fit on the current display line, then the field begins on a new line.

STATEMENT PARTS

display-list The display list contains one or more display fields and their formatting parameters. Several fields can be displayed. The fields and their formatting parameters are separated by commas; the field/format-parameter combinations are separated from each other by colons, as shown in the following general format:

```
display-field[,format-parameter]...
   [:display-field,[format-parameter]...]...
```

If you omit *display-list*, the display is formatted according to the default format described above

display field The following options are possible for display fields:

- A reference to a data item name in the list register;
- A child item name whose parent item is in the list register; or
- A character string delimited by quotation marks.

If the requested item cannot be found in the list register, then the processor generates an error at execution time.

Five system variables can also be used as display fields:

$CPU	displays the cumulative amount of CPU time used by the Transact processor for the program, in milliseconds.
$DATELINE	displays the current date and time in the form Fri, Jul 16, 1983, 3:07 P.M.
$PAGE	displays the current page number.
$TIME	displays the current time; the default format is HH:MM AA (e.g., 03:07 PM).
$TODAY	displays the current date; the default format is MM/DD/YY (e.g., 07/16/83).

format-parameters	One or more of the following formatting parameters can follow the display field name:

CCTL=*number*	Issue a carriage control code of *number* (decimal representation) for the display line containing the associated display field. Carriage control codes (octal representation) are found in the *MPE Intrinsics Manual*.
CENTER	Center a display field on a line. The entire field, including leading or trailing blanks, is centered.
COL=*number*	Start the display field in the absolute column position specified by *number*. The first column position is 1. If the display is already at a column position greater than the line width of the display device, the field is truncated if it is a character field or pound signs are displayed for a numeric field. If no part of the field fits, it is not displayed.
EDIT="*edit-string*"	Characters that designate edit masks. The following characters have special edit mask meanings (all other characters are treated as insert characters):

^	Insert the character from the source data field into this position in the display field.
Z	Suppress leading zeros. (Z must be uppercase.)
$	Use floating dollar signs.
*	Fill field with leading asterisks.
.	Align the implied decimal point as specified in the dictionary or in a DEFINE(ITEM) definition statement with the decimal point in the edit mask.
!	Ignore the implied decimal point and replace this character with a decimal point.

To denote a negative value with a trailing minus sign, use the minus sign as the final character of the edit string. To denote negative values with a trailing "DR" or "CR", add "CR" or "DR" to the edit string. Some edit-string examples:

Number	Edit String	Result
1234	\$\$,\$\$\$!^^	\$12.34
123456	\$\$,\$\$\$!^^	\$1,234.56
123456	***,**\$!^^	*\$1,234.56
000009	ZZZZ!^^	.09
-123456	\$\$,\$\$\$!^^CR	\$1,234.56CR
-123456	Z,ZZZ!^^-	1,234.56-
230479	^^/^^/^^	23/04/79

System variables (except $DATELINE) can also be edited. The edit mask characters just defined can be used for $CPU and $PAGE. Special editing characters are used for $TIME and $TODAY.

For $TIME, characters in the edit-mask string are processed as follows:

H	Displays hour with no leading blank or zero if hour < 10.
ZH	Displays hour with leading blank if hour < 10.
HH	Displays hour with leading zero if hour < 10.
24	Displays hour as expressed on a 24-hour clock; used as a prefix to H.
M	Displays minute with no leading blank or zero if minute < 10.
ZM	Displays minute with leading blank if minute < 10.
MM	Displays minute with leading zero if minute < 10.
S	Displays second with no leading blank or zero if second < 10.
ZS	Displays second with leading blank if second < 10.

SS	Displays second with leading zero if second < 10.
T	Displays tenth of a second.
A	Displays the next letter in the AM or PM sequence in uppercase.
a	Displays the next letter in the AM or PM sequence in lowercase.

Except for "a", all other $TIME edit mask characters must be in uppercase. All characters other than edit mask characters are inserted on a character by character basis.

Here are some examples of how edit masks change the format of the $TIME value 3:07:32 PM:

Edit Mask	Displayed Time
HH:MM:SS	03:07:32
24H:M:S	15:7:32
H:MM:SS a.a.	3:07:32 p.m.
ZH:ZM:SS AA	3: 7:32 PM

For $TODAY, characters in the edit mask string are processed as follows:

D	Displays day of the month with no leading blank or zero if day < 10.
ZD	Displays day of the month with leading blank if day < 10.
DD	Displays day of the month with leading zero if day of the month < 10.
DDD	Displays Julian day of year.
M	Displays month with no leading blank or zero if month < 10.
ZM	Displays month with leading blank if month < 10.
MM	Displays month with leading zero if month < 10.
nM	Displays the first n letters of month name in uppercase; if n > number of letters in month name, trailing blanks are not inserted.

nm	Displays the first n letters of month name in lowercase except for the first letter, which appears in uppercase.
YY	Displays last two digits in current year.
YYYY	Displays current year.
nW	Displays first n letters of day of week in uppercase; if n > length of the week name, no trailing blanks are inserted.
nw	Displays first n letters of day of week in lowercase except for the first letter, which appears in uppercase.

All edit string characters must be in uppercase, except for "m" and "w". All characters not defined as an edit string character are inserted on a character by character basis.

Various edit masks applied to the $TODAY date July 16, 1982, make it appear as follows:

Edit Mask	Displayed Date
3w. 3m DD, YYYY	Fri. Jul 16, 1982
DD 3M, YY	16 JUL, 82
M-DD-YY	7-16-82
MM/DD/YY	07/16/82
DDD, YYYY	197, 1982

HEAD= "*character-string*"	Use the *character-string* as a heading rather than the default, which is the heading from the dictionary, the heading from DEFINE(ITEM), or the item or system variable name.
JOIN[=*number*]	Place this number of spaces between the last non-blank character of the current line and the first character of the current display field. To concatenate the character strings, use JOIN=0. Default = 1.
LEFT	Left justify the data item value in the display field. This is the default specification.

LINE[=*number*]	Start the display field on a new line or on a line after a line skip count specified by *number*. If the print device being used can over-print and you want it to do so, you should specify "LINE=0". Default = 1.
LNG=*number*	Truncate the display field to this number of characters. If this option refers to a compound item, then that item is displayed within a display field length of *number*; if necessary, new lines are generated.
NEED=*number*	Print the current line at the top of the next page if there are fewer than the specified number of lines between the current line and the bottom of the page.
NOCRLF	Do not issue a carriage return and line feed for the display line containing the display field.
NOHEAD	Suppress the default heading for this item reference.
NOSIGN	Allow no sign position in the display field. If a negative value occurs, the display field contains a string of minus signs (-).
PAGE[=*number*]	Start the display field on a new page or on a page after a page skip count specified by *number*. Default = 1.
RIGHT	Right justify the data item value in the display field.
ROW=*number*	Place the display field at absolute line location *number*. The first line position is 1. If the display is already at a line position greater than *number*, then the display goes to line 1.
SPACE[=*number*]	Place this number of spaces between the end of the previous display field and the start of the current display field. To concatenate fields, use SPACE=0. Default=1.
TITLE	Display the associated display field and any preceding display fields only at the start of each new page for which this statement applies.
TRUNCATE	Truncate this display field if a character field overflows the end of the display line; display pound signs if field is numeric.
ZERO[E]S	Right justify a numeric data value in the display field and insert leading zeros.

EXAMPLES

The following example uses an OUTPUT statement to retrieve information from a data set DETAIL and then display it in a format set up by the preceding FORMAT statement. All headings are suppressed by the first SET(OPTION) statement, rather than by NOHEAD options for individual items. The final RESET(OPTION) statement resets the NOHEAD option for subsequent displays.

```
SET(OPTION) NOHEAD;
FORMAT "Mailing List:",COL=15:
       " ",LINE=3,TITLE:
       FIRST-NAME,COL=5,LINE:
       ADDRESS,COL=5,LINE:
       CITY,COL=5,LINE:
       ",",JOIN=0:
       STATE:
       ZIP,COL=30;
OUTPUT(SERIAL) DETAIL;
RESET(OPTION) NOHEAD;
```

This code produces the following:

```
              Mailing List:



    Harry Swartz
    1 Main St.
    Anywhere, CA                 12345
```

GET

Moves data to the data register from a data set, file, or formatted screen

```
***************************************************************
*                                                             *
*  GET[(modifier)] source[,option-list];                      *
*                                                             *
***************************************************************
```

GET retrieves a single entry from an IMAGE data set or a KSAM or MPE file.

It is also used to move data values into the data register from a terminal under the control of a VPLUS screen.

STATEMENT PARTS

modifier	To specify the type of access to the data set or file, choose one of the following modifiers:
none	Retrieve a master set entry based on the value in the argument register; this option does not use the match register.
CHAIN	Retrieve an entry from an IMAGE detail set or KSAM chain. It retrieves the first entry to meet any match criteria set up in the match register. The matching items must be included in a LIST= option. The contents of the key and argument registers specify the chain in which the retrieval occurs. If no match criteria are specified, it retrieves the first entry in the chain. If no matching entry is found, GET issues a run-time error.
CURRENT	Retrieve the last entry that was accessed from the MPE or KSAM file or IMAGE data set.
DIRECT	Retrieve the entry stored at a specified record number in an MPE or KSAM file, or an IMAGE detail or master set. Before using this modifier, you must store the record number as a doubleword integer in the item specified in the RECNO= option.
FORM	GET(FORM) displays a VPLUS form at the user's terminal and then waits for the user to press ENTER in order to transfer data from the form to the data register. If the user presses a function key instead of ENTER, no data is transferred, unless the AUTOREAD option is used.
KEY	Execute a calculated access on an IMAGE master data set using the key and argument register contents, but transfer

no data. The LIST= option may not be specified with this modifier. (Use GET with no modifier for a calculated retrieval from a master data set.)

This modifier is most useful when you combine it with the ERROR and/or NOFIND options to check for the existence of a key value in a master data set. It allows programmatic control of the result of the checking; it is the equivalent of a CHECK or CHECKNOT on a PROMPT statement.

PRIMARY	Retrieve the IMAGE master set entry stored at the primary address of a synonym chain. The primary address is located through the key value contained in the argument register.
RCHAIN	Retrieve an entry from an IMAGE detail set or a KSAM chain in the same manner as the CHAIN option, only in reverse order. For a KSAM file this operation is identical to CHAIN.
SERIAL	Retrieve an entry in serial mode from an MPE or KSAM file or an IMAGE detail or master set. It retrieves the first entry that matches any match criteria set up in the match register. If no match criteria are specified, it retrieves the first entry in the file or data set. The match items must be included in a LIST= option. If no entry matches or if the file is empty, GET issues a run-time error.
RSERIAL	Retrieve an entry from an MPE or KSAM file or an IMAGE data set in the same manner as the SERIAL option, except in reverse order. For a KSAM or MPE file, this operation is identical to SERIAL.

source The file, data set, or form to be accessed in the retrieval operation. If the data set is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows: *set-name*(*base-name*)

For GET(FORM) only, *source* may be specified as any of the following:

form-name	Name of the form to be displayed by GET(FORM).
(*item-name*)	Name of an item that contains the name of the form to be displayed by GET(FORM).
*	Display the form identified by the "current" form name; that is, the form name most recently specified in a statement that references VPLUS forms. Note that this option is not the same as the CURRENT option (described

under *option-list*), which indicates the currently displayed form.

&	Display the form identified as the "next" form name; that is the form name defined as "NEXT FORM" in the FORMSPEC definition of the current form, where current form means the form name most recently specified in a statement that references VPLUS forms.

option-list The LIST option is available with or without the FORM modifier. Other options, described below, are restricted for use as specified.

LIST= (*range-list*) The list of items from the list register to be used for the GET operation. (Note that this option must not be included with a GET(KEY) operation.)

Only the items specified in a LIST= option have their match conditions applied if match conditions are set up in the match register. (The match register may be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.)

If the LIST= option is omitted with any modifier except FORM, the current contents of the list register are assumed. If the LIST= option is omitted for GET(FORM), the list of items specified for the form in the SYSTEM statement or the dictionary and which appear in the list register is assumed.

The options for *range-list* and the items they cause GET to retrieve are:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current list register pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.

(*item-name1*, *item-name2*, ... *item-name*n)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM, VPLUS, and MPE, items must be specified in the order of their occurrence in the record or form. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
()	A null item list. That is, access the file or data set, but do not retrieve any data.

OPTIONS AVAILABLE WITHOUT THE FORM MODIFIER

ERROR=*label* ([*item-name*])	Suppress the default error return that the processor normally takes. Instead, branch to the statement identified by *label*, and set the stack pointer for the list register to the data item *item-name*. The processor generates an error at execution time if the item cannot be found in the list register. If you specify no *item-name*, as in ERROR=*label*();, the list register is reset to empty. If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not changed. For more information, see "Automatic Error Handling," in section 5.
LOCK	Lock the specified file or data base unconditionally. If you are accessing a data set, the LOCK option causes Transact to lock the entire data base while the GET executes. If LOCK is omitted, Transact unlocks the data base after each IMAGE call made by GET.
NOFIND	Ensure that a matching entry is not present in the referenced IMAGE master data set. If such an entry is found, an error message is generated. If the STATUS option has also been specified, the code returned in the STATUS register for the error condition is 1, meaning that a record was found.
NOMATCH	Ignore any match criteria set up in the match register.
NOMSG	Suppress the standard error message produced by the processor as a result of a file or data base error; all other error actions occur.

RECNO=*item-name*	With the DIRECT modifier: you must define *item-name* to contain the doubleword integer number (I(9,,4)) of the record to be retrieved. With other modifiers: Transact returns the record number of the retrieved record in the doubleword integer *item-name*.
STATUS	Suppress processor action defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option. When STATUS is specified, the effect of a GET statement is described by the value in the status register:

Status Register Value	Meaning
0	The GET operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with GET:

- The normal rewind done by the GET is suppressed, so CLOSE should be used before GET(SERIAL).
- The normal find of the chain head by the GET is suppressed, so PATH should be used before GET(CHAIN).

In the following example, GET with the STATUS option allows you to process the "nonexistent permanent file" error yourself. This coding lets you access a file that may be in another account by setting up a file equation through a PROC call to the command intrinsic.

```
 <<1st access, no CLOSE required before SERIAL operation>>

GET(SERIAL) DATA-FILE,
    LIST=(A:N),
    STATUS;
IF STATUS <> 0 THEN   <<An error occurred, check further>>
  IF STATUS <> 52 THEN    <<error is other than expected>>
    GO TO ERROR-CLEANUP
  ELSE                     <<52 - nonexistent permanent file>>
   DO
    LET (CR) = 8205;      <<8205 = space,carriage return>>
    <<could have used (CR)=3360 for carriage return,space>>
    MOVE (COM-STRING) =
      "FILE DATAFILE=DATAFILE.PUB.OTHERONE"+(CR);

    <<Try opening DATAFILE in another group>>

    PROC COMMAND (%(COM-STRING),(ERROR),(PARM));
    IF (ERROR) <> 0 THEN     <<command error>>
      GO TO ERROR-CLEANUP;

    <<Try again, give up if unsuccessful>>

    GET(SERIAL) DATA-FILE,
        LIST=(A:N),
    STATUS;
    IF STATUS<>0 THEN
      GO TO ERROR-CLEANUP;
   DOEND;
```

OPTIONS AVAILABLE ONLY WITH THE FORM MODIFIER

APPEND	Append the next form to the form specified in this statement, overriding any freeze or append condition specified for the form in its FORMSPEC definition. APPEND sets the FREEZAPP field of the VPLUS comarea to 1.
AUTOREAD	Accept any function key not specified in an F*n*=*label* option in order to transfer data from the form to the data register. If a key has been specified in an F*n*=*label* option, then GET does not execute AUTOREAD for that key.
CLEAR	Clear the specified form when the next form is displayed, overriding any freeze or append condition specified for the form in its FORMSPEC definition. CLEAR sets the FREEZAPP field of the VPLUS comarea to zero.
CURRENT	Use the form currently displayed on the terminal screen; that is, perform all the GET(FORM) processing except retrieving and displaying the form. Use this option to avoid the processing that normally occurs when a new form is displayed.
FEDIT	Perform the field edits defined in the FORMSPEC definition immediately before displaying the form.
FKEY=*item-name*	Move the number of the function key the operator presses in this retrieval operation to the single-word integer (I(4)) *item-name*. The function key is determined by the contents of the field LAST-KEY in the VPLUS comarea. It may have a value of 0 through 8, inclusive, where 0 indicates the ENTER key and 1 through 8 indicate function keys 1 through 8, respectively.
F*n*[(AUTOREAD)]= *label*	Control passes to the labelled statement if the operator presses function key *n*. This option may be repeated for each desired function key as many times as necessary in a single GET(FORM) statement. If (AUTOREAD) is included, then transfer the data from the form to the data register before going to the specified label. F0, or ENTER, always transfers data.
FREEZE	Freeze the specified form and append the next form to the specified form, overriding any freeze or append conditions specified for the form in the FORMSPEC definition. FREEZE sets the FREEZAPP field of the VPLUS comarea to 2.
INIT	Initialize the fields in a VPLUS form to any initial values specified for the form by FORMSPEC, or perform any Init

	Phase processing specified for the form by FORMSPEC. The INIT processing is performed before the form is displayed on the screen.
STATUS	Suppress the display of VPLUS field edit error messages in window; Transact conversion messages are sent to the window. Transfer control immediately back to the program after the user has pressed ENTER or the appropriate function key. If field edit errors exist, Transact sets the value of the processor status field to a negative count of the number of errors (given by the NUMERRS field of the VPLUS comarea). Otherwise, the value in the status field is 0.
WINDOW= ([*field*,] *message*)	Place a message in the window area of the screen and, optionally, enhance a field in the form. The fields *field* and *message* can be specified as follows:
field	Either the name of the field to be enhanced, or an *item-name* within parentheses which will contain the name of the field to be enhanced at run time.
message	Either a *string* enclosed in quotation marks that specifies the message to be displayed, or an *item-name* within parentheses containing the message string to be displayed in the window.

The following example illustrates this option when the field name and the message are specified directly.

```
GET(FORM) FORM1,
   INIT,
   LIST=()
   WINDOW=(field1,"This field must be numeric");
```

In the following example, both the field and the message are specified through an item-name reference:

```
DEFINE(ITEM) ENHANCE U(16):
             MESSAGE U(72);

MOVE (ENHANCE) = "field1";
MOVE (MESSAGE) = "This field must be numeric.";
        .
        .
GET(FORM) *,
   INIT,
   WINDOW=((ENHANCE),(MESSAGE));
```

EXAMPLES

```
PROMPT(PATH) CUST-NO;
LIST CUST-NAME:
     CUST-PHONE;
GET(CHAIN) DETAIL,
     LIST=(CUST-NAME:CUST-PHONE);
```

The first entry in the chain is retrieved from the data set DETAIL using the items CUST-NAME through CUST-PHONE in the list register.

```
PROMPT(PATH) CUST-ID;
LIST CUST-NAME:
     CIST-PHONE;
GET(RCHAIN) DETAIL,
     LIST=(CUST-NAME:CUST-PHONE);
```

```
DATA(PATH) CUST-ID;
DATA(MATCH) CUST-NAME;
GET(RCHAIN) DETAIL,
     LIST=(CUST-NAME:CUST-PHONE);
```

The first GET retrieves the last record in the chain. The second GET reads the chain in reverse order until a record matches the criteria set up by the PROMPT(MATCH) statement.

```
GET(FORM) CUSTFORM,
   INIT,
   LIST= (CUST-NAME, CUST-ADDR, CUST-PHONE);
```

This statement displays the form CUSTFORM, performs any initialization specified by FORMSPEC, retrieves values entered into the form, performs any FORMSPEC edits, and then transfers the entered values to the data register areas associated with the specified list items.

The following example illustrates a method for "structured programming" with VPLUS forms:

```
 START:
   DISPLAY "Start of program";
   PERFORM GETINFO;
   DISPLAY "End of program";
   EXIT;

 GETINFO:
   GET(FORM) MENU,
     F1=ADD,
     F2=UPDATE,
     F3=DELETE,
     F4=LIST,
     F5=START,
     F6=START,
     F7=START,
     F8=ENDIT;
   <<Process ENTER here>>
     .
     .
     .
 ADD:
   <<process F1 here>>
   RETURN;
 UPDATE:
   <<process F2 here>>
   RETURN;
 DELETE:
   <<process F3 here>>
   RETURN;
 LIST:
   <<process F4 here>>
   RETURN;
 ENDIT:
   EXIT;
```

An alternate method is to use the FKEY=*item-name* construct, and then test the value of *item-name* with an IF statement.

Transfers control to a labelled Transact statement

```
*****************************************************************
*                                                               *
*  GO TO label;                                                 *
*                                                               *
*****************************************************************
```

GO TO specifies an unconditional branch to the statement identified by *label*.

STATEMENT PARTS

label	The label to which the program should branch.

EXAMPLE

```
   GO TO NEW-TOTAL;
```

This statement transfers control to the statement labelled "NEW-TOTAL".

IF

Performs a specified action based on a conditional test

```
*****************************************************************
*                                                               *
*  IF condition-clause THEN statement [ELSE statement];         *
*                                                               *
*****************************************************************
```

IF specifies a test to be performed on a *test-variable*. The test is the *condition-clause*; it contains the *test-variable*, the *relational-operator*, and one or more *values*. If the condition is true, then the specified statement is performed. You may provide an alternate statement to be performed if the condition is not true by including the ELSE clause. If you do not include an ELSE clause and the condition is not true, control passes to the statement following the IF statement.

NOTE: Do not terminate the statement preceding the ELSE clause with a semicolon (;).

STATEMENT PARTS

condition-clause — A *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value [,*value*]...

test-variable — May be one or more of the following:

(*item-name*)	The value in the data register that corresponds to *item-name*.
EXCLAMATION	Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. If the null response is set, the EXCLAMATION test variable is a positive integer; if not set, it is zero. Default is 0.
FIELD	Current status of FIELD option. If an end user qualifies a command with FIELD, the FIELD test variable is a positive integer; otherwise, it is a negative integer. Default is <0.
INPUT	The last value input in response to the INPUT prompt.
PRINT	Current status of PRINT or TPRINT option. The PRINT test variable is an integer greater than zero and less than 10; if a command is qualified with TPRINT, PRINT is an integer greater than 10; if neither

	qualifier is used, PRINT is a negative integer. Default is <0.
REPEAT	Current status of REPEAT option. If an end user qualifies a command with REPEAT, the REPEAT test variable is a positive integer; otherwise, REPEAT is a negative integer. Default is <0.
SORT	Current status of SORT option. If an end user qualifies a command with SORT, the value of the SORT test variable is a positive integer; otherwise SORT is a negative integer. Default is <0.
STATUS	The value of the status register set by the last data set or file operation, data entry prompt, or external procedure call.

relational-operator Specifies the relation between the *test-variable* and the *value*. It may be one of the following:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

value The value against which the *test-variable* is compared. The allowed value depends on the test variable

If *test-variable* is:	Then *value* must be:
(*item-name*)	An alphanumeric string, a numeric value, or a reference to a variable as in (*item-name*).
INPUT	An alphanumeric string
EXCLAMATION FIELD PRINT REPEAT SORT	A positive or negative integer
STATUS	An integer number

Alphanumeric strings must be enclosed in quotation marks.

If more than one value is given, then:

- The *relational-operator* can be "=" only, and
- The action is taken if the *test-variable* is equal to *value1* OR *value2* OR...*valuen*. In other words, a comma in a series of values is interpreted as an OR.

statement Any simple or compound Transact statement; a compound statement is one or more statements bracketed by a DO...DOEND pair

EXAMPLES

```
IF INPUT = "YES", "Y" THEN
   GO TO PROCEED;
```

This statement causes a program branch to the "PROCEED" label if "YES" or "Y" was input in response to the INPUT prompt. If INPUT contains any other value, control passes to the next statement.

```
IF (COUNT) > 3 THEN
   GO TO TOO-HIGH;
```

This statement causes a program branch to the "TOO-HIGH" label if the data register value for the item-name COUNT is greater than 3.

```
IF STATUS <> 0 THEN END;
```

This statement causes an exit from the current command sequence if the status register value does not equal 0.

```
IF INPUT = "Y" THEN
  DO
   DISPLAY "PART NUMBER IS": PART-NO;
   PERFORM ADD-INFO;
  DOEND
ELSE IF (A) = (B) THEN
       DO
        DISPLAY "DUPLICATE ENTRY";
        PERFORM SAME-PART;
        IF (A) = (C) THEN
          IF (D) < 50 THEN
            MOVE (A) = (D);
       DOEND
     ELSE PERFORM MORE-INFO;
```

The statements within the first DO/DOEND pair execute if the value in the input register is "Y".

Otherwise, if the value for A equals the value for B, the statements at the label SAME-PART are executed.

The value for D is moved to the space reserved for A if:

- INPUT does not equal "Y", and
- A equals B, and
- A equals C, and
- D is less than 50.

The statements at label MORE-INFO are executed if:

- INPUT does not equal "Y", and
- A does not equal B.

INPUT

Prompts for a value and places it in the input register

```
**************************************************************
*                                                            *
*  INPUT "prompt-string"[,option-list];                      *
*                                                            *
**************************************************************
```

INPUT generates a prompt that requests a user response. Usually the value input as a response to *prompt-string* is tested by a subsequent IF statement. Then the response may be used to programmatically change program flow during execution. Transact upshifts all entered values.

The value returned by INPUT cannot be displayed or moved. Thus, INPUT is useful mainly to test a user response. To save or display a user response, you should use another verb, such as DATA or PROMPT, that transfers the response to an item defined in your program.

STATEMENT PARTS

prompt-string	The prompt that appears on the user's terminal; it must be enclosed within quotes.
option-list	One or more of the following options separated by commas:

BLANKS	Do not suppress leading blanks supplied in the input value.
NOECHO	Do not echo the input value to the terminal.
STATUS	Suppress normal processing of "]" or "]]", which causes an escape to a higher processing or command level. Instead, these characters set the status register to -1 or -2, respectively. If the user enters one or more blanks and no non-blank characters, this sets the status register to -3. (The status register normally contains the number of characters entered in response to the prompt, excluding the carriage return.) The STATUS option allows you to control subsequent processing by testing the contents of the register with an IF statement.

EXAMPLES

```
INPUT "DO YOU WISH THE REPORT ON THE LINE PRINTER?";
IF INPUT = "Y", "YES" THEN
  DO
    SET(OPTION) PRINT;
    DISPLAY "LINE PRINTER SELECTED FOR OPTION PRINT";
  DOEND;
```

This example illustrates a typical use of the INPUT verb. INPUT accepts a user response, and then the IF statement tests for a particular value of this response.

ITEM

Defines variables for use in the program that have not been defined in the dictionary. The DEFINE(ITEM) verb is preferred. Refer to the DEFINE(ITEM) description for the syntax option description and additional information.

Specifies arithmetic operations or sets up array manipulation

```
****************************************************************
*                                                              *
* LET destination-variable =arithmetic expression;             *
*                                                              *
****************************************************************
```

Depending on the particular syntax option, LET can:

- Perform arithmetic operations (Syntax Option 1) or
- Aid in array manipulation (Syntax Option 2).

LET, unlike MOVE, checks that the data types of items being assigned are compatible with the item to which they are assigned. If necessary, LET performs type conversion.

STATEMENT PARTS

destination-variable	An item name that identifies a location in the data register, or the processor-defined name of a special purpose register. The result of the operation is placed in this variable. The destination variable may be any of the following:

(*item-name*)	The computed or assigned value of *item-name*. The item name identifies a location in the data register.
LINE	An integer of type I(4) that contains the computed or assigned value of the line counter for the current line of terminal display or line printer output.
OFFSET (*child-item*)	An integer of type I(4) that contains the offset of a child item within its parent item, starting at position zero.
PAGE	An integer of type I(4) that contains the computed or assigned value of the page counter.
PLINE	An integer of type I(4) that contains the computed or assigned value of the line counter for the current line of line printer output.
STATUS	An integer of type I(4) that contains the computed or assigned value of the status register.
TLINE	An integer of type I(4) that contains the computed or assigned value of the line counter for the current line of terminal display output.

arithmetic-expression	A single variable, or multiple variables connected by arithmetic operators in the format:

[-]*variable1* [*operator variable2*]...

[-]	If the expression is preceded by a minus sign, its negative is assigned.
variable1	An item name within parentheses, a numeric constant, or one of the processor-defined names listed above under the description of *destination-variable*.
operator	+ addition - subtraction * multiplication / division giving the quotient // division giving the remainder
variable2	The same as *variable1*.

The item names can identify an item of any type and any decimal scale. All arithmetic operations are performed at the highest scale and the result is rounded to the scale of the receiving item. If it is not possible to convert all operands to the highest scale, then Transact chooses the highest scale that will accommodate all operands; this may result in a loss of least significant digits.

If all items are single or doubleword integers, then all operations are performed in doubleword integer arithmetic; if all are floating point, then long format (double precision floating point) is used. If the items are a mixture of types, or if the desired precision is greater than the machine precision, then all arithmetic operations are performed using the hardware packed decimal instructions.

In multiple operations, the arithmetic is done from left to right, in the following order:

//	division giving remainder (first)
/	division giving quotient
*	multiplication
-	subtraction
+	addition (last)

You can change the order by using square brackets. For example, the following two statements may yield different results:

```
LET (A)=(B) + (C)/(D);
LET (A)=[(B) + (C)]/(D);
```

Note that Transact does not provide a direct means of performing exponentiation, square roots, or logarithmic functions.

SYNTAX OPTIONS

(1) LET (*variable*)=[-]*arithmetic-expression*;

Choose this option to move a single value or the result of an arithmetic operation into a location in the data register (*item-name1*) or into one of the processor-defined names allowed for the destination variable.

The following are examples of this syntax option:

```
LET (TOTAL)=(TOTAL) + (AMOUNT);        <<Add values of AMOUNT and TOTAL>>

LET (PERCENT)=9.8;                     <<Set value of PERCENT to 9.8>>

LET (INVERSE)=1/(DIVISOR);             <<Calculate inverse value>>

LET (COUNT)=-(COUNT);                  <<Negate value of COUNT>>

LET (DEDUCTION)=-[(TOTAL)-(BENEFIT)];  <<Negate TOTAL less BENEFIT>>

LET PAGE=200;                          <<Set page counter to 200>>

LET LINE=60-(REMAINING-LINES);         <<Calculate value of current line>>

LET (STAT) = STATUS;                   <<Set STAT to contents of status
                                         register>>

LET STATUS = STATUS+1;                 <<Increment value of status register>>

LET STATUS = 0;                        <<Clear status register>>
```

(2) LET OFFSET(*child-name*)=[-]*arithmetic-expression*

(*child-name*)	Identifies an item that has been defined as a child item.

This option of the LET verb sets the value of OFFSET for a particular child item. It allows you to refer to a child item within a parent item, or array, by telling the processor the byte number of the parent item at which the child item begins. By changing the value of OFFSET, you may refer to any child item within the parent item.

Suppose an array and its child items are defined as follows:

```
DEFINE(ITEM) SALES 3X(10):
             YEAR X(10)=SALES(1);

            SALES
    ----------------------
    |      |      |      |
    | YEAR | YEAR | YEAR |
    |      |      |      |
    ----------------------
```

Initially, the OFFSET of YEAR within SALES is 0, which actually refers to byte position 1 of SALES. That is, YEAR(1)= SALES(1), and, therefore, YEAR refers to the first 10 bytes of SALES. To refer to other elements of SALES, you must change the OFFSET of YEAR. You may do it as follows:

LET OFFSET(YEAR)=(*element-number* - 1) * *element-size*

where *element-size* is expressed in bytes.

For example, to point to the third element of SALES, which is 10 bytes long, and then move a value to that element, use the following statements:

```
LET OFFSET(YEAR)= 2 * 10;     << (3rd element-1) * element size   >>
MOVE(YEAR)=(VALUE-STRING);
```

To access and display the second and third positions, use the following statements:

```
SYSTEM TEST;
DEFINE(ITEM) SALES 3X(10):
      YEAR X(10)=SALES(1);
PROMPT SALES;
DISPLAY SALES;
DISPLAY YEAR;
LET OFFSET(YEAR)= 1 * 10;      <<Access 2nd element of SALES (2-1) >>
DISPLAY YEAR;
LET OFFSET(YEAR)= 2 * 10;      <<Access 3rd element of SALES (3-1) >>
DISPLAY YEAR;
END;
```

Note that the offset is counted from zero. Thus, to access the second position in SALES, you specify an offset of 1; to access the third position of SALES, you specify an offset of 2.

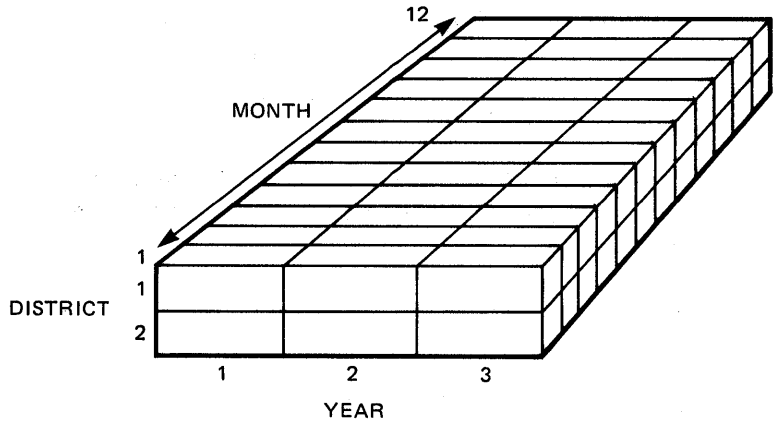
It is possible to step through a parent item using the following form of the LET statement:

LET OFFSET(*child item*)=OFFSET(*child item*) + (*byte-length-of-child item*)

For example, assuming the same array SALES, you can specify the next child item as follows:

```
LET OFFSET(YEAR) = OFFSET(YEAR) + 10
```

You can also use the OFFSET option of LET to manipulate multi-dimensional arrays. Consider the following three-dimensional matrix of sales figures. Its dimensions are district, year, and month. Each cell which is a child item, contains a sales figure in integer format with two decimal places. Note that each value in each cell requires four bytes of storage.



This SALES matrix requires the following DEFINE(ITEM) statement:

```
DEFINE(ITEM) SALES 72I(10,2):
        DIST 36I(10,2) = SALES(1):
        YEAR 12I(10,2) = DIST(1):
        MONTH I(10,2) = YEAR(1);
```

To locate the position of one cell within the matrix, you must use three LET OFFSET statements. To locate the byte position of the second district of the third year of the seventh month, use the following three LET OFFSET statements:

```
LET OFFSET(DIST)= 1 * [36*4];
LET OFFSET(YEAR)= 2 * [12*4];
LET OFFSET(MONTH)= 6 * [1*4];
```

Defines processing levels within a program

```
*************************************************************
*                                                           *
*  LEVEL[(label([item-name]))];                             *
*                                                           *
*************************************************************
```

LEVEL specifies a new processing level. LEVEL allows repeated entries and retention of information during data entry and eliminates redundant data entry operations. Match, update, and list register entries within a level are unique to that level. When an end of level occurs, these registers are reset to their condition upon entering the level.

STATEMENT PARTS

label	The statement to which the program should branch at the end of the level sequence
item-name	The location in the list register where the pointer is to be set.
	If you specify no *item-name*, for example, LEVEL(*label*());, the list register is reset to empty.
	If you use an "*" instead of *item-name*, as in LEVEL(*label*(*));, then the list register is not changed.

EXITS FROM LEVEL SEQUENCES

Four types of exits from LEVEL sequences are possible, two of which the end user controls and two of which you control. They are described below.

(1)]	When the end user enters "]" in response to any prompt in a level sequence, control passes to the next previous processing level, which may be the command level. Any changes made to the match, list, or update registers within the level are cleared.
(2)]]	When the end user enters "]]" in response to any prompt in a level sequence, control passes to Transact command level, or if not in a command sequence, Transact issues the EXIT OR RESTART(E/R)> prompt.
(3) END(LEVEL)	When a level sequence ends with the statement END(LEVEL);, any changes made to the list, match, or update registers within the level are reset, and then control passes to the statement immediately following END LEVEL;.

(4) END	When a level sequence ends with the statement END; current processing of the level ends, any changes made to the list, match, or update registers within the level are cleared, and control returns to the currently active LEVEL statement.

Nested level sequences are possible, as the example illustrates.

EXAMPLES

The following statements minimize the data entry required for a sequence of entries from a time card. It requires values for employee name and year, and multiple entries for day, activity code, and hours:

```
PROMPT YEAR:
       MONTH;
LEVEL;
   PROMPT EMPLOYEE;
   LEVEL;
      PROMPT DAY;
      LEVEL;
         PROMPT ACTIVITY:
                HOURS;
         PUT TIME-RECORD;
END;
```

Execution of these statements causes a prompt for each data item value and then a loop at the lowest level. When the user has entered all activity items for a specific day, he or she should enter a "]" in response to "ACTIVITY". Control passes to the next higher level and the user is prompted with "DAY". When all days have been entered for one employee, then the user should enter "]" in response to "DAY". Then he or she is prompted for the next employee.

Adds item names to list, key, match, and/or update registers

```
*************************************************************
*                                                           *
*  LIST[(modifier)] item-name [,option-list]                *
*                   [:item-name [,option-list]]...;         *
*                                                           *
*************************************************************
```

LIST adds data item names to the list, key, match, and/or update registers. The registers affected depends on the verb modifier. You may choose from the following:

- none Add specified item name to list register, reserve space and, optionally, place value in data register (see Syntax Option 1).
- KEY Place specified item name in key register (see Syntax Option 2).
- MATCH Add specified item name to list register, and copy existing value for that item from the data register to the match register (see Syntax Option 3).
- PATH Add specified item name to list register and place it in key register (see Syntax Option 4).
- UPDATE Add specified item name to list register and copy value for that item from the data register to the update register (see Syntax Option 5).

Consider the following when setting up your list register:

- For use with IMAGE data base access, list items must be consecutive but in any order.
- For use with KSAM or MPE files, list items must be consecutive and in the same order as in the file.
- For use with VPLUS forms, list items may appear in any order and need not be consecutive, although consecutive order allows simpler range lists in the data management statements.
- Child item names may not be specified as list items in a LIST statement; instead, the associated parent item name must be specified.
- System variables cannot be put in a LIST statement; they can only be used in DISPLAY or FORMAT statements.

STATEMENT PARTS

modifier	Change or enhancement to the action of LIST; often the register to which the input value should be added or the register whose value should be changed
item-name	The *item-name* to be added or changed in the list, key, match, or update registers; must not be a child item name.
option-list	Values specific to Syntax Options (1) and (3)

SYNTAX OPTIONS

(1) LIST *item-name*[,*option-list*]

LIST with no modifier adds the *item-name* to the list register and reserves space in the data register. If you do not include an option from the list below, Transact does not change the original contents of the data register. If you choose an option from the list below, it places the corresponding value in the data register.

option-list: Specifies a value to be placed in the data register. Note that the options listed below are not variable names and need not be defined in a DEFINE(ITEM) statement or in a dictionary.

ACCOUNT	An X(8) item that contains the account name from logon.
DATE	An X(6) item that contains the current system date in MMDDYY format. If the data item size is not six characters, then truncation or blank fill occurs. This option is normally used to set up a data item that is to contain the current date.
DATE/D	An X(6) item that contains the current system date in DDMMYY format.
DATE/J	An X(5) item that contains the current system date in Julian YYDDD format.
DATE/L	An X(27) item that contains the current system date/time message.
DATE/Y	An X(6) item that contains the current system date in YYMMDD format.
GROUP	An X(8) item that contains the group name from logon.
HOMEGROUP	An X(8) item that contains the home group of the logged-on user.

INIT[IALIZE]	Blanks if the data item type is an alphanumeric string, or binary zero for all other types
PASSWORD	An X(8) item that contains the first password value entered during Transact/3000 system logon.
PROCTIME	An I(9) item that contains the doubleword integer of session cpu time in milliseconds.
TERMID	An I(4) item that contains the terminal logical device number.
TIME	An X(8) item that contains the current time in HHMMSSTT format.
TIMER	An I(9) item that contains the doubleword integer of system time in milliseconds.
SESSION	An X(1) item than contains an "S" or a "J" to indicate that the current process is running as a session or a job, respectively.
USER	An X(8) item that contains the user name from logon.

For example, the following statements define the item MYPASS, move it to the list register, allocate it space in the data register, and place the user's password in that space:

```
DEFINE(ITEM) MYPASS  X(8);
LIST MYPASS, PASSWORD;
```

(2) LIST(KEY) *item-name*;

LIST(KEY) places *item-name* in the key register only.

(3) LIST(MATCH) *item-name*[,*option-list*];

LIST(MATCH) adds *item-name* to the list register and copies the existing value from the data register into the match register as a selection criterion for subsequent file or data set operations. MATCH is typically used when a previous retrieval operation has placed a value in the data register and that value is now to be used for the next selection criterion. The *item-name* for the new data item list may differ from the *item-name* used for the previous retrieval. When you are remapping the data register, you can initialize the value by using one of the choices from *option-list* shown with Syntax Option 1.

The following values for *option-list* specify a match selection to be performed on a basis other than equality.

option-list:

NE	Not equal to
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
LEADER	Matched item must begin with the input string; equivalent to the use of trailing "^" on input
SCAN	Matched item must contain the input string; equivalent to the use of trailing "^^" on input
TRAILER	Matched item must end with the input string; equivalent to the use of a leading "^" on input

(4) LIST(PATH) *item-name*;

LIST(PATH) adds *item-name* to the list register and places it in the key register.

(5) LIST(UPDATE) *item-name*;

LIST(UPDATE) adds *item-name* to the list register and places the value already in the data register into the update register for a subsequent data set or file operation using the REPLACE verb.

EXAMPLES

The first example places item names NAME, ADDRESS, CITY, and DATE in the list register and reserves areas for their values in the data register. The areas for NAME, ADDRESS, and CITY are initialized to blanks and the area for DATE is initialized to the current system date in MMDDYY format.

```
DEFINE(ITEM) NAME X(20):
             ADDRESS X(20):
             CITY X(10):
             DATE X(6);
LIST NAME,INIT:
     ADDRESS,INIT:
     CITY,INIT:
     DATE,DATE;
```

The data register is your stack; it is never cleared, only mapped and remapped through the list register. To illustrate this point, consider the following example that references two data bases. In one, a customer name is identified by two items, "LAST-NAME" and "FIRST-NAME; in the other, the same name is identified by a single item, "CUST-NAME".

```
SYSTEM TEST1,
       BASE=CUST-BASE,
            PROD-BASE;
DEFINE(ITEM)  LAST-NAME   X(10):
              FIRST-NAME  X(10):
              CUST-NAME   X(20);

LIST LAST-NAME: FIRST-NAME;        <<map data register with LIST statement>>
GET CUST-MAST,
    LIST=(LAST-NAME:FIRST-NAME);   <<retrieve name, move to data register >>

RESET(STACK) LIST;                 <<reset list register to its beginning >>
LIST CUST-NAME;                    <<map same data with new list register >>

PUT CUST-INFO(PROD-BASE),
    LIST=(CUST-NAME);              <<write name to other data base         >>

END TEST1;
```

Note that the list register was reset programmatically with the RESET(STACK) statement.

In the next example, the company code is used to retrieve and display data from one data set (CO-MSTR) and then the same value, renamed by LIST(PATH) as the department code, is used to access another data set (DEPT-MSTR).

```
PROMPT(PATH) COMPANY-CODE,    << get company code for subsequent retrieval >>
     CHECK=CO-MSTR;           << from CO-MSTR data set                     >>
LIST A:
     B:
     C;
OUTPUT CO-MSTR;
RESET(STACK) LIST;
LIST(PATH) DEPT-CODE;         << use same value as department code for    >>
LIST X:                       << subsequent retrieval from DEPT-MSTR      >>
     Y:
     Z;
OUTPUT DEPT-MSTR;
```

In the following example, Transact resets the list register automatically when a new command sequence starts.

Because Transact resets the list register at the start of each new command sequence, you should define any global variables before the first command sequence, and then redefine the global variables within each command sequence preceding any local variables. For example, suppose the variables, "VENDOR-ID" and "VENDOR-NAME" are to be used by both sequences UPDATE PRODUCT and UPDATE VENDOR. Before executing either sequence, you can define these items and place values for them in the data register. In order to retain these values, all you need do is remap the list register at the start of each sequence.

```
   LIST VENDOR-ID:          << map global variables in list reg.  >>
        VENDOR-NAME;
   DATA VENDOR-ID:          << prompt user for data               >>
        VENDOR-NAME;

$$UPDATE:                   << new command sequence -             >>
 $PRODUCT:                  << Transact resets list register      >>
   LIST VENDOR-ID:          << remap global variables             >>
        VENDOR-NAME:
        PROD-NUM:           << variables local to UPDATE PRODUCT  >>
        DESCRIPTION;

 $VENDOR:                   << Transact resets list register again >>
   LIST VENDOR-ID:          << remap global variables             >>
        VENDOR-NAME:
        VENDOR-ADDRESS:     << variables local to UPDATE VENDOR   >>
        VENDOR-ZIP;
```

MOVE

Places data into a specified data register space

```
*************************************************************
*                                                           *
*  MOVE (item-name1)=[-]source-field;                       *
*                                                           *
*************************************************************
```

MOVE places data into the data register location specified by *item-name1*. You should use MOVE particularly when you want to move a character string into a data register location. Unlike LET, MOVE does not check data types during the operation; if it is necessary to check data types between the source and the receiving fields, you should use the LET verb.

If the *source-field* length is not the same as the receiving field length (*item-name1*), the source value is truncated on the right or filled with blanks on the right.

STATEMENT PARTS

(*item-name1*)	Specifies that you want the data moved into the data register location identified by *item-name1*.
source-field	may be one of the following:
[-](*item-name2*)	The value in the data register location for *item-name2*. If you include the "-", then the source value is placed in the destination field with opposite justification. That is, source data that is right justified is left justified in the destination field and vice-versa.
[-]"*character-string*"	A programmer-defined character string. If you include the minus sign (-), then the source field is right justified in the destination field. If *character-string* is null, as in "", then the receiving field is filled with binary zeros. To fill the receiving field with blanks, use a space," ", for the character string.
source1 + *source2*	Concatenates the contents of *source1* and *source2* and places the result in the destination field. Any trailing blanks are stripped on concatenation.
source1 - *source2*	Removes the contents of *source2* from *source1* as many times as the contents of *source2* are found in *source1*. Places the results in the destination field.

STATUS(*parm*)	Moves a value to the destination field, depending on the value of *parm*. If *parm* is:
DB	Moves status block used in last IMAGE call to the data register location specified by *item-name*.
BASE	Moves the data base name referenced in the last IMAGE call to the data register location specified by *item-name*.
FILE	Moves the name of the data set or file referenced by the last IMAGE, KSAM, or MPE call to the data register location specified by *item-name*.

EXAMPLES

1) MOVE (FIELD-B)=(FIELD-A);

```
                 Before MOVE    After MOVE

                 ---------      ---------
FIELD-A X(4)     |S|A|M| |      |S|A|M| |         (no change)
                 ---------      ---------
                 -----------    -----------
FIELD-B X(5)     |C|H|U|C|K|    |S|A|M| | |
                 -----------    -----------
```

2) MOVE (MONTH) = (DATE);

```
                 Before MOVE    After MOVE

                 -------------  -------------
DATE X(6)        |1|0|0|7|7|0|  |1|0|0|7|7|0|     (no change)
                 -------------  -------------
                 -----          -----
MONTH X(2)       |1|2|          |1|0|
                 -----          -----
```

Note that the last four digits are truncated.

3) The following example illustrates concatenation:

```
MOVE (NEWFIELD)=(FIELD1)+(FIELD2);

                 Before MOVE    After MOVE

                ---------      ---------
FIELD1 X(4)     |A|B| | |      |A|B| | |       (no change)
                ---------      ---------
                -------        -------
FIELD2 X(3)     |C|D|E|        |C|D|E|         (no change)
                -------        -------
                -------------  -------------
NEWFIELD X(6)   |1|2|3|4|5|6|  |A|B|C|D|E| |
                -------------  -------------
```

Note that the trailing blanks in FIELD1 are stripped when the two fields are concatenated.

4) The following example illustrates "removal" of characters:

```
MOVE(DATE) = (FDATE) - (SLASH);

                 Before MOVE         After MOVE

               -----------------   -----------------
FDATE  X(8)    |0|1|/|3|1|/|8|2|   |0|1|/|3|1|/|8|2|   (no change)
               -----------------   -----------------
               ---                 ---
SLASH  X(1)    |/|                 |/|                 (no change)
               ---                 ---
               -------------       -------------
DATE   X(6)    | | | | | | |       |0|1|3|1|8|2|
               -------------       -------------
```

5) And, this example illustrates justification:

```
MOVE (FIELDY)= -(FIELDX);

                 Before MOVE    After MOVE

                ---------      ---------
FIELDX X(4)     |A|B|C| |      |A|B|C| |       (no change)
                ---------      ---------
                ---------      ---------
FIELDY X(4)     |1|2|3|4|      | |A|B|C|
                ---------      ---------
```

OUTPUT

Causes a multiple data retrieval from a file or data set and displays the data

```
**************************************************************
*                                                            *
*  OUTPUT[(modifier)] file-name[,option-list];               *
*                                                            *
**************************************************************
```

OUTPUT specifies a data base or file retrieval operation. It adds each retrieved record to the data register, but only selects for output those records that satisfy any selection criteria in the match register. For each selected record, OUTPUT displays all the items in the current list register. If you want to select items from the list register, you should precede the OUTPUT statement with a FORMAT statement.

The OUTPUT statement displays the selected entries after any PERFORM= statements are executed. This allows you to display the results of any PERFORM= statements. However, this makes nesting of OUTPUT statements difficult. The output from the most deeply nested OUTPUT statement is displayed first. To produce nested output in the more usual order, you can use a FIND statement to retrieve the data with a PERFORM= option to display the data.

If a FORMAT statement appears before the OUTPUT statement, then the display is formatted according to the specifications in that statement. If there is no preceding FORMAT statement, the display is formatted according to the default format described below. Once all entries have been displayed according to a preceding FORMAT statement, subsequent OUTPUT statements revert to the default format unless control passes again through a FORMAT statement.

The default format for OUTPUT is:

- Display values in order they appear in data register.
- Accompany each value with a heading consisting of:
    - the heading specified for that value in a HEAD= option of a DEFINE(ITEM) statement,
    - the heading taken from the dictionary definition of the item, or
    - the associated data item name in the list register.
- Display each value in a field whose length is either the data item size or the heading length, whichever is longer.
- A single blank character separates each value field. If a field cannot fit on the current display line, then the field begins on a new line.

STATEMENT PARTS

modifier	To specify the type of access to the data set or file, choose one of the following modifiers:
none	Retrieve an IMAGE master set entry based on the value in the argument register. This option does not use the match register.
CHAIN	Retrieve entries from an IMAGE detail data set or a KSAM chain. The entries must meet any match criteria set up in the match register. The contents of the key and argument registers specify the chain in which the retrieval is to occur. If no match criteria are specified, all entries are selected. If match criteria are specified, the match items must be included in a LIST= option of the OUTPUT statement.
CURRENT	Retrieve the last entry that was accessed from an MPE or KSAM file or an IMAGE data set.
DIRECT	Retrieve the entry stored at a specified record number from an MPE or KSAM file or an IMAGE data set. Before using this modifier, store the record number as a doubleword integer in the item referenced by the RECNO= option.
PRIMARY	Retrieve the IMAGE master set entry stored at the primary address of a synonym chain. The primary address is located through the key value contained in the argument register.
RCHAIN	Retrieve entries from an IMAGE detail data set chain in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.
RSERIAL	Retrieve entries from an IMAGE data set in the same manner as the SERIAL option, except in reverse order. For a KSAM or MPE file, this operation is identical to SERIAL.
SERIAL	Retrieve entries in serial mode from an MPE or KSAM file or an IMAGE data set that meet any match criteria set up in the match register. If no match criteria are specified, all entries are selected. If match criteria are specified, the match items must be included in a LIST= option of the OUTPUT statement.
file-name	The file or data set to be accessed in the retrieval operation. If the data set is not in the home base as defined in the SYSTEM statement, specify the base name in parentheses: *set-name*(*base-name*)

option-list:	One or more of the following options separated by commas:

ERROR=*label* ([*item-name*])	Suppress the default error return that the processor normally takes. Instead, the program branches to the statement identified by *label*, and Transact sets the list register pointer to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register. If you do not specify an item-name, as in ERROR=*label*();, the list register is reset to empty. If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not changed. For more information, see "Automatic Error Handling", in section 5.
LIST= (*range-list*)	The list of items from the list register to be used for the data retrieval portion of the OUTPUT operation. The display portion follows the same rules as the DISPLAY statement. If the LIST= option is omitted, the entire list register is used for the data retrieval. Only the items specified in a LIST= option have their match conditions applied if match conditions are set up in the match register. (The match register may be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.) Each retrieved entry is placed in the area of the data register indicated by LIST= before any PERFORM= is executed. The options for *range-list* and the items they cause OUTPUT to retrieve are:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.

(*item-name1:*)	The items from *item-name1* through the item indicated by the current list register pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2.*
(*item-name1, item-name2,* ... *item-namen*)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM and MPE, items must be specified in the order of their occurrence in the record. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
()	A null item list. That is, access the file or data set, but do not transfer any data.

LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked while the OUTPUT executes. If LOCK is not specified, the data base is locked before each entry is retrieved, remains locked while the entry is processed by any PERFORM= statements, but is unlocked briefly before the next entry is retrieved.
NOCOUNT	Suppress the message normally generated by the processor to indicate the number of entries found.
NOHEAD	Suppress default headings for the displayed values.
NOMATCH	Ignore any match criteria set up in the match register. This option is useful if you want to leave the match register set up but do not want to use it.
NOMSG	Suppress the standard error message produced by the processor as a result of a file or data base error; all other error recovery actions occur.
PERFORM=*label*	Execute the code following the specified label for every entry retrieved by the OUTPUT operation. The entries may be optionally selected by MATCH criteria, in which case the PERFORM= statements are executed only for the selected entries. This option allows operations to be performed on retrieved entries without your having to code loop-control logic. You may nest up to 10 PERFORM= options.

RECNO=*item-name*	With the DIRECT modifier: You must define *item-name* to contain the doubleword integer number (I(9,,4)) of the record to be retrieved. With other modifiers: Transact returns the record number of the retrieved record in *item-name*, a doubleword integer (I(9,,4)).
SINGLE	Retrieve and display only the first entry that satisfies any selection criteria.
SOPT	Suppress the processor optimization of IMAGE calls. This option is intended to support a data base operation in a routine that is called recursively. The option allows a different path of the same detail data set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of an IMAGE call list of "*" after the first call is made. (For an example using SOPT in a recursive routine, see the FIND verb examples.)

```
SORT=[(item-name1:item-name2)] (item-name3[(ASC)]
                                          [(DES)]
     [,item-name4[(ASC)]]...);
                 [(DES)]
```

This option sorts each occurrence of *item-name3* and, optionally, *item-name4*, and so forth.

The list used to define the sort file record is either the range of items specified by *item-name1:item-name2*, or if *item-name1* and *item-name2* are omitted, the entire list register. You can use the optional range to prevent unneeded variables from being written to the sort file. In general, only send to the sort file the items that will be formatted for output.

The OUTPUT statement always sorts after processing any PERFORM= statements. The processing sequence for the sort is:

- First, retrieve each selected record,
- then, execute any PERFORM= statements,
- then write the specified items to the sort file, and, after writing all the records to the sort file,
- sort the sort file, and
- display the sorted output.

(See the FIND verb description for a different processing sequence.)

You may specify either ascending or descending sort order. The default is ascending order.

STATUS

Supress processor action defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of an OUTPUT statement is described by the value in the status register:

Status Register Value	Meaning
0	The OUTPUT operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with OUTPUT:

- Normal multiple accesses become single.
- Supresses the normal rewind done by OUTPUT, so CLOSE should be used before OUTPUT(SERIAL).
- Suppresses the normal find of the chain head by OUTPUT, so PATH should be used before OUTPUT(CHAIN).

When you want to avoid the normal error recovery so you can print totals on an error, use STATUS with OUTPUT. The following example transfers control to a PERFORM label to compute the totals and then displays the error code.

```
TRYAGAIN:
   PROMPT(PATH) ACCT-NO;
   OUTPUT(CHAIN) DETAIL-SET,
      LIST=(A:N),
      ERROR=ERROR-MSG(*),
      PERFORM=TOTAL;
   PERFORM OUTPUT-TOTAL;
   END;

   ERROR MSG:
      DISPLAY "INVALID ACCT-NO";
      GO TO TRYAGAIN;
```

EXAMPLES

The following two examples of OUTPUT retrieve data according to a value entered by the user. Then they display the data according to the preceding FORMAT statement.

```
LIST NAME:
     ADDRESS:
     CITY:
     ZIP;
PROMPT(KEY) CUST-NO;
FORMAT NAME,COL=5:
       ADDRESS,COL=20:
       CITY,SPACE=5:
       ZIP,SPACE=5;
OUTPUT MASTER,
       LIST=(NAME:ZIP);



PROMPT(PATH) CUST-NO;
LIST COMPANY:
     CO-ADDR:
     CO-STATE:
     ZIP;
FORMAT COMPANY, COL=5:
       CO-ADDR,COL=40:
       CO-STATE,LINE,COL=5:
       ZIP, COL=40;
OUTPUT(CHAIN) DETAIL,
       LIST=(COMPANY:ZIP);
```

The following example retrieves the entries that satisfy the match criterion LAST-NAME = SMITH from the data set CUSTOMER, then sorts the entries according to FIRST-NAME and displays only the sorted names.

```
LIST LAST-NAME:
     FIRST-NAME;

MOVE (LAST-NAME) = "SMITH";
SET(MATCH) LIST (LAST-NAME);

FORMAT LAST-NAME:           << Items to be displayed >>
       FIRST-NAME, JOIN=2;

OUTPUT(SERIAL) CUSTOMER,
     NOCOUNT, NOHEAD,
     SORT=(FIRST-NAME);     << Sort on first name    >>
```

The resulting display looks like:

```
Smith  Abraham
Smith  John
Smith  Joseph
Smith  Mary
Smith  Thomas
```

In the next example, some of the items selected for sorting and display are calculated in a PERFORM= routine.

```
   LIST INV-NO:
        PRICE:
        QUANTITY:
        AMOUNT:
        TOT-AMT;

   OUTPUT(SERIAL) INVENTRY,
       LIST=(INV-NO:QUANTITY),
       PERFORM=TOTAL,
       SORT=(INV-NO:AMOUNT)
            (AMOUNT);

 TOTAL:
   LET (AMOUNT) = (PRICE) * (QUANTITY);
   LET (TOT-AMT)= (TOT-AMT) + (AMOUNT);
   RETURN;
```

Establishes a chained access path to an IMAGE data set or a KSAM file

```
**************************************************************
*                                                            *
*  PATH file-name[,option-list];                             *
*                                                            *
**************************************************************
```

PATH uses the key and argument registers to locate the key that defines a given chain in a KSAM file or an IMAGE detail data set. If you do not include a STATUS option in the PATH statement, the status register is set to the number of entries in the chain of an IMAGE detail set; this information is not returned for a KSAM file.

You must use a PATH statement to establish the path for chained access to an IMAGE data set or a KSAM file when the STATUS option is included in a subsequent data access statement. PATH may not be used with MPE files.

STATEMENT PARTS

file-name The KSAM file or IMAGE data set to be accessed. If the data set is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows: *set-name*(*base-name*)

If you specify a set name and do not include the STATUS option, the status register is set to the number of entries in the data set chain; the status register does not contain this information for a KSAM file.

option-list One or more of the following fields, separated by commas:

ERROR=*label* ([*item-name*]) Suppress the default error return that Transact normally takes. Instead, the program branches to the statement identified by *label*, and Transact sets the list register pointer to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register.

If you do not specify an *item-name*, as in ERROR=*label*();, the list register is reset to empty.

If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not changed.

For more information, see "Automatic Error Handling," in section 5.

LIST= (*range-list*)	Used only with KSAM files to map out a record. The list option is needed to locate the key in the KSAM record. The options for *range-list* and the records upon which they operate include the following:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current list pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.
(*item-name1*, *item-name2*, ... *item-namen*)	The items are selected from the list register. Items must be specified in the order of their occurrence in the record. This option incurs some system overhead.

NOMSG	The standard error message produced by Transact as a result of a file or data base error is to be suppressed.

STATUS	Suppress processor action defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of a PATH statement is described by the value in the status register:

Status Register Value	Meaning
0	The PATH operation was successful
-1	A KSAM end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or KSAM file system error documentation corresponding to the value.

Note that when STATUS is omitted, the status register contains a -1 if the argument value for a PATH operation on a detail data set is not found in the associated master data set. Otherwise it contains the number of entries in the chain.

EXAMPLES

The following example uses a PATH statement to locate the head of a KSAM chain, and then retrieves the first item in that chain.

```
LIST DEL-WORD:
     CUST-NO:
     LAST-NAME:
     FIRST-NAME:
     INITIAL;
PROMPT(KEY) CUST-NO ("Enter Customer Number");  <<set up key/arg registers>>
PATH KFILE,                                    <<Locate head of chain in KFILE>>
     LIST=(DEL-WORD:INITIAL);                  <<Map KFILE record>>
IF STATUS <> 0 THEN
  GET(CHAIN) KFILE,                            <<retrieve first record>>
     STATUS,
     LIST=(DEL-WORD:INITIAL);
```

The next example uses a PATH statement to determine the number of records in an IMAGE detail data set.

```
PROMPT(PATH) CUST-NO;
PATH CUST-DETAIL;
LET (NUM-RECS) = STATUS;
DISPLAY NUM-RECS, NOHEAD:
        "Records in this Path";
```

PATH is required before you use the STATUS option in a data base access statement because the STATUS option suppresses the usual determination of a chain head. In the following example, the PATH statement is needed prior to the FIND(CHAIN) statement that includes a STATUS option:

```
 SET(KEY) LIST(CUST-NO);
 PATH CUST-DETAIL;
GET-NEXT:
 FIND(CHAIN) CUST-DETAIL,
     LIST=(CUST-NO:ZIP),
     STATUS,
     PERFORM=PROCESS-ENTRY;
 IF STATUS <> 0 THEN
   GO TO ERROR-ROUTINE
 ELSE
   GO TO GET-NEXT;
```

Note that the STATUS option also suppresses the normal multiple retrieval performed by FIND; you must specifically code the loop logic.

Transfers control to a labelled statement

```
*****************************************************************
*                                                               *
*  PERFORM label;                                               *
*                                                               *
*****************************************************************
```

PERFORM transfers execution to the statement identified by *label*; execution continues until one of the following is encountered:

RETURN;	Returns control to the statement immediately following the last PERFORM statement executed
END;	Specifies the end of the current processing level and returns control to the previous processing level, or to command level if no previous processing level is active within the perform block.
command or sub-command label	Specifies the end of the current command sequence. The compiler generates an END statement, and the effect is the same as END;.

PERFORM statements can be nested up to a maximum of 75 levels. Note that this differs from PERFORM= options in data management verbs, which allow a maximum of 10 levels of nesting. This difference is due to the number of entries in an internal table: the PERFORM statement uses a 2-word entry, whereas PERFORM= uses a 15-word entry. The total table size is approximately 150 words.

Although GO TO statements can branch into and out of PERFORM statement loops, this is not generally good coding practice.

STATEMENT PARTS

label	The label that identifies the sequence of statements called by PERFORM.

EXAMPLES

```
IF INPUT = "YES", "Y" THEN
   PERFORM ADDIT
ELSE GO TO GET-ACCT;
PROMPT INV-NUM("Invoice Number"), RIGHT;
   .
   .
   .
END;

ADD-IT:
   PUT CUST-FILE,
       LIST=(NAME:ZIP);
   LET (NUM) = (NUM) + 1;
   DISPLAY NAME, COL=5, NOHEAD:
           "HAS BEEN ADDED TO CUSTOMER FILE.", JOIN;
RETURN;
```

When the response to INPUT causes a transfer to the label ADD-IT, the statements between ADD-IT and RETURN execute. Control then returns to the PROMPT statement that immediately follows the IF statement.

Calls a procedure that has been loaded into a segmented library (SL) file

```
**************************************************************
*                                                            *
*  PROC procedure-name [(parameter-list)] [,option-list];    *
*                                                            *
*                                                            *
**************************************************************
```

PROC calls an MPE system intrinsic or other compiled procedure that is resident in an SL file. SL files are searched in the following order: group SL, account SL, system SL.

The PROC statement does not directly support intrinsics with an optional number of parameters (Option Variable Intrinsics); you may call such intrinsics by using a bit map to specify the parameters you want passed. (Refer to the *SPL Reference Manual* for more information on Option Variable bit maps.)

Ensure that any intrinsics called are declared using DEFINE(INTRINSIC).

STATEMENT PARTS

procedure-name The name under which the procedure is listed in the SL directory

parameter-list Optional items in the *parameter-list* specify one or more variables that are passed between the Transact program and the external procedure. The list may contain any of a number of variables in any order. The order in which you place the variables is determined by the order in which the called procedure expects them. The only exception is that a function return variable can be placed anywhere in the list; a function return variable is indicated by a preceding "&" (see "NOTE" below).

The variable identifiers should be separated by commas. You may indicate to the called procedure the existence of a null parameter by placing consecutive commas on the list. Transact passes a single word value of zero for this null parameter. Use two commas if the parameter has a double-word value.

All addresses specified by the items in *parameter-list* are word addresses. If you want to specify a byte address, precede the item-name with "%". For example, ITEM(NUM) specifies a word address, whereas %ITEM(NUM) specifies a byte address. PROC does not automatically align data parameters on word boundaries.

NOTE: The following special characters may precede any parameter:

% Passes the given parameter as a byte address

# Passes the given parameter by value rather than by reference

& Copies the function value returned by the intrinsic to the field in the data register associated with the given item, or to the STATUS register. Only one such designated parameter may be included in the *parameter-list*, and it may appear anywhere in the list.

The *parameter-list* may consist of any of the following:

(*item-name*)	Address of a logical array containing the value of an item in the data register. Use this parameter to pass any values defined in your program. It is up to you to make sure that the item is on a word boundary in the data register if you want to pass a word address. The beginning of the data register is on a word boundary; if you add items with an odd number of bytes, you should add a dummy fill character to retain word boundaries.

You can include any of the following key words in a *parameter-list*. If the key word has an argument, it must immediately follow the key word with no intervening blanks. Transact supplies a value (usually an address) whenever it finds one of these key words in a parameter list.

ARG	Address of a logical array containing the argument value currently associated with the key for data set or file operations.
ARGLNG	Address of an integer word containing the byte length of the argument value.
BASE [(*base-name*)]	Address of a logical array containing the name of the given data base preceded by the two-character *base-id* supplied by IMAGE, and followed by a blank character. If no *base-name* is specified, then the home base is assumed.
BASELNG [(*base-name*)]	Address of an integer word containing the byte length of the given *base-name*, including the terminating blank.

BYTE(*item-name*)	Address of an integer word containing the byte length of the value of the given item.
COUNT (*item-name*)	Address of an integer word containing any sub-item occurrence count for the given item. A value of 1 means that the given item is not a compound type containing sub-items.
DECIMAL (*item-name*)	Address of an integer word containing the decimal place count for the given item.
FILEID (*file-name*)	Address of an integer word containing the identifier assigned to *file-name* by MPE when the file was opened by this process. The following special files can also be used in conjunction with the FILEID parameter: ● TRANIN Transact input file ● TRANOUT Transact output file ● TRANLIST Transact printer output file
INPUT	Address of the logical array containing the value that was last input in response to an INPUT statement prompt.
INPUTLNG	Address of an integer word containing the byte length of the input value.
ITEM(*item-name*)	Address of a logical array containing the name of the given item.
ITEMLNG (*item-name*)	Address of an integer word containing the byte length of the given item name.
KEY	Address of a logical array containing the name of the data item currently used as a key for data set or file operations. The data item name must be terminated by a semicolon (;).
KEYLNG	Address of an integer word containing the byte length of the data item name in the key, including the terminating semicolon.
POSITION (*item-name*)	Address of an integer word containing the position, that is, the byte offset, of a child item within its parent item. This parameter is set to -1 to indicate that there is no parent item.
SET(*set-name*)	Address of a logical array containing the name of the given data set followed by a blank.

SETLNG (*set-name*) Address of an integer word containing the byte length of the given data set name, including the terminating blank.

SIZE(*item-name*) Address of an integer word containing the byte length of the display or entry format for the given item.

STATUS Address of an integer word containing the value for the status register set by the Transact processor. If the STATUS parameter is NOT used, then the status register is set to one of the condition codes generated by the called procedure (CCL, CCE, or CCG). Condition codes are defined as follows:

```
CCL  =  -1
CCE  =   0
CCG  =  +1
```

Condition codes in the status register can be tested with a subsequent IF statement. For example:

```
IF STATUS < 0 THEN
   GO TO CCL-PROCESS;
```

Where CCL-PROCESS will handle a CCL condition.

STATUS(DB) Address of the IMAGE condition word block.

STATUS(IN) Address of an integer word containing the STATUS value following the most recent user input statement (PROMPT/DATA/INPUT). See the appropriate verb for the interpretation of the STATUS value.

TYPE(*item-name*) Address of an integer word containing a code that represents the data type of *item-name*. The integer code represents the data type by its position in the sequence: X, U, 9, Z, P, I, J, K, R, E,@; thus, the code corresponds to a data type as follows:

0=X,1=U,2=9,3=Z,4=P,5=I,6=J,7=K,8=R,9=E, and 10=@ (the marker item)

VCOM(*form-file*) Address of the logical array containing the VPLUS communication area being used for the referenced *form-file*.

option-list One or more of the following options may follow the parameter list, separated by commas:

UNLOAD Unload the procedure being called following execution; that is, remove it from the Loader Segment Table. By default, Transact leaves an entry in the Loader Segment Table for each called procedure after it executes. Only use this option if you do not need the procedure again. Otherwise, Transact incurs extra overhead loading the procedure the next time it is called.

NOTRAP	Ignore any arithmetic trap detected in the operation of the procedure. By default, Transact issues an error message and terminates the called procedure when it encounters an arithmetic error.
NOLOAD	Load the called program the first time it is called rather than when the program is initiated. By default, Transact loads all external procedures when it initiates the calling program. Used in combination with UNLOAD, this option can save Loader Segment Table space. NOLOAD is ignored if the called procedure is an MPE system intrinsic declared in a DEFINE(INTRINSIC) statement; if you want such a procedure to be loaded dynamically, do not include it in a DEFINE(INTRINSIC) statement

EXAMPLES

The format of the intrinsic ASCII in the *MPE Intrinsics Manual* is:

```
I              LV   IV   BA
numchar:=ASCII(word,base,string);
```

The PROC verb to call the ASCII intrinsic has the following format:

```
PROC ASCII (#(WORD),#(BASE),%(STRING),&(NUMCHAR));
```

WORD, BASE, and STRING are program variables that correspond to the parameters of the intrinsic and NUMCHAR is a functional return variable to which the procedure returns the number of characters, NUMCHAR, translated by the ASCII intrinsic. Note that NUMCHAR is at the end of the PROC parameter list rather than in its position in the intrinsic definition. WORD and BASE are preceded by a "#" symbol because they are passed by value; STRING is a byte address as indicated by the preceding "%".

The following example calls the intrinsics CREATE and ACTIVATE. (Refer to the *MPE Intrinsics Reference Manual* for the syntax of these intrinsics.) Since both intrinsics are Option Variable, a bit map (MAP) is included at the end to indicate which parameters to pass. Because this map and the CFLAG parameter are passed by value, they are preceded by a "#" symbol.

```
DEFINE(ITEM) ROUTINE X(20):       <<Process name>>
             CPIN    I(4):        <<PIN of process>>
             CFLAG   I(4):        <<Flags>>
             MAP     I(4);        <<Bit map for optional parameters>>
$$A:
    LIST ROUTINE,INIT:
         CPIN,INIT:
         CFLAG,INIT:
         MAP,INIT;

    LET (MAP) = 672;   <<decimal equivalent of bit map "1010100000">>
    LET (CFLAG) = 73;  <<decimal equivalent of bit map "1001001"    >>
    DATA ROUTINE("WHICH PROCESS?");

    PROC CREATE (%(ROUTINE),,(CPIN),,#(CFLAG),,,,,,#(MAP));

    LET (MAP) = 3;
    LET (CFLAG) = 3;

    PROC ACTIVATE (#(CPIN),#(CFLAG),#(MAP));

    END;
```

Note that the MAP parameter sets up a bit map for an intrinsic that is type OPTION VARIABLE.

The next example calls the VPLUS/3000 procedure VPRINTFORM to print a form on the line printer.

```
      SYSTEM TEST,
          VPLS=CUSTFORM;             << Form definition in Dictionary  >>
      DEFINE(ITEM)  PRINTCNTL I(2):
                    PAGECNTL  I(2):
                      .
                      .
                      .
      DEFINE(INTRINSIC) VPRINTFORM;
                      .
                      .
                      .
   PRINT:
      LIST PRINTCNTL:
           PAGECNTL;

      LET (PRINTCNTL) = 1;           << Underline Fields >>
      LET (PAGECNTL)  = 0;           << CR/LF Off        >>

      PROC VPRINTFORM (VCOM(CUSTFORM),
                       (PRINTCNTL),
                       (PAGECNTL)),
```

Note that Transact supplies the comarea location for the forms file CUSTFORM automatically through the parameter VCOM(file name).

The next example calls the IMAGE intrinsic DBCLOSE using the BASE, SET, and STATUS key-word parameters.

```
   SYSTEM TEST,
      BASE=CUSTOMER ("MANAGER");
   DEFINE(ITEM) MODE  I(2);
   DEFINE(INTRINSIC) DBCLOSE;
       .
       .
   LET (MODE) = 5;
   PROC DBCLOSE(BASE(CUSTOMER),
                SET(CUST-MAST),
                (MODE),
                STATUS(DB));
```

The next example shows a call to DSG/3000 intrinsics. The data register size is increased because of DSG requirements:

```
SYSTEM DSG,
       DATA=4000,10;

DEFINE(ITEM) GRAF   1415I+(2,,2):
             GRAFSIZE   I(4,,2):
             LANG       I(1,,2);

LIST GRAF:GRAFSIZE:LANG;

LET (GRAFSIZE) = 1415;
LET (LANG)     = 0;

PROC GINITGRAF((GRAF),(GRAFSIZE),(LANG));

DISPLAY "Return from GINITGRAF";
```

The following example illustrates the use of the FWRITE intrinsic in conjunction with the Transact terminal output file TRANOUT:

```
SYSTEM Demo01;

DEFINE(INTRINSIC) FWRITE;

DEFINE(ITEM) MSG X (30);
DEFINE(ITEM) COUNT I(4);
DEFINE(ITEM) CONTROL I(4);

LIST MSG : COUNT : CONTROL;

MOVE (MSG) = "HELLO THERE WORLD!!";
LET (COUNT) = -19;
LET (CONTROL) = 0;

PROC FWRITE (#FILEID(TRANOUT),(MSG), (COUNT), (CONTROL));
```

Accepts input from the user terminal and places the supplied values into the list, data, argument, match, and/or update registers

```
**********************************************************************
*                                                                    *
* PROMPT[(modifier)] item-name[("prompt-string")][,option-list];     *
*                                                                    *
**********************************************************************
```

PROMPT prompts the user for values and, depending on the syntax option chosen, places the value in one or more registers. The register affected depends on the verb modifier. You may choose from the following:

● none	Add item name to list register and input value to data register (see Syntax Option 1).
● KEY	Add item name to key register and add input value to argument register (see Syntax Option 2).
● MATCH	Add item name to list and match registers and add input value to data register; also set up input value in match register as a match criterion (see Syntax Option 3).
● PATH	Add item name to list and key registers, and add input value to data and argument registers (see Syntax Option 4).
● SET	Add item name to list register and add input value to data register, unless response is a carriage return (see Syntax Option 5).
● UPDATE	Add item name to list and update registers and input value to data register; also add input value to update register for subsequent replace operation (see Syntax Option 6).

PROMPT is used to set up and perform a data entry operation, usually for a subsequent data set or file operation. At execution time it prompts the end user with a prompt string, the entry text associated with the item, or with the item name to request the value of the data item. An entry text can be associated with an item in the dictionary or in the DEFINE(ITEM) defintion of the item.

Transact validates the input value as to type, length, or any other characteristics specified in the dictionary or in a DEFINE(ITEM) statement before it modifies the specified register. If Transact detects an error, it displays an appropriate error message and reissues the prompt automatically.

STATEMENT PARTS

modifier	Changes or enhances the PROMPT verb; usually determines the register in which to place the item name and the register to which the input value should be added or the register whose value should be changed.
item-name	The name of the data item to be placed in the list register and/or another register, and whose value should be added to or changed in the data register and/or another register. The item name cannot be the name of a child item.
prompt-string	The string that prompts the terminal user for the input value. If omitted, the prompt is the entry text associated with the item, or if there is no entry text, then the prompt is the item name.
option-list	A field specifying how the data should be formatted and/or other checks to be performed on the value. Choose one or more (separated by commas) of the following options for any syntax option. (See Syntax Option 3, PROMPT(MATCH) for additional options):

BLANKS	Do not suppress leading blanks supplied in the input value (Leading and trailing blanks are normally stripped.)
CHECK=*set-name*	Check the input value against the IMAGE master set *set-name* to ensure that a corresponding search item value already exists. If the value is not in the data set at execution time, Transact displays an appropriate error message and reissues the prompt. This option cannot be used to check against MPE or KSAM files, nor can it be included in a PROMPT(MATCH) statement.
CHECKNOT= *set-name*	Check input value against the IMAGE master set *set-name* to ensure that a corresponding search item value does not already exist. If the value is in the data set at execution time, then Transact displays an appropriate error message and reissues the prompt. This option cannot be used to check against MPE or KSAM files, nor can it be included in a PROMPT(MATCH) statement.
NOECHO	Do not echo the input value to the terminal. If omitted, the input value is displayed on the terminal.
RIGHT	Right justify the input value within the data register field. By default, the input value is left justified.

STATUS	Suppress normal processing of "]" or "]]", which causes an escape to a higher processing or command level. Instead, set the status register to -1 if the end user enters a "]" or to -2 if the end user enters a "]]". If the user enters one or more blanks and no non-blank characters, then the status register is set to -3. (The status register normally contains the number of characters entered in response to the prompt, including leading blanks if the BLANKS option is used; it never counts trailing blanks.) The STATUS option allows you to control subsequent processing by testing the contents of the register with an IF statement.
	If the CHECK or CHECKNOT option is also used, then "]", "]]", or a carriage-return suppresses the data set operation and control passes to the next statement.

SYNTAX OPTIONS

(1) PROMPT *item-name* [("*prompt-string*")] [,*option-list*];

PROMPT with no modifier adds the *item-name* to the list register and the input value to the data register.

(2) PROMPT(KEY) *item-name* [("*prompt-string*")] [,*option-list*];

PROMPT(KEY) places the *item-name* in the key register and the input value in the argument register. The data item and its value are used as a retrieval key for a subsequent data set or file operation.

(3) PROMPT(MATCH) *item-name*[("*prompt-string*")] [,*option-list*];

PROMPT(MATCH) adds the *item-name* to the list and match registers. In addition, it adds the input value to the data register and also sets up this value as a selection criterion in the match register for a subsequent file operation.

The user response to PROMPT(MATCH) may be any of the valid selection criteria described under "Responding to a Match Prompt" in section 5. If the response is a carriage return, then all values for the data item are selected. If the response contains several values separated by connectors, only the first value is placed in the data register space for the item. If a particular value is input, then all entries that match the associated data item are selected.

The MATCH modifier allows one or more of any of the *option-list* items listed above under "Statement Parts", except for CHECK= and CHECKNOT=, which are not allowed in a PROMPT(MATCH) statement. You may, in addition, select one of the following options, to specify that a match selection is to be performed on a basis other than equality.

option-list:

NE	Not equal to
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
LEADER	Matched item must begin with the input string; equivalent to the use of trailing "^" on input
SCAN	Matched item must contain the input string; equivalent to the use of trailing "^^" on input
TRAILER	Matched item must end with the input string; equivalent to the use of a leading "^" on input

For example, for the following command and response sequence, the data base or file entries selected will contain EMPL values starting with "LIT", AGE values less than 35, and LOS values greater than or equal to 5:

```
PROMPT(MATCH) EMPL:
              AGE, LT:
              LOS, GE;

EMPL> LIT^
 AGE> 35
 LOS> 5
```

(4) PROMPT(PATH) *item-name* [("*prompt-string*")] [,*option-list*];

PROMPT(PATH) adds the *item-name* to the list register and the key register. In addition, the input value is added to the data register and the argument register. Use this modifier to set up a data item for a data set or file operation and its value for use as a retrieval key.

(5) PROMPT(SET) *item-name* [("*prompt-string*")] [,*option-list*];

PROMPT(SET) adds the *item-name* to the list register and the input value to the data register only if the input value is not a carriage return. If the user responds to the prompt with a carriage return, no additions are made to the list and data registers. The modifier is primarily used to set up a data item list for a data set or file operation using the UPDATE verb, where the user controls that list by means of his or her responses.

For example, the following PROMPT(SET) statement and the responses to its prompts produce a list register content of "PHONE" and "ROOM" and a data register content of the associated supplied values:

```
     PROMPT(SET) EMPL:
                 DEPT:
                 PHONE:
                 ROOM:
                 LOCATION;


     EMPL>
     DEPT>
    PHONE> 278
     ROOM> 312
 LOCATION>
```

Note that if you use the CHECK= option and the item is not found in the data set, you must clear this value from the match register before you reissue the prompt.

(6) PROMPT(UPDATE) *item-name* [("*prompt-string*")] [,*option-list*]
[:*item-name*...]...;

PROMPT(UPDATE) adds the *item-name* to the list and update registers, and adds the input value to the data register. In addition, it sets up the input value in the update register for a subsequent data set or file operation using REPLACE. When a subsequent REPLACE statement is executed, it replaces any value for the specified data item with the value added to the update register.

EXAMPLES

```
$$ADD:                        <<Add a new record>>
   $CUSTOMER:
      PROMPT CUST-NAME("CUSTOMER'S NAME"):
             CUST-ADDR:
             CUST-CITY:
             CUST-PHONE;
```

This example causes the following sequence of prompts to appear on the terminal:

```
CUSTOMER'S NAME>
CUST-ADDR>
CUST-CITY>
CUST-PHONE>
```

The following example adds a new customer number to the data set and then adds transactions for that customer. It checks to make sure that the customer number entered by the user is not already in the data set and that the transactions apply to a customer number that is in the data set.

```
$$ADD:                     <<Add new customer>>
   $CUSTOMER:
      PROMPT(PATH) CUST-NUMBER,
            CHECKNOT=CUST-MASTER;
        .
        .
      PUT CUST-MASTER;
   $TRANS:
      PROMPT(PATH) CUST-NUMBER,
            CHECK=CUST-MASTER;
      PROMPT INV-NUMBER:
             AMOUNT;
        .
        .
      PUT CUST-DETAIL;
```

Moves data from the data register to a file, data set, or a formatted screen

```
****************************************************************
*                                                              *
*  PUT[(FORM)] destination [,option-list];                     *
*                                                              *
****************************************************************
```

PUT moves an entry from the list and data registers into a file or a data set; or it displays data in a VPLUS form.

STATEMENT PARTS

FORM — Optional modifier. If "FORM" is specified, then PUT(FORM) displays a VPLUS form on a 262X or 264X series terminal, and moves data to the form from the data register.

destination — The file, data set, or form to be accessed in the write operation.

If the destination is a data set that is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows:

set-name(*base-name*)

In a PUT(FORM) statement, the destination must identify a form in a forms file that was named in the SYSTEM statement. For PUT(FORM) only, *destination* may be specified as any of the following:

form-name — Name of the form to be displayed by PUT(FORM).

(*item-name*) — Name of an item that contains the name of the form to be displayed by PUT(FORM).

* — Display the form identified by the "current" form name; that is, the form name most recently specified in a statement that references VPLUS forms. Note that this option is not the same as the CURRENT option (described under *option-list*), which indicates the currently displayed form.

& — Display the form identified as the "next" form name; that is the form name defined as "NEXT FORM" in the FORMSPEC definition of the current form.

option-list	The LIST option is available with or without the FORM modifier. Other options, described below, may be used only without or only with the FORM modifier. If the LIST= option is omitted from a PUT statement without the FORM modifier, all the items from the current list register are used for the file or data set operation. If LIST= is omitted from a PUT(FORM) statement, it uses all the items both currently in the list register and defined for the form in the SYSTEM statement or the dictionary.
LIST= (*range-list*)	The list of items from the list register to be used for the PUT operation. For PUT(FORM) only, items in the range list can be child items. The options for *range-list* and the records upon which they operate include the following:

(*item-name)*	A single item.
(*item-name1: item-name2*)	All the items from *item-name1* through *item-name2.* If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1:*)	The items from *item-name1* through the item indicated by the current stack pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2.*
(*item-name1, item-name2,* ... *item-name*n)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM, VPLUS, and MPE, items must be specified in the order of their occurrence in the record or form. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
()	A null item list. That is, access the file or data set, or display the form, but do not transfer any data.

OPTIONS AVAILABLE WITHOUT THE FORM MODIFIER

ERROR=*label* ([*item-name*])	Suppress the default error return that Transact normally takes. Instead, the program branches to the statement identified by *label*, and the stack pointer for the list register is set to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register. If you do not specify an *item-name*, as in ERROR=*label*();, the list register is reset to empty. If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not changed. For more information, refer to "Automatic Error Handling," in section 5.
LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked while the PUT executes.
NOMSG	Suppress the standard error message produced by Transact as a result of a file or data base error.
RECNO= *item-name*	Place the record number of the new entry into the data register space for *item-name*.
STATUS	Suppress processor action defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of a PUT statement is described by the value in the status register:

Status Register Value	Meaning
0	The PUT operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

PUT with the STATUS option could be used in the following instance:

When a data set is full, you may want to write to an overflow file. To trap and display the full error condition, you could use the following code:

```
PUT DATA-SET,
    LIST=(A:N),
    STATUS;
IF STATUS<>0 THEN              <<Error, Check it out>>
   IF STATUS<>16 THEN          <<Unexpected error   >>
      GO TO ERROR-CLEANUP
   ELSE                        <<Write to overflow  >>
     DO                        <<Set full           >>
       PUT OVERFLOW,
           LIST=(A:N),
           STATUS;
       IF STATUS<>0 THEN
           GO TO ERROR-CLEANUP;
       DISPLAY "OVERFLOW FILE USED";
     DOEND;
```

OPTIONS AVAILABLE ONLY WITH THE FORM MODIFIER

APPEND	Append the next form to the specified form, overriding any freeze or append condition specified for the form in its FORMSPEC definition. APPEND sets the FREEZAPP field of the VPLUS comarea to 1.
CLEAR	Clear the specified form when the next form is displayed, overriding any freeze or append condition specified for the form in its FORMSPEC definition. CLEAR resets the FREEZAPP field of the VPLUS comarea to zero.
CURRENT	Use the form currently displayed on the terminal screen; that is, perform all the PUT(FORM) processing except retrieving and displaying the form. Use this option to avoid the processing that normally occurs when a new form is displayed.
FEDIT	Perform the field edits defined in the FORMSPEC definition for the form immediately before displaying it.

FKEY=*item-name*	Move the number of the function key pressed by the operator in this operation to the single word integer *item-name*. The function key number is a digit from 1 through 8 for function keys f1 through f8, or zero for the ENTER key. Transact determines which function key was pressed from the value of the field LAST-KEY in the VPLUS comarea.
F*n*=*label*	Control passes to the labelled statement if the operator presses function key *n*. *n* may have a value of 0 through 8, inclusive, where zero indicates the ENTER key. This option may be repeated as many times as necessary in a single PUT(FORM) statement.
FREEZE	Freeze the specified form on the screen and append the next form to it, overriding any freeze or append condition specified for the form in its FORMSPEC definition. FREEZE sets the FREEZAPP field of the VPLUS comarea to 2.
INIT	Initialize the fields in the displayed form to any initial values defined for the form by FORMSPEC, or perform any Init Phase processing specified for the form by FORMSPEC. PUT(FORM) performs the INIT processing before it transfers any data from the data register and before it displays the form on the screen.
WAIT[=F*n*]	Do not return control to the program until the terminal user has pressed the function key *n*. *n* may have a value of 0 through 8, where 1 through 8 indicate the keys f1 through f8 and 0 indicates the ENTER key.

If the end user presses any function key other than one requested by the WAIT option, Transact displays a message in the window and waits for the next function key to be pressed.

If F*n* is omitted, PUT(FORM) waits until any function key is pressed. If the WAIT option is omitted altogether, PUT(FORM) clears the screen and returns control to the program immediately after displaying the form with its data. For example:

```
PUT(FORM) (FORMNAME),  <<display form named in FORMNAME>>
   LIST=(A:C),
   WAIT;               <<wait for user to press any key>>
```

WINDOW= ([*field*] *message*)	Place a message in the window area of the screen and, optionally, enhance a field in the form. The enhancement is done according to the definition of the destination form in FORMSPEC. *field* and *message* can be specified as follows:

field	Either the name of the field to be enhanced, or an *item-name* within parentheses containing the name of the field to be enhanced.
message	Either a "*string*" in quotes that comprises the message to be displayed, or an *item-name* within parentheses containing the message string to be displayed in the window.

The following example illustrates this option when the field name and message are specified directly:

```
PUT(FORM) FORM1,
  LIST=(A,C,E),
  WINDOW=(A,"Press f1 if data is correct."),
  WAIT=F1;
```

In the next example, both the field and the message are specified through an item-name reference:

```
DEFINE(ITEM) ENHANCE  U(16):
             MESSAGE  U(72);

MOVE (ENHANCE) = "FIELD1";
MOVE (MESSAGE) = "This field may not be changed.";

PUT(FORM) *,      <<display current form >>
  LIST=(),
  WINDOW=((ENHANCE),(MESSAGE));
```

EXAMPLES

The following command sequence prompts for new customer information and adds this information to the customer master file:

```
$$ADD:
   $CUSTOMER:
      PROMPT CUST-NO:
             CUST-NAME:
             CUST-ADDR:
             CUST-CITY:
             CUST-STATE:
             CUST-ZIP;
      PUT CUST-MAST,
          LIST=(CUST-NO:CUST-ZIP);
```

The next example displays a header form and then appends a form with data to the header. After appending the data form 10 times, each time with new data, the program asks the user if he wants to continue. The data to be displayed is taken from the data register; the particular items determined by the LIST= option. In this example, the data in the data register is retrieved from a IMAGE data set by the FIND statement.

```
     LIST CUST-NO:
          LAST-NAME:
          FIRST-NAME:
          COUNT;

     PUT(FORM) HEADER,                   << Freeze header form on screen    >>
               LIST=(),
               FREEZE;

     LET (COUNT) = 0;

     FIND(SERIAL) CUSTOMER,               << Get data from data base        >>
          LIST=(CUST-NO:FIRST-NAME),
          PERFORM=LIST-FORM;
       .
       .
       .
   LIST-FORM:

     IF (COUNT) < 10 THEN                      << Append data form 9 times >>
       DO
         LET (COUNT) = (COUNT) + 1;
         PUT(FORM) CUSTLIST,
             LIST=(CUST-NO:FIRST-NAME),
             APPEND;
       DOEND
     ELSE
       DO
         LET (COUNT) = 0;
         PUT(FORM) CUSTLIST,                   << At 10th iteration,       >>
             LIST=(CUST-NO:FIRST-NAME),        << wait for user input      >>
             WINDOW=("Press any function key to continue"),
             APPEND,
             WAIT=;
       DOEND;

     RETURN;
```

REPEAT

Causes repeated execution of a simple or compound statement until a specified condition is true

```
*****************************************************************
*                                                               *
*  REPEAT statement UNTIL condition-clause;                     *
*                                                               *
*****************************************************************
```

When REPEAT is encountered, the simple or compound statement following it is executed and then the *condition-clause* is tested. The *condition-clause* includes a *test-variable*, a *relational-operator*, and one or more *values*. Execution of the statement following REPEAT continues until the test gives a value of true.

STATEMENT PARTS

statement	A simple or compound Transact statement may follow REPEAT. A compound statement is bracketed with a DO...DOEND pair.
condition-clause	A *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value [,*value*]...

test-variable May be one or more of the following:

(*item-name*)	The value in the data register that corresponds to *item-name*.
EXCLAMATION	Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. If the null response is set, the EXCLAMATION test variable is a positive integer; if not set, it is zero. Default is 0.
FIELD	Current status of FIELD option. If an end user qualifies a command with FIELD, the FIELD test variable is a positive integer; otherwise, it is a negative integer. Default is <0.
INPUT	The last value input in response to the INPUT prompt.
SORT	Current status of SORT option. If an end user qualifies a command with SORT, the value of the SORT test variable is a positive integer; otherwise SORT is a negative integer. Default is <0.

PRINT	Current status of PRINT or TPRINT option. If an end user qualifies a command with PRINT, the PRINT test variable is an integer greater than zero and less than 10; if a command is qualified with TPRINT, PRINT is an integer greater than 10; if neither qualifier is used, PRINT is a negative integer. Default is <0.
REPEAT	Current status of REPEAT option. If an end user qualifies a command with REPEAT, the REPEAT test variable is a positive integer; otherwise, REPEAT is a negative integer. Default is <0.
STATUS	The value of the status register set by the last data set or file operation, data entry prompt, or external procedure call.

relational-operator Specifies the relation between the *test-variable* and the *value*. It may be one of the following:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

value The value against which the *test-variable* is compared. The allowed *value* depends on the *test-variable*:

If the *test-variable* is:	Then *value* must be:
(*item-name*)	An alphanumeric string or a numeric value.
INPUT	An alphanumeric string.
EXCLAMATION FIELD PRINT REPEAT SORT	A positive or negative integer.
STATUS	An integer number.

Alphanumeric strings must be enclosed in quotation marks.

If more than one value is given, then the following are true:

- The relational-operator can be "=" only, and
- The action is taken if the *test-variable* is equal to *value* OR *value2* OR...*valuen*.

EXAMPLES

The following example performs the compound statement between the DO...DOEND pair until the value of OFFICE-CODE exceeds 49.

```
REPEAT
  DO
    GET(SERIAL) MASTER;
    .
    .
    .
    PUT SEQFILE;
  DOEND
UNTIL (OFFICE-CODE) > 49;
```

Changes the values contained in a KSAM or MPE record or an IMAGE data set entry

```
*****************************************************************
*                                                               *
*  REPLACE[(modifier)] file-name[,option-list];                 *
*                                                               *
*****************************************************************
```

REPLACE allows you to replace one or more records or entries in a file or data set. REPLACE may use the values in the update register as the new values for the updated entries. REPLACE differs from UPDATE in that it allows you to change search or sort items in an IMAGE data set as well as key items in a KSAM file, and because it can perform a series of changes to a file or data set.

Note that it will only replace key (search) items in an IMAGE manual master set if there are no detail set entries linked to that key; and it will not replace IMAGE detail set entries with search items that do not exist in manual master sets associated with that detail.

The REPLACE operation performs the following steps:

1. It retrieves a data record from the file or data set and places it in the data register area specified by the LIST= option of REPLACE, overwriting any prior data in this area.

2. It checks whether this record contains values that match any selection criteria set up in the match register. If the retrieved data does not meet the match criteria, it returns to step 1 to retrieve the next record. If the record meets the selection criteria specified in the match register, or if there are no match criteria, it first performs any PERFORM= processing; then it executes steps 3 through 5.

3. It replaces the values in the data register of the items to be updated with the values in the update register. Or, if there are no values in the update register, it uses the current values in the data register. The update register can be set up by a routine specified in PERFORM= option since the PERFORM= processing is done prior to the actual replacement. A PERFORM= routine can also be used to place new values directly into the data register.

4. It writes a new record with updated values from the data register to the file or data set, and then deletes the old record.

5. It returns to step 1 unless the end of the file or chain has been reached, or unless the SINGLE option or the CURRENT modifier has

specified replacement of a single entry only. At the end of the file or chain or if only retrieving a single entry, it goes to the next statement.

In order to use REPLACE effectively, these are the steps you must perform:

1. Specify the entries to update. Set up the key and argument registers if you will use REPLACE with no modifier or with the CHAIN or RCHAIN modifiers. Set up the match register if you want to replace particular entries when you use the CHAIN, RCHAIN, SERIAL, or RSERIAL modifiers.

   If you plan to replace a key (search) item in an IMAGE master set, first delete all chains linked to that item from associated detail sets. If you plan to replace a search item in an IMAGE detail set that is linked to a manual master set, make sure the new search item exists in the manual master.

2. Get the new values and place them in the update register or, if you are not using the update register, in the data register. Note that REPLACE always uses the values in the update register if there are any. You may get the new values from an end user with a DATA(UPDATE) or PROMPT(UPDATE) statement, or you may place them directly in the update register with a SET(UPDATE) statement. When you update multiple entries with different values, you should set up the update register in a routine identified by a PERFORM= option of the REPLACE statement; otherwise, the same items are updated with the same values in each of the multiple entries.

3. Use the REPLACE statement to replace the selected entries, or to replace all entries if no match criteria are specified. Make sure that the entire record or entry is specified in a LIST= option; otherwise, REPLACE will write null values into items not specified in the list register when it writes the updated entry back to the file or data set.

REPLACE both adds the updated record and deletes the original entry so that any data item that has not been specified in the list register will have a null value after the operation. This is why you should make sure that the list register contains every data item name in the set entry. If a chained or serial access mode is specified (multiple entry updates), the data items to be updated must have been specified in the update register by using the PROMPT, DATA, LIST, or SET statements with the UPDATE option. Transact automatically restores the original record if an error occurs during the storage operation.

REPLACE with the UPDATE option only replaces that part of the record or entry that is not a search or sort item. Unlike the other forms of REPLACE, it does not delete the original entry and replace it by a new entry. Thus, for this option, only update items, not the whole record, need be present in the list register.

STATEMENT PARTS

modifier	To specify the type of access to the data set or file, choose one of the following modifiers:
none	Update an entry in an IMAGE master set based on the key value in the argument register; this option does not use the match register. If the search item is to be changed, there must not be any entries in detail data sets linked to the old search item.
CHAIN	Update entries in an IMAGE detail set or KSAM chain based on the key value in the argument register. The entries must meet any match selection criteria in the match register. If no match criteria are specified, all entries are updated. If the search item is to be changed in a chain linked to an IMAGE manual master set, the new item must exist in the associated master set.
CURRENT	Update the last entry that was accessed from the file or data set. This modifier only replaces one entry, overriding the iterative capability of REPLACE.
DIRECT	Update the entry stored at the specified record number. Before using this modifier, you must store the record number as a doubleword integer in the item referenced by the RECNO= option.
PRIMARY	Update the IMAGE master set entry stored at the primary address of a synonym chain. The primary address is located through the key value contained in the argument register.
RCHAIN	Update entries in an IMAGE detail set chain in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.
RSERIAL	Update entries from a file in the same manner as the SERIAL option, except in reverse order. For a KSAM or MPE file, this operation is identical to SERIAL.
SERIAL	Update entries that meet any match criteria set up in the match register in a serial mode. If no match criteria are specified, all entries are updated. Note that you cannot use this modifier to replace key items in IMAGE master set.

file-name — The KSAM or MPE file or IMAGE data set to be accessed in the replace operation. If the data set is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows:

set-name(*base-name*)

option-list — One or more of the following fields, separated by commas:

ERROR=*label* ([*item-name*]) — Suppress the default error return that Transact normally takes. Instead, the program branches to the statement identified by *label*, and Transact sets the list register pointer to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register.

If you do not specify an *item-name*, as in ERROR=*label*();, the list register is reset to empty.

If you use an "*" instead of *item-name*, as in ERROR=*label*(*);, then the list register is not changed.

For more information, see "Automatic Error Handling," in section 5.

LIST= (*range-list*) — The list of items from the list register to be used for the REPLACE operation.

The options for *range-list* include the following:

(*item-name)*	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current list pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.

(*item-name1*, *item-name2*, ... *item-namen*)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM and MPE, items must be specified in the order of their occurrence in the record. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
()	A null item list. That is, delete the entry or entries, but do not retrieve any data.

LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked while the REPLACE executes. (Note that Transact always locks a data base opened in mode 1 while REPLACE executes unless the NOLOCK option is used with SET or RESET.)
NOCOUNT	Suppress the message normally generated by Transact to indicate the number of updated entries.
NOMATCH	Ignore any match criteria set up in the match register.
NOMSG	Suppress the standard error message produced by Transact as a result of a file or data base error.
PERFORM=*label*	Execute the code following the specified label for every entry retrieved by the REPLACE verb before replacing the values in the entry. The entries may be optionally selected by MATCH criteria. This option allows you to perform operations on retrieved entries without your having to code loop control logic. It is also useful for setting up the update register for the replacement. You may nest up to 10 PERFORM= options.
RECNO=*item-name*	With the DIRECT modifier: You must define *item-name* to contain the doubleword integer number of the record to be updated. With other modifiers: Transact returns the record number of the replaced record in the doubleword integer *item-name*.
SINGLE	Update only the first selected entry, and then pass to the statement following REPLACE.

SOPT	Suppress Transact optimization of IMAGE calls. This option is primarily intended to support a data base operation in a performed routine that is called recursively. The option allows a different path to the same detail data set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of an IMAGE call list of "*" after the first call is made.
STATUS	Suppress processor actions defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of a REPLACE statement is described by the value in the status register:

Status Register Value	Meaning
0	The REPLACE operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, see IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with REPLACE:

- Makes the normal multiple accesses single.
- Suppresses the normal rewind done by REPLACE, so CLOSE should be used before REPLACE(SERIAL).
- Suppresses the normal find of the chain head by REPLACE, so PATH should be used before REPLACE(CHAIN).

The following example uses marker items to declare a range. If a key item is involved, this code logs the change and uses REPLACE instead of UPDATE to make the change. (Remember that you cannot be sure which items are in a list delimited by marker items.) STATUS must be used to capture the error of attempting to update a key or sort item:

```
UPDATE DETAIL-SET,
       LIST=(MARKER1:MARKER2),
       STATUS;
IF STATUS<>0 THEN           <<Error, Check it out>>
  IF STATUS<>41 THEN        <<Unexpected         >>
    GO TO ERROR-CLEANUP
  ELSE                      <<Log and Complete Update>>
    DO
      PUT LOG-FILE,
          LIST=(MARKER1:MARKER2);
      REPLACE(CURRENT) DETAIL-SET,
          STATUS,
          LIST=(MARKER1:MARKER2);
      IF STATUS<>0 THEN
        GO TO ERROR-CLEANUP;
    DOEND;
```

UPDATE

When this option is used, REPLACE does not update search or sort items. It should be used to perform an iterative update on a data set or file where you do not want to change search or sort items. You should use this option when replacing a non-key item in an IMAGE manual master set; otherwise, a DUPLICATE KEY IN MASTER error occurs when REPLACE adds the new entry.

EXAMPLES

The first example replaces a search item value in an IMAGE master set with a new value. Before making the replacement, it makes sure that a detail set linked to the master set through CUST-NO has no entries with the value being replaced.

```
PROMPT(PATH) CUST-NO ("Enter customer number to be changed");
FIND(CHAIN) SALES-DET, LIST=();      <<Look for old number in detail set>>
IF STATUS <> 0 THEN                  << and,if chain exists, delete it. >>
 DO
   DISPLAY "Before replacing customer number, delete from SALES-DET";
   PERFORM DELETE-SALES-REC;
 DOEND;

<<No chains linked to this customer number; so continue with update    >>

LIST LAST-NAME:                      <<Set up rest of list register     >>
     FIRST-NAME:
     STREET-ADDR:
     CITY:
     STATE:
     ZIP;
REPLACE CUST-MAST;                   <<Replace specified customer number>>
     LIST=(CUST-NO:ZIP),             << with new number entered in      >>
     PERFORM=GET-NEW-NAME;           << GET-NEW-NAME routine            >>
      .
      .
      .
GET-NEW-NAME:
 DATA(UPDATE) CUST-NO ("Enter new customer number"):
 RETURN;
```

The next example changes the product number in a master set PRODUCT-MAST, and then updates the related detail entries in the associated detail set PROD-DETL. When the detail set entries have all been updated, it deletes the master entry for the old product number for PRODUCT-MAST.

```
PROMPT PROD-NO ("Enter new product number"):
       DESCRIPTION ("Enter a one-line description");
PUT PRODUCT-MAST,
    LIST=(PROD-NO:DESCRIPTION);

SET(UPDATE) LIST(PROD-NO);        <<Set up update register with new value>>

DATA(KEY) PROD-NO                 <<Set up key and argument registers    >>
     ("Enter product number to be changed");
RESET(STACK) LIST;                <<Release stack space >>
```

```
<<Now, update the product number in each entry of associated detail set >>

DISPLAY "Updating product number in PROD-DETL", LINE=2;
LIST PROD-NO:                     <<Allocate space for PROD-DETL entry    >>
     INVOICE-NO:
     QTY-SOLD:
     QTY-IN-STOCK;
REPLACE(CHAIN) PROD-DETL,              <<Replace each entry in detail set >>
    LIST=(PROD-NO:QTY-IN-STOCK);
RESET(STACK) LIST;

DELETE PRODUCT-MAST,                   <<Delete old entry from master set >>
    LIST=();
```

The following example replaces each occurrence of a non-key item, ZIP, with a new value. It asks the end user to enter the value to be replaced as a match criterion for the retrieval. Before making the replacement, it uses a PERFORM= routine to display the existing record and ask the user for a new value:

```
  LIST LAST-NAME:                                <<Set up list for update >>
       FIRST-NAME:
       STREET-ADDR:
       CITY;
  PROMPT(MATCH) ZIP ("Enter ZIP code to be replaced");
  REPLACE(SERIAL) MAIL-LIST-DETL,                <<Replace each occurrence>>
        LIST=(LAST-NAME:ZIP),                    << of specified zip code,>>
        UPDATE,                                  << a non-key item.       >>
        PERFORM=GET-ZIP;
  EXIT;

GET-ZIP:
  DISPLAY;
  DATA(UPDATE) ZIP ("Enter new ZIP code");
  RETURN;
```

RESET

Resets execution control parameters, the match or update registers, the list register stack pointer, or delimiter values

```
***************************************************************
*                                                             *
*  RESET(modifier) [target;]                                  *
*                                                             *
***************************************************************
```

The function of RESET depends on the verb's modifier, and the different modifiers determine the syntax of the statement. The allowed modifiers and the associated syntax options are:

- COMMAND Clear user responses from the input buffer (Syntax Option 1).
- DELIMITER Reset delimiter values to Transact defaults (Syntax Option 2).
- OPTION Reset various execution control parameters (Syntax Option 3).
- STACK Reset the stack pointer for the list register (Syntax Option 4).

Syntax Options

(1) RESET(COMMAND);

RESET(COMMAND) clears the input buffer, TRANIN, that contains the responses to prompts issued by a Transact program. This option is particularly useful to clear unprocessed responses from the input buffer when there is a need to reissue a prompt. Unprocessed responses can occur when the end user responds to multiple prompts with a series of responses separated by a currently defined delimiter.

For example:

```
 GET-NAME:
   DATA CUST-NO ("Please enter a customer number and name"):
        CUST-NAME;

   SET(KEY) LIST(CUST-NO);
   FIND CUST-MAST;
   IF STATUS = 0 THEN          <<CUST-NO not found >>
     DO
      DISPLAY "Invalid Customer Number. Please re-enter.";
      RESET(COMMAND);          <<Clear input buffer before returning>>
      GO TO GET-NAME;
     DOEND;
```

When the DATA prompt is issued, suppose the user response is:

```
   Please enter a customer number and name> 30335, Jones, James
```

Without the RESET(COMMAND) statement, the unprocessed response "James" would appear to Transact as a response to the CUST-NO prompt.

(2) RESET(DELIMITER);

RESET(DELIMITER) resets the delimiters used in input fields to the defaults of "," and "=".

(3) RESET(OPTION) *option-list*;

RESET(OPTION) and the one or more fields chosen from *option-list* programmatically reset any options that have already been set by means of a SET statement. Or, the OPTION modifier can reset the match and update registers.

option-list One or more of the following fields, separated by commas:

END	Resets the END option. If END or "]" or "]]" is encountered during execution, control passes to the end of sequence.
FIELD	Resets the FIELD option. The lengths of prompted-for fields are not indicated on 264X series terminals.
MATCH	Clears the MATCH register so that you can set up new match criteria.

NOHEAD	Resets the NOHEAD option. Data item headings are to be generated on any subsequent displays set up by DISPLAY or OUTPUT statements.
NOLOCK	Re-enables automatic locking disabled by a previous SET NOLOCK option.
PRINT	Resets the PRINT option. Any displays generated by the DISPLAY or OUTPUT statements are directed to the user terminal.
SORT	Resets the SORT option. Any listings generated by subsequent OUTPUT statements are not sorted before display.
SUPPRESS	Resets the SUPPRESS option. Multiple blank lines sent to the display device are not to be suppressed.
TPRINT	Resets the TPRINT option. Any displays generated by the DISPLAY or OUTPUT statements and directed to the terminal are not line printer formatted.
UPDATE	Clears the UPDATE register so you can set up new update parameters.
VPLS	Indicates to Transact that the terminal is no longer in block mode. Error messages are no longer sent to the window. (Refer to SET(OPTION) VPLS description.) If SET(OPTION) VPLS=*item-name* has been specified, you must follow this statement with a RESET(OPTION) VPLS statement. The VPLS option causes RESET to write the contents of *item-name* back to the VPLUS comarea. Only as much of the comarea as was transferred by SET(OPTION) VPLS is written back to the VPLUS comarea by RESET(OPTION) VPLS. You must not include any Transact statement that references VPLUS forms between the SET(OPTION) VPLS=*item-name* and the RESET(OPTION) VPLS statements. If you do, Transact returns to command mode and issues an error message.

(4) RESET(STACK) LIST;

RESET(STACK) resets the list register so that a new list can be generated by PROMPT and LIST statements. The contents of the data register are not touched.

EXAMPLES

```
RESET(OPTION)
     MATCH,
     UPDATE;
```

This example removes all current match criteria and item update values from the match and update registers.

```
RESET(STACK) LIST;
```

This statement resets the list register to its beginning so you can use the same area for new list items.

RETURN

Terminates a PERFORM block

```
***************************************************************
*                                                             *
*  RETURN[(level)];                                           *
*                                                             *
***************************************************************
```

RETURN transfers control from a PERFORM block to another statement. RETURN is also used to return to a data base access loop called with the PERFORM= option.

STATEMENT PARTS

none	Transfers control to the statement immediately following the last PERFORM statement executed; also used to return to data base access loop called with the PERMFORM= option.
level	Transfers control to the statement immediately following one of the previous PERFORM statements in the command sequence.

If *level* is:	then Transact:
1-128	Skips that many PERFORM levels and transfers control to the statement following the correct PERFORM statement.
@	Transfers control to the statement following the top PERFORM statement in the current command sequence. Control passes through all active perform levels to level 0.

EXAMPLES

```
MAIN:
   PERFORM A;
   EXIT;
   .
   .
A:
   PERFORM B;
   .
   .
   RETURN;
B:
   PERFORM C;
   .
   .
   RETURN;
C:
   PERFORM D;
   .
   .
   RETURN;
D:
   PERFORM E;
   .
   .
   RETURN;
E:.
  .
  .
   IF(VALUE)="SAM" THEN
     RETURN;                <<Transfer control to >>
                            <<1st statement following PERFORM E;>>
   IF(VALUE)="ALLAN" THEN
     RETURN(1);             <<Transfers control to >>
                            <<1st statement following PERFORM D;>>
   IF(VALUE)="BROWN" THEN
     THEN RETURN(@);        <<Transfers control to >>
                            <<1st statement following PERFORM A;>>
```

SET

Alters execution control parameters, sets the match, update, or key registers, sets the list register stack pointer, sets up data for subsequent display on a VPLUS form, or sets alternate delimiters

```
**************************************************************
*                                                            *
*  SET(modifier) target;                                     *
*                                                            *
**************************************************************
```

The function of SET depends on the verb's modifier, and the different modifiers determine the syntax of the statement. The allowed modifiers and the associated syntax options are:

- COMMAND Specify processor commands (Syntax Option 1).
- DELIMITER Specify processor delimiters (Syntax Option 2).
- FORM Specify data transfer to a VPLUS form buffer for subsequent display (Syntax Option 3).
- KEY Set the value of the key and argument registers (Syntax Option 4).
- MATCH Set up match selection criteria in the match register (Syntax Option 5).
- OPTION Specify various execution control parameters (Syntax Option 6).
- STACK Change the value of the stack pointer for the list register (Syntax Option 7),
- UPDATE Set the value of the update register (Syntax Option 8).

SYNTAX OPTIONS

(1) SET(COMMAND) *argument;*

SET(COMMAND) programmatically invokes command mode and performs any command identified in *argument*.

argument The commands specified in the *argument* parameter may be any of the following:

EXIT	Generates an exit from Transact; control passes to the operating system or calling program.
INITIALIZE	Generates an exit from the current program and causes Transact to prompt for a different program name, which it will then initiate.
COMMAND [(*command-label*)]	Lists the commands or subcommands defined in the currently loaded program. If a particular *command-label* is specified, it lists all the subcommands associated with that command; if no *command-label*, it lists all the commands in the program.
"*input-string*"	Specifies possible user responses to command prompts and/or to prompts issued by PROMPT, DATA, or INPUT statements. This construct allows the program to simulate user responses to prompts. This option transfers control to and executes any command sequences specified by *input-string*. The code does not return automatically to the point from which it was called.

EXAMPLES OF SET(COMMAND)

```
SET(COMMAND) COMMAND;
```

This statement lists all the commands in the current program and returns to the next statement.

```
SET(COMMAND) COMMAND(ADD);
```

This statement lists all the subcommands in the command sequence beginning with $$ADD and returns to the next statement.

```
SET(COMMAND) "REPEAT ADD ELEMENT";
```

This statement executes ADD ELEMENT until an end user enters "]" or "]]"; it then returns to command mode and issues the ">" prompt for another command.

```
SET(COMMAND) "ADD CUSTOMER";
```

This statement executes the code associated with the command/subcommand:

```
$$ADD:
   $CUSTOMER:
```

It does not return.

(2) SET(DELIMITER) "*delimiter-string*";

SET(DELIMITER) replaces the built-in processor input field delimiters ("," and "=") with the delimiter characters in the delimiter string. Note: a blank is not a valid delimiter. A maximum of eight characters can be defined as a *delimiter-string*.

For example:

If *delimiter-string* is:	Then Transact:
"#/"	recognizes the characters "#" and "/" as field delimiters.
""""	recognizes quotation marks as field delimiters.
""	recognizes no delimiters, which means the user cannot enter multiple field responses.

(3) SET(FORM) *form*[*,option-list*];

SET(FORM) is used prior to another statement that actually displays the form. It can be used to transfer data to the VPLUS form buffer for subsequent display by a GET(FORM), PUT(FORM) or UPDATE(FORM) statement. It can also be used to set up window messages and field enhancements for subsequent displays.

Used with the LIST= option, SET(FORM) allows you to initialize fields in a form with values from the data register rather than with values specified through FORMSPEC. With the inclusion of other *option-list* options, SET(FORM) also provides form sequence control for the specified form and for the next form after that form.

SET(FORM) opens the forms file, but not the terminal.

form	A form in the VPLUS forms file that is used for the subsequent display. It may be specified as one of the following:

form-name	Name of the form as defined by FORMSPEC.
(*item-name*)	Name of an item that contains the form name.
*	The form identified by the "current" form name; that is, the form name most recently specified in a Transact statement that references VPLUS forms. Note that this does not necessarily mean the form currently displayed.
&	The form identified as the "next" form name; that is, the form name defined as "NEXT FORM" in the FORMSPEC definition of the current form.

option-list	One or more of the following options, separated by commas, should be specified in a SET(FORM) statement:

NOTE: The scope of the APPEND, CLEAR, and FREEZE options is both the previous form (accessed by the last form specifiction before this SET operation) and the specified form. Therefore if the CLEAR option is used, not only will the previous form be CLEARed when the specified form is displayed, but also the specified form will be CLEARed when the next form is displayed, regardless of the FORMSPEC definitions of the two forms.

APPEND	Append the next form to the specified form, overriding any current or next form processing specified for the form in its FORMSPEC definition. APPEND sets the FREEZAPP field of the VPLUS comarea to 1.
CLEAR	Clear the specified form when the next form is displayed, overriding any freeze or append condition specified for the form in its FORMSPEC definition. CLEAR sets the FREEZAPP field of the VPLUS comarea to zero.
FEDIT	After transferring data to the form, perform any field edits specified in the FORMSPEC definition for the form.
FREEZE	Freeze the specified form on the screen when the next form is displayed, and append the next form to it. FREEZE sets the FREEZAPP field of the VPLUS comarea to 2.
INIT	Initialize the fields in the specified form to any initial values defined for the forms by FORMSPEC, or perform any Init Phase processing specified for the form by FORMSPEC.
LIST= (*range-list*)	The list of items from the list register to be transferred from the data register to the VPLUS buffer for subsequent processing. If this option is omitted, items that appear in both the list register and SYSTEM definition for the form are transferred.

The options for *range-list* and the records upon which they operate include the following:

(*item-name)*	A single item.
(*item-name1: item-name2*)	All the items from *item-name1* through *item-name2*.

	If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data is transferred.
(*item-name1:*)	The items from *item-name1* through the item indicated by the current list register pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.
(*item-name1, item-name2,* ... *item-name*n)	The items are selected from the list register. Items must be specified in the order of their occurrence in the form.
()	A null item list. Do not transfer any data.

WINDOW= ([*field,*] *message*)	Place a message in the window area of the screen and, optionally, enhance a field in the form. The enhancement is done according to the definition of the form in FORMSPEC. If the LIST=( ) option is in effect, the window message overwrites any previous window messages for the form, but the field enhancement is in addition to any field enhancement already on the form. The parameters *field* and *message* can be specified as follows:
field	Either the name of the field to be enhanced, or an *item-name* within parentheses whose data register value is the name of the field to be enhanced.
message	Either a *"string"* of characters within quotes that comprises the message to be displayed, or an *item-name* within parentheses whose data register value is the message string to be displayed in the window.

EXAMPLES OF SET(FORM)

```
SET(FORM) MENU,
   CLEAR;
```

This statement clears any prior forms from the screen when a subsequent statement displays the form MENU. If MENU is the current form, this statement clears the MENU when the next form is displayed, regardless of the value of the MENU's FREEZAPP option.

```
SET(FORM) LIST-FORM,
   LIST=(LIST-DATE),
   WINDOW=(LIST-DATE,"Only enter orders for this date");
GET(FORM) *,
   LIST=(ORDER-NO:QTY-ON-HAND);
```

This example moves a value from the data register area identified by LIST-DATE to the VPLUS buffer for subsequent display by GET(FORM). It also sets up a field to be enhanced and a message for display when GET(FORM) displays LIST-FORM.

```
PUT(FORM) (FORMNAME), FREEZE;
SET(FORM) &,
   LIST=(ITEM-A),
   WINDOW=((ITEM-A), (MESSAGE));
PUT(FORM) *,
   WAIT=F1;
```

This example is highly general. The first PUT(FORM) statement displays whatever form is identified by FORMNAME and freezes that form on the screen. SET(FORM) then specifies that the value of ITEM-A is to be displayed and enhanced in the next form and also specifies a message (MESSAGE) to be issued when the next form is displayed by the subsequent PUT(FORM) statement.

(4) SET(KEY) LIST ({*item-name*});
{   *   }

SET(KEY) sets the key and argument registers to the values associated with *item-name* in the list and data registers. Transact generates an error message at execution time if the item name cannot be found in the list register. You typically use this modifier on multiple data set operations where the necessary key value has been retrieved by a previous operation.

If an * is used as the *item-name*, the last item added to the list register is used.

EXAMPLE OF SET(KEY)

```
SET(KEY) LIST(ACCT-NO);
OUTPUT(CHAIN) ORDER-DETAIL,
   LIST=(ACCT-NO:QTY-ON-HAND);
```

This example identifies the key as the item named ACCT-NO and moves the associated value in the data register to the argument register for the subsequent data set retrieval by the OUTPUT statement.

```
(5) SET(MATCH) LIST ({item-name})[,option-list];
                    {    *     }
```

SET(MATCH) places *item-name* in the list register and then sets up a match criterion in the match register using the specified item name and its current value in the data register. The resulting match criterion is used for subsequent data set and file operations. By default, the relation between the item name and its value is equality; you can choose a value from *option-list* if the match is to be performed on a basis other than equality.

If an * is specified, the last item added to the list register is used.

You can set up as many match criteria as you desire using separate SET(MATCH) statements for each. Match criteria set up with the same item name and no option are joined by a logical OR; those set up with different item names or with one of the options shown below are joined by a logical AND. (Refer to the PROMPT(MATCH) and DATA(MATCH) descriptions for other ways to set up match criteria.)

option-list	Any one of the following options may be selected:
NE	Not equal to
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
LEADER	Matched item must begin with the input string; equivalent to the use of trailing "^" on input
SCAN	Matched item must contain the input string; equivalent to the use of trailing "^^" on input
TRAILER	Matched item must end with the input string; equivalent to the use of a leading "^" on input

EXAMPLES OF SET(MATCH)

```
   LET (QTY-ON-HAND) = 10;
   SET(MATCH) LIST (QTY-ON-HAND), LT;
```

This sets up the match register with the selection criterion:

```
   +-------------+
   | QTY-ON-HAND |
   |  less than  |
   |     10      |
   +-------------+
```

```
   MOVE (STATE) = "CA";
   SET(MATCH) LIST(STATE);
   MOVE (STATE) = "NM";
   SET(MATCH) LIST(STATE);
   LET (DATE) = 010182;
   SET(MATCH) LIST(DATE), GE;
```

These statements set up the match register with the selection criteria shown below. Note that criteria with the same item name are joined by a logical OR, those with a different name by a logical AND. These criteria select entries whose value for STATE is either CA or NM and whose value for DATE is 010182.

```
   +---------------------------------------------+
   |   STATE          STATE          DATE        |
   |  equal to   OR  equal to  AND  greater than |
   |   "CA"           "NM"          010182       |
   +---------------------------------------------+
```

(6) SET(OPTION) *option-list*;

SET(OPTION) and one or more option fields included in *option-list* programmatically set the Transact command options or override default execution parameters. The options in *option-list* are separated by commas.

option-list	Select one or more of the following options:
DEPTH=*number*	Sets the terminal display area depth to a line count of *number*. The default value is 22. The depth value defines how many lines are displayed on the terminal before Transact automatically generates the prompt

"CONTINUE(Y/N)?". This option allows the video terminal user to view a listing in a controlled page mode. If *number* is 0, information is displayed continuously on the terminal, with no generation of the "CONTINUE (Y/N)?" prompt.

END=*label*	Transact branches to the statement marked *label* if an end of sequence is encountered, either by an explicit or implicit END or by "]" or "]]" input in response to a prompt at execution time. This control function can be re-assigned to a different *label* or reset at any point in the program logic.
FIELD[="*ab*"]	Enhance or change the prompts for data item fields on the terminal display. By default, an item name prompt issued by a PROMPT or DATA statement shows the item name followed by the character ">".

The parameters *a* and *b* specify alternate display options, where *a* specifies the leading prompt character, *b* specifies the trailing prompt character. If a is a caret, "^", then the leading prompt character is suppressed.

If both *a* and *b* are omitted, the FIELD option encloses the response field with the delimiters ">" and "<".

This option has the same effect as the FIELD command qualifier (see section 5).

```
If the statement is:      then, the prompt is:

SET(OPTION) FIELD;         NAME> field-length <
SET(OPTION) FIELD=":";     NAME:
SET(OPTION) FIELD="^";     NAME
SET(OPTION) FIELD="[]";    NAME[ field-length ]
```

Note that the cursor is positioned in the second character position following the left delimiter. If no delimiter is used, the cursor is positioned in the second character position following the field name.

Normally *b* sets the trailing prompt character to its value; however, if *b* is one of the characters "A" through "O" or "@", entry fields are enhanced as described in the 262X or 264X terminal user handbooks.

For example:

```
  SET(OPTION) FIELD= " J";
```

This statement enhances the response field with half-bright inverse video.

HEAD	Generates headings for the next DISPLAY verb encountered with the TABLE option, regardless of page position.
LEFT	Left justifies data items for any subsequent displays set up by the DISPLAY or OUTPUT statements. Since this is the default option, it is normally used to reset justification after a SET(OPTION) RIGHT, or ZEROS statement.
NOBANNER	Suppresses the default page banner containing date, time, and page number on any subsequent displays set up by the DISPLAY or OUTPUT statements. The default printer page depth then becomes 60.
NOHEAD	Suppresses data item headings on any subsequent displays set up by the DISPLAY or OUTPUT statements.
NOLOCK	Disables the automatic locking of a data base opened in mode 1 for a DELETE, PUT, REPLACE, or UPDATE operation. NOLOCK does not reset the LOCK option specified with a data base access verb (DELETE, FIND, GET, OUTPUT, PUT, REPLACE, or UPDATE). Use NOLOCK when you want to set up data set or data item locks through a PROC statement.
PALIGN=*number*	Right justifies the prompts on a display device to column *number* on the display screen.
PDEPTH=*number*	Sets the printer page depth to a line count of *number*. The default value is 58 unless the NOBANNER option is specified, in which case the default value is 60. If *number* is 0, the page heading is suppressed on any subsequent displays directed to the printer.
PRINT	Sets the PRINT option. Any displays generated by the DISPLAY or OUTPUT statements are directed to the line printer instead of to the user terminal. This option has the same effect as the PRINT command qualifier (see section 5).

You can redirect results to the printer immediately by using this option before issuing a DISPLAY or OUTPUT statement, and then closing the print file with a CLOSE $PRINT statement. For example:

```
SET(OPTION) PRINT;
DISPLAY "PRINT THIS NOW";
CLOSE $PRINT;
```

PROMPT=*number*	Sets the line feed count between prompts issued by the PROMPT, DATA, or INPUT statements to *number*. The default value is 1.
PWIDTH=*number*	Sets the printer line width to a character count of *number*. The default value is 132.
REPEAT	Sets the REPEAT option. At execution time, Transact repeats the associated statement sequence until the end user enters one of the following special characters:] Terminate execution of the current command sequence, and pass control to the first statement in the sequence.]] Terminate repeated execution of this command sequence and pass control to command mode. The end user can enter "REPEAT" and then a command name during execution to control a loop. This option has the same effect as the REPEAT command qualifier. Information on this procedure is contained in section 5, "Command Qualifiers."
RIGHT	Right justifies data item values for any subsequent displays set up by the DISPLAY or OUTPUT statements.
SORT	Sets the SORT option. Any listing generated by subsequent OUTPUT statements is sorted before display. The sort is performed in the order that the display fields appear in the list register. This option has the same effect as the SORT command qualifier (see section 5).
SUPPRESS	Suppress blank lines of data; only the first of a series of blank lines is sent to the line printer.
TABLE	Right justifies numeric fields and left justifies alphabetic fields for display.
TPRINT	Sets the TPRINT option. Any displays generated by the DISPLAY or OUTPUT statements and directed to the terminal are line printer formatted. This option has the same effect as the TPRINT command qualifier (see section 5).
VPLS=*item-name*	Informs Transact that you want to reference the VPLUS comarea directly. It directs error messages to the window, and moves the VPLUS comarea to the area in the data register itentified by *item-name*.

Item-name is the name of a data field containing all or part of the VPLUS comarea, depending on the size of the specified item. When this option is used as much of the current VPLUS comarea as will fit in the specified item is moved to the data register area associated with that item. You may then examine or change comarea fields.

A SET(OPTION) VPLS statement must always be followed by a RESET(OPTION) VPLS statement. You must not use a Transact statement that references VPLUS forms between a SET(OPTION) VPLS and a RESET(OPTION) VPLS statement. If you do, Transact returns to command mode and issues an error message.

If you plan to open the forms file and terminal with PROC statements, you should use a SET(OPTION) VPLS statement just before you place the terminal in blockmode with a call to VOPENTERM. Reset with a RESET(OPTION) VPLS statement following the call to VCLOSETERM to return the terminal to character mode. If you do not call VOPENTERM or VCLOSETERM directly, or if you do not plan to reference the comarea directly, you need not use SET(OPTION) VPLS. Instead, in these cases, use the VCOM parameter of the PROC statement (see the PROC verb description.)

If the VPLUS form is already open, you can use this option in conjunction with a RESET(OPTION) VPLS statement to retrieve or change comarea values.

For example, you could change the window enhancement in the VPLUS comarea:

```
DEFINE(ITEM) COMAREA    X(16):  <<1st 8 words, comarea>>
             WINDOW-ENH X(1)    <<right byte of word 8>>
              = COMAREA(16);
LIST COMAREA;
    .
    .
UPDATE(FORM) *;
SET(OPTION) VPLS=COMAREA;
MOVE (WINDOW-ENH)="K";      <<half-bright, inverse video>>
RESET(OPTION) VPLS;
```

WIDTH=*number*	Sets the terminal line width to a character count of *number*. The default value is 79.
ZERO[E]S	Right justifies numeric data item values and inserts leading zeros for any subsequent displays set up by the DISPLAY or OUTPUT statements.

EXAMPLES OF SET(OPTION)

```
SET(OPTION) PALIGN=25,PROMPT=2;
```

This statement aligns the prompt character on column 25, with two blank lines between the prompt lines.

```
SET(OPTION) NOHEAD,SORT,DEPTH=0;
```

This statement sorts subsequent OUTPUT listings to the terminal. It suppresses item headings and suppresses the usually automatic "CONTINUE (Y/N)?" prompt.

```
(7) SET(STACK) LIST ({item-name});
                    {    *    }
```

SET(STACK) moves the stack pointer for the list register from the current position to the one identified by *item-name*. Transact generates an error at execution time if it cannot find the data item in the list register.

Transact begins the search at the data item prior to the current (last) one in the list register and performs a reverse scan to the beginning of the list. The scan does not move the pointer, however. The stack pointer is moved only when the search finds the first occurrence of the data item. The stack pointer will not be moved if *item-name* is the current data item and it occurs only once in the list register.

When a data item has more than one appearance in the list register, each occurrence can be located by using additional SET(STACK) statements.

You typically use SET(STACK) to manipulate the list register for more than one file or data set operation or to redefine the data register contents. You may choose to redefine the data register contents for the following reasons:

- o To transfer values from one data item to another in a different set,
- o To access subfields of a data item by adding several item names in place of the original item name, or
- o To manipulate data item arrays.

EXAMPLES OF SET(STACK)

To move the stack pointer for the list register from the current data item to the item immediately prior to it, use the following format:

```
SET(STACK) LIST(*);
```

NOTE: When the stack pointer moves down the list register, the items above the new current item are removed from the list register.

```
SET(STACK) LIST (PROD-NO);
```

This statement moves the stack pointer back to the item PROD-NO. If PROD-NO appears more than once in the list register, the pointer is set to the first occurrence of this item going back down the list; that is, the item nearest the top of the list register stack.

(8) SET(UPDATE) LIST({*item-name*});

```
                     {    *    }
```

SET(UPDATE) specifies that the *item-name* in the list register and the current value for *item-name* in the data register are to be placed in the update register for a subsequent file or data set operation using the REPLACE verb. If * is used as the item name, the current item name is used.

SYSTEM

Names the Transact program and any data bases, files, or forms that are used by the program

```
****************************************************************
*                                                              *
*  SYSTEM program-name[,definition-list];                      *
*                                                              *
****************************************************************
```

The SYSTEM statement names the program and describes data bases, files, or forms files that the program uses. It overrides the default space allocations that Transact uses. It must be the first statement in the program.

STATEMENT PARTS

program-name A 1 to 6 character string of letters or digits that names the program. Transact stores the output from the compiler in a file called "IPxxxxxx" where "xxxxxx" is the program name. *program-name* is also used to call up the program for execution when the user enters it in response to Transact's "SYSTEM NAME>" prompt.

definition-list Description of the files or data sets used during execution. Each group of fields describes a file. Within the group, the fields can be in any order and separated by commas.

BANNER="*text*" Causes the text string to be placed at the top left position on every page of line printer output generated during execution of the program.

BASE=*base-name1*[(["*password*"][,*mode*])]
 [,*base-name2*[(["*password*"][,*mode*])]]...

base-name The name of an IMAGE data base used in the program. This data base has the attributes described in the IMAGE/3000 Reference Manual. *base-name1* is termed the "home base" and any references in the program to this data base need not include a base qualifier.

The BASE description opens the IMAGE data base. The home base can be opened a second time by repeating its name in the data base list in the SYSTEM statement. This feature allows two independent and concurrent access paths to the same detail data set without losing path position in either access. This might be necessary for a secondary access of a detail set during processing of a primary access path in the same set.

Set references to bases other than the home base must be qualified by including the name of the data base in parentheses following the set name:

set-name(*base-name*)

password	Used by Transact for opening the data base. If no password is provided, then at execution time Transact prompts with PASSWORD FOR *databasename*> If the user enters an incorrect password, Transact issues an error message and then reprompts for the password.
mode	Used by Transact for opening the data base. This specification overrides any mode given by the user at execution time in response to the "SYSTEM NAME>" prompt. Default=1. For example, to specify the data base STORE to be opened with the password "MANAGER" in mode 1:

```
SYSTEM MYPROG,
   BASE=STORE("MANAGER",1);
```

DATA=*data-length, data-count*

The DATA= specifications given in a main program establish the data register used by all called programs and take precedence over any DATA= specifications in called programs.

data-length	The maximum word size of the data register. Default=1024.
data-count	The maximum number of entries allowed in the list register. Default=128.

```
FILE=file-name1
         [(access[(file-option-list)]
         [,record-length[,blocking-factor
         [,file-size[,extents[,initial-allocation
         [,file-code]]]]]]))]
    [,file-name2...]
```

file-name	The MPE file name assigned or to be assigned to the file. A back-referenced file name using a leading "*" is permitted.
access	One of the following access modes: READ, WRITE, SAVE, APPEND, R/W (read/write), UPDATE, SORT. SORT is identical to R/W with the additional SORT capability. In other words, an end-of-file is automatically written into the file before the SORT, and the file is rewound following the SORT. The default is READ.
file-option-list	Any of the following fields provided that they do not conflict in meaning: OLD, NEW, TEMP, $STDLIST, $NEWPASS, $OLDPASS, $STDIN, $STDINDX, $NULL, ASCII, CCTL, SHARE, LOCK, NOFILE. See FOPEN in *MPE Intrinsics Manual* for a full explanation of these options and terms. The default is OLD (old file), binary, no carriage control, and file equation permitted.
record-length	Record length of records in file. A positive value indicates words, a negative value indicates bytes. Default=byte length required by file operation
blocking-factor	Blocking factor used to block records. Default=1 record/block
file-size	Size of the file in records. Default=10000 records
extents	Number of extents used by file. Default=10 extents
initial-allocation	Initial allocation of extents Default=1 extent
file-code	MPE file code for the file. Default=0

For example, to define a file with Read/Write access, 40 words per record, a blocking factor of 3 words per record, and a total of 100 records:

```
SYSTEM FREC,
   FILE=WORK((R/W),40,3,100);
```

In an MPE file or a KSAM file, you can then define the entire record as a parent item, and define individual fields as child items. This allows you to access the entire record by its parent name, and also refer to individual fields. For example:

```
DEFINE(ITEM) RECORD X(80):
             ITEM1  X(25) = RECORD(1):
             ITEM2  X(30) = RECORD(26):
             ITEM3  X(15) = RECORD(56):
             ITEM4  X(10) = RECORD(71);

LIST RECORD;

GET(SERIAL) WORK,
   LIST=(RECORD);
DISPLAY ITEM1: ITEM2: ITEM3: ITEM4;
DATA(SET) ITEM1: ITEM2: ITEM3: ITEM4;
```

KSAM=*file-name1*
          [(*access* [(*file-option-list*)])]
   [,*file-name2* ...]

file-name	Name of a KSAM data file.
access	One of the following access modes: READ, WRITE, R/W (read/write), UPDATE, SAVE, APPEND. Default=READ
file-option-list	Any of the following fields provided that they do not conflict in meaning: OLD, $STDLIST, $NEWPASS, $OLDPASS, $STDIN, $STDINDX, $NULL, ASCII, CCTL, SHARE, LOCK, NOFILE. See FOPEN in *KSAM/3000 Reference Manual* for a full explanation of these options and terms. Default = OLD (old file), binary, no carriage control, and file equation permitted.

OPTION=*option-list* — Either enable or disable the test facility for this program execution.

TEST — Enables the TEST command during execution of the program.

NOTEST — Disables the TEST command during execution of the program.

Default=TEST

SIGNON="*text*" — Causes the text string to be displayed as a sign-on message each time the program is executed. For example:

```
SYSTEM MYPROG,
  SIGNON="Test Version of MYPROG A02.31";
```

SORT=*number* — Specifies the number of records in the sort file. Default = 10,000

VPLS=*file-name1*[(*form-name1*[(*item-list1*)]...)]...
      [,*file-name2*[(...)]...]...

file-name — The name of a VPLUS forms file that is used in the program. Every forms file referenced in a Transact program must be specified in the SYSTEM statement.

form-name — The name of a form defined within the VPLUS forms file. If omitted, the dictionary definitions of all the forms in the specified forms file are used.

For example, if forms file CUSTFORM has a dictionary definition, you may specify:

```
SYSTEM MYPROG,
   VPLS=CUSTFORM;
```

If not, you must name each form in the forms file. For example, assuming CUSTFORM has three forms, MENU, FORM1, and FORM2; MENU has no fields, FORM1 has 3 fields, and FORM3 has 4 fields:

```
SYSTEM MYPROG,
   VPLS=CUSTFORM(MENU(),
                 FORM1(F1,F2,F3),
                 FORM2(F4,F5,F6,F7));
```

item-list — A list of item names used in the program, in the order in which they appear on the VPLUS form, which is in a left to right and top to bottom direction. The names need not be the same as the names specified for the fields by FORMSPEC, but the items must have the same display lengths as the fields. If omitted, the dictionary definitions of all the fields in the specified form are used.

For example, suppose the fields in FORM2 are defined in the dictionary:

```
SYSTEM MYPROG,
   VPLS=CUSTFORM
       (MENU(),
        FORM1(F1,F2,F3),
        FORM2);
```

WORK=*work-length*,*work-count*

work-length	The maximum word size of the work area containing the match, update, and input registers. This work area is used by Transact to set up temporary values used during execution of the program. Default=256
work-count	The maximum number of entries allowed in the work area. Default=64

UPDATE

Modifies a single entry in a KSAM or MPE file or in an IMAGE data set, or modifies a VPLUS form.

```
*************************************************************
*                                                           *
*  UPDATE[(FORM)] destination[,option-list];                *
*                                                           *
*************************************************************
```

UPDATE modifies data items that are not key items in an IMAGE master or detail set entry or in a KSAM or MPE record. The item to be updated must have been retrieved by a prior FIND or GET statement. When used with the FORM modifier, UPDATE modifies and redisplays a currently displayed VPLUS form.

The UPDATE verb does not use the update register. The new value must be placed in the data register before UPDATE is executed. The value may be retrieved from an end user, or from a data set or file,

To update a non-key value with UPDATE, perform the following steps:

1. Fetch the record or entry to update and place it in the data register. You may do this with a GET or FIND statement. If you want to update several entries, updating the same item in each entry with a different value, use a FIND statement with a PERFORM= option that calls a routine containing the UPDATE statement. If you want to update a single entry, use a GET statement.

2. Place the new value in the data register. You can get the new value from a data set or file, or from an end user. If you are getting a value from the end user, PROMPT(SET) or DATA(SET) statement is useful since it allows the end user to choose whether to leave an existing value in the data register or enter a new value.

3. Use the UPDATE statement to write the new values to the entry or record. Since UPDATE always updates the last entry retrieved, it needs no access modifiers. You must include the names of any items to be updated in a LIST=option.

Note that if you want to update several entries, updating the same data item in each entry with the same value, you should use the REPLACE statement rather than the UPDATE statement. (Refer to REPLACE verb description.)

STATEMENT PARTS

FORM	Optional. If FORM is specified, then this verb transfers data from the data register to a VPLUS form displayed at a 262X or 264X series terminal by PUT(FORM) or GET(FORM). If the requested form is not currently displayed on the terminal, an error results.
destination	The name of a file, data set, or form to be accessed in the update operation. If *destination* identifies a data set that is not in the home base as defined in the SYSTEM statement, the base name must be specified in parentheses as follows: *set-name*(*base-name*) In an UPDATE(FORM) statement, the destination must identify a form in a forms file that was named in the SYSTEM statement. For UPDATE(FORM), *destination* may be specified as any of the following:
form-name	Name of a form to be updated by UPDATE(FORM).
(*item-name*)	Name of an item whose associated data register location contains the name of the form to be updated by UPDATE(FORM).
*	The form identified by the "current" form name; that is, the form name most recently specified in a statement that references a VPLUS form. Note that this does not necessarily mean the form currently displayed.
&	The form identified as the "next" form name; that is, the form name specified as the "NEXT FORM" in the FORMSPEC definition of the current form.
option-list	The LIST option is available with or without the FORM modifier. Other options, described below, may be used only with or only without the FORM modifier.
LIST= (*range-list*)	The list of items from the list register to be used for the UPDATE operation. If the LIST= option is omitted, UPDATE uses all the items in the current list register for a file or data set update; for a form update, it uses all the items in the list register corresponding to fields in the VPLUS form definition. The options for *range-list* and the records they update include the following:

(*item-name*)	A single item.
(*item-name1*: *item-name2*)	All the items from *item-name1* through *item-name2*. If *item-name1* and *item-name2* are marker items (see DEFINE(ITEM) verb), and if there are no items between the two in the list register, no data base access is performed.
(*item-name1*:)	The items from *item-name1* through the item indicated by the current stack pointer.
(:*item-name2*)	The items from the beginning of the list register through *item-name2*.
(*item-name1*, *item-name2*, ... *item-name*n)	The items are selected from the list register. For IMAGE, items can be specified in any order. For KSAM, VPLUS, and MPE, items must be specified in the order of occurrence in the record or form. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
()	A null item list. That is, access the file or data set or display the form, but do not transfer data.

OPTIONS AVAILABLE WITHOUT THE FORM MODIFIER

ERROR=*label* ([*item-name*])	Suppress the default error return that Transact normally takes. Instead, the program branches to the statement identified by *label*, and Transact sets the list register pointer to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register. If you specify no *item-name*, as in ERROR=*label*();, the list register is reset to empty. If you use an "*" instead of *item- name* as in ERROR=*label*(*);, then the list register is not changed. For more information, see the section entitled "Automatic Error Handling," in section 5.

LOCK	Lock the specified file or data base unconditionally. If a data set is being accessed, the entire data base is locked while the UPDATE executes. (Note that Transact always locks a data base opened in mode 1 while UPDATE executes unless the NOLOCK option is used with SET.
NOMSG	The standard error message produced by Transact as a result of a file or data base error is to be suppressed.
STATUS	Suppress processor actions defined in section 5 under "Automatic Error Handling". You will probably have to add coding if you use this option.

When STATUS is specified, the effect of an UPDATE statement is described by the value in the status register:

Status Register Value	Meaning
0	The UPDATE operation was successful.
-1	A KSAM or MPE end-of-file condition occurred.
>0	For a description of the condition that occurred, refer to IMAGE condition word or MPE/KSAM file system error documentation corresponding to the value.

The following example uses marker items to declare a range. If a key item is involved, you should log the attempt. STATUS must be used to capture the error of attempting to update a key or sort item:

```
UPDATE DETAIL-SET,
       LIST=(MARKER1:MARKER2),
       STATUS;
IF STATUS<> 0 THEN           <<Error, Check it out>>
  IF STATUS <> 41 THEN       <<Unexpected error   >>
     GO TO ERROR-CLEANUP
  ELSE                       <<Log and complete update>>
     DO
       PUT LOG-FILE,
           LIST=(MARKER1:MARKER2);
       DISPLAY "key update attempted";
     DOEND;
```

OPTIONS AVAILABLE ONLY WITH THE FORM MODIFIER

APPEND	Append the next form to the specified form, overriding any freeze or append condition specified for the form in its FORMSPEC definition. APPEND sets the FREEZAPP field of the VPLUS comarea to 1.
CLEAR	Clear the specified form when the next form is displayed, overriding any freeze or append condition specified for the form in its FORMSPEC definition. CLEAR sets the FREEZAPP field of the VPLUS comarea to zero.
FEDIT	Perform any field edits defined in the FORMSPEC definition immediately before redisplaying the form.
FKEY=*item-name*	Move the number of the function key pressed by the operator in this operation to the single word integer *item-name*. The function key number is a digit from 1 through 8 for function keys f1 through f8, or zero for the ENTER key. Transact determines which function key was pressed from the value of the field LAST-KEY in the VPLUS comarea.
F*n*=*label*	Control passes to the labelled statement if the operator presses function key *n*. *n* may have a value of 0 through 8, inclusive, where zero indicates the ENTER key. This option may be repeated as many times as are necessary in a single UPDATE(FORM) statement.
FREEZE	Freeze the specified form on the screen and append the next form to it, overriding any freeze or append condition specified for the form in its FORMSPEC definition. FREEZE sets the FREEZAPP field of the VPLUS comarea to 2.
INIT	Initialize the fields in a VPLUS form to values defined by the forms design utility FORMSPEC and perform any Init Phase processing before transferring data.
WAIT[=F*n*]	Do not return control to the program until the terminal user has pressed function key *n*. *n* may have a value of 0 through 8, where 1 through 8 indicate the keys f1 through f8 and 0 indicates the ENTER key. If the end user presses a different function key, Transact sends a message to the window saying which key is expected. If F*n* is omitted, then UPDATE(FORM) waits until any function key is pressed.

WINDOW= ([*field*,] *message*	Place the message *string* in the window area of the screen and, optionally, enhance a field on the form. The enhancement is done according to the enhancements specified for the form by FORMSPEC. *field* and *message* can be specified as follows:
field	Either the name of the field to be enhanced, or an *item-name* within parentheses containing the name of the field to be enhanced.
message	Either a "*string*" within quotes that comprises the message to be displayed, or an *item-name* within parentheses containing the message string to be displayed in the window.

EXAMPLES

```
PROMPT(PATH) INV-NMBR ("INVOICE NUMBER");
PROMPT(MATCH) ITEM-NUM ("ITEM NUMBER");
LIST ITEM-QTY;
GET(CHAIN) ORDER-LINE,
   LIST=(ITEM-QTY);
DISPLAY;
DATA(SET) ITEM-QTY
   ("Enter new quantity or press return to keep old quantity");
UPDATE ORDER-LINE,
   LIST=(ITEM-QTY);
```

This example prompts the user for the values required to find a record. After it is retrieved, the user is prompted for the new quantity for the item and the record is updated. Note that the LIST= option for both the retrieval and the update only need specify the item to be updated.

The next example is similar, except that it allows the end user to update all the entries in a chain, rather than a single entry.

```
   PROMPT(PATH) INV-NMBR ("INVOICE NUMBER");
   PROMPT(MATCH) ITEM-NUM ("ITEM NUMBER");
   LIST ITEM-QTY;
   FIND(CHAIN) ORDER-LINE,
      LIST=(ITEM-QTY),
      PERFORM=UPDATE-QTY;
      .
      .
      .
  UPDATE:QTY:
   DISPLAY;
   DATA(SET) ITEM-QTY
      ("Enter new quantity or press return to keep old quantity");
   UPDATE ORDER-LINE,
      LIST=(ITEM-QTY);
   RETURN;
```

The next example uses an UPDATE(FORM) statement to update the current form. It highlights the item identified in FIELD-ENH and sends the message contained in WINDOW-MSG to the window area of the form:

```
   DEFINE(ITEM) FIELD-ENH   U(16):    <<contains name of field in VPLUS form>>
                WINDOW-MSG  U(72);    <<contains message for VPLUS window   >>
   .
   .
   MOVE (FIELD-ENH) = "FIELD1";
   MOVE (WINDOW-MSG) = "This field must be numeric";
   .
   .
   UPDATE(FORM) *,
      WINDOW=((FIELD-ENH),
              (WINDOW-MSG));
```

In this particular case, as a result of the prior MOVE statements, the UPDATE statement highlights FIELD1 in the current form and displays the message "This field is numeric" in the window area of that form.

Repeatedly tests a condition clause and executes a simple or compound statement while the condition is true

```
*****************************************************************
*                                                               *
*  WHILE condition-clause statement;                            *
*                                                               *
*****************************************************************
```

WHILE causes Transact to test a *condition-clause*. The condition clause includes the *test-variable*, the *relational-operator*, and the *value*. If the result of that test is true, then the *statement* following the condition is executed. Then the condition clause is tested again and the process repeated while the result of the test is true. When the result of the test is false, control passes to the statement following the WHILE *statement*.

STATEMENT PARTS

condition-clause	A *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value[,*value*]...

test-variable	The value to be tested; it may be one or more of the following:

(*item-name*)	The value in the data register that corresponds to *item-name*.
EXCLAMATION	Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. If the null response is set, the EXCLAMATION test variable is a positive integer; if not set, it is zero. Default is 0.
FIELD	Current status of FIELD option. If an end user qualifies a command with FIELD, the FIELD test variable is a positive integer; otherwise, it is a negative integer. Default is <0.
INPUT	The last value input in response to the INPUT prompt.

PRINT	Current status of PRINT or TPRINT option. If an end user qualifies a command with PRINT, the PRINT test variable is an integer greater than zero and less than 10; if a command is qualified with TPRINT, PRINT is an integer greater than 10; if neither qualifier is used, PRINT is a negative integer. Default is <0.
REPEAT	Current status of REPEAT option. If an end user qualifies a command with REPEAT, the REPEAT test variable is a positive integer; otherwise, REPEAT is a negative integer. Default is <0.
SORT	Current status of SORT option. If an end user qualifies a command with SORT, the value of the SORT test variable is a positive integer; otherwise SORT is a negative integer. Default is <0.
STATUS	The value of the status register set by the last data set or file operation, data entry prompt, or external procedure call.

relational-operator Specifies the relation between the *test-variable* and the values; it may be one of the following:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

value The value against which the variable is tested;

If *test-variable* is:	then *value* must be:
(*item-name*)	An alphanumeric string or a numeric value.
INPUT	An alphanumeric string.
EXCLAMATION FIELD PRINT REPEAT SORT	A positive or negative integer.
STATUS	An integer number.

Alphanumeric strings must be enclosed in quotation marks.

If more than one value is given, then the following are true:

- The relational-operator can be "=" only, and
- The action is taken if the *test-variable* is equal to *value1* OR *value2* OR...*valuen*.

statement A simple or a compound Transact statement. A compound statement must be bracketed with a DO...DOEND pair, as the example illustrates.

EXAMPLES

```
WHILE (SUB-TOTAL) >= 0
  DO
   GET(CHAIN) ORDERS;
    .
    .
    .
   LET (SUB-TOTAL)=(SUB-TOTAL) - (OUT-BAL);
  DOEND;
```

TRANSACT TEST FACILITY

SECTION VII

The Transact test facility enables you to trace a program through execution for program debugging. The format for the TEST command is:

```
TEST [numeric-parameter [,[segment1.]starting-instruction-address]
        [,[segment2.]ending-instruction-address]]
```

STATEMENT PARTS

numeric-parameter	An integer number that specifies the particular test mode. The specific test modes are described in Table 7-1.
segment1	Segment number where test should begin. If none is given, the root segment (segment 0) is assumed.
starting-instruction-address	Instruction address where the trace should begin. This address is the same as the *internal locati shown in the* compiler listing produced when a Transact program is compiled with the LIST option.
segment2	Segment number where test should stop. If none is given, segment 0 is assumed.
ending-instruction-address	Instruction address where trace should end; as with the *starting-instruction-address*, this is the *internal-location* shown on a compiler listing.

To use the test facility, enter the TEST command with a numeric parameter anytime you are in command mode. The test facility stays in effect until you enter the TEST command with no numeric parameter.

For example, if you are in command mode and want to execute all subsequent code in test mode 25, enter the following command:

```
>TEST 25
```

If you want to use test mode only between instructions 0 and 8 of the root segment, enter the following command:

```
>TEST 25,0,8
```

You terminate test mode as follows:

```
   >TEST
```

If you want to use the test facility during execution of a program, when you are NOT in command mode, you must take the following steps:

- Press CNTL-Y to enter command mode,
- Enter the appropriate TEST command, and
- Enter the command RESUME.

Execution of the program continues from where it stopped, with the specified test facility.

You may enter the test parameters (without the keyword TEST) following the program name and mode parameter in response to Transact's SYSTEM NAME> prompt.

For example, to execute in test mode 25 between instructions 0 and 8 of the root segment of program MYPROG, enter the following response to SYSTEM NAME>:

```
                         mode parameter omitted
                        /
   SYSTEM NAME> MYPROG,,25,0,8
```

OUTPUT FROM TEST

Normally the output from TEST is sent to the file TRANOUT; this is your terminal in a session or the line printer in a batch job. If you want to change the test destination, you can precede the test numeric parameter with a minus sign. Then the output goes to TRANDUMP (the formal-file-designator for the destination of the test mode output). TRANDUMP is assigned to the line printer by default. You may change the assignment of TRANDUMP to a different device through a file equation.

For example, if you are executing in a command sequence, you can direct the test mode to the line printer simply by preceding the mode with a minus sign:

```
   >TEST -25            <---to request test mode 25 with output to TRANDUMP
```

If you are not in a command sequence, you can accomplish the same results as follows:

```
   CONTROL-Y BREAK      <---to stop execution
   TEST -25             <---to request test mode 25 with output to TRANDUMP
   RESUME               <---enter command to resume execution
```

You could also direct the test mode output to a disc file you create for that purpose. For example, to send the test output to the file TEST:

```
:BUILD TEST; REC=-80,,F,ASCII    <---create a file for test output
:FILE TRANDUMP=TEST              <---equate file TRANDUMP with file TEST
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> MYPROG,,-25         <---send test output to TRANDUMP(=TEST)
```

Another method is to defer test mode by setting the output priority for TRANDUMP to 1. For example:

```
:FILE TRANDUMP; DEV=,1           <---defer test mode output
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> MYPROG,,-25
```

After executing VTEST, you can run SPOOK.PUB.SYS to examine the test mode information saved in a spool file.

If you use test mode for statements that access a VPLUS forms file, you should either direct the test output to a terminal other than the one where the VPLUS forms are displayed, or else direct the forms to a different terminal. Otherwise, the test output will appear on the terminal screen with the forms. You could also defer test output as shown above.

For example, you can direct the test output to another terminal whose logical device number is 19 as shown:

```
:FILE TRANDUMP; DEV=019    <---direct test output to ldev 19
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> VTEST,,-34    <---run VTEST in mode 34; output to TRANDUMP
```

An alternative is to direct the VPLUS forms to another terminal, while the test results are sent to your terminal. To redirect the VPLUS forms, use the TRANVPLS formal file designator:

```
:FILE TRANVPLS; DEV=019    <---direct VPLUS forms to ldev 19
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> VTEST,,34     <---run your program with test mode 34
```

Now, test mode and character mode output appears at your terminal, but the VPLUS forms appear on another terminal identified by its logical device number.

TEST PARAMETERS

Table 7-1 lists the allowed test parameters and their functions.

Table 7-1. Numeric Parameters for Test Facility

Parameter	Function
(none)	Switch off existing test mode.
1	Display data block with information about the file or data base operations only if an error occurs.
2	Display each instruction address and the compiler code at that address.
3	Display each instruction address, the compiler code at that address, the space used by the list and data registers, and the amount of remaining processor work space.
4	Display each instruction address, the compiler code at that address, the instruction timings, and the HP3000 data stack pointers Z, S, Q, and DL.
22	Display each instruction address, the compiler code at that address, and data block for any instructions that perform data base and file operations. The data block includes the values and offsets of items in the key and argument registers used by the data base or file operation.
23	Display the instruction address, the compiler code at that address, and the data block for any instructions that perform data base or file operations. The display follows a multiple record operation. The data block includes the values and offsets of items in the list, data, key, argument, match, and update registers specifically used by the data base or file operation.

Table 7-1. Numeric Parameters for Test Facility (continued)

Parameter	Function
24	Display the instruction address, the compiler code at that address, and the data block for any instructions that perform data base and file operations. The data block includes the values and offsets of items in the list, data, key, argument, match, and update registers specifically used by the data base or file operation. This display is issued only when an accessed record meets the selection criteria in the match register. If there are no selection criteria for this operation or if the NOMATCH option is in effect, the display is issued for every record retrieved by the data base or file operation.
25	Display the instruction address, the compiler code at that address, and the data block for any instructions that perform data base and file operations. Display the values and offsets of items in the list, data, key, argument, match, and update registers for items specifically used by the data base or file operation. This display is issued for every record accessed by the data base or file operation.
34	Display the instruction address, the compiler code for that address, and the contents of the VPLUS buffer following an instruction generated by a statement that references a VPLUS form
42	Display instruction address and compiler code for that address only if the instruction is not listed in the compiler listing. Display contents of the list and data registers whenever the content of the list register (not the data register) changes.

Table 7-1. Numeric Parameters for Test Facility (continued)

Parameter	Function
43	Display the instruction address, the compiler code, and the contents of the list and data register for every instruction.
44	Display the instruction address, the compiler code, and the contents of the list, data, key, argument, match, and update registers for every instruction.
101	List the data and workspace recovery statistics for every command sequence.
102	List the data and workspace recovery statistics for the entire program.
121	Issue an overlay call trace whenever another program is called by a CALL statement.
122	Issue a trace whenever a file is locked or unlocked.
123	Issue a workspace recovery message whenever recovery is needed.

EXAMPLES

The following annotated examples illustrate various test modes. The compiler listing shown below is for the ADD PROGRAMMER command sequence used in the examples of test modes 1, 3, and 4:

```
             starting-location
            /
176.000   0077      $$ADD: <<begin the ADD commands>>
177.000   0077      $$A:
178.000   0077        $: <<help for the ADD command>>
179.000   0078
180.000   0078            display "The sub-commands for ADD are: ",line=2;
181.000   0080            set(command) command(ADD);
183.000   0082
184.000   0084        end; <<end of help for ADD>>
185.000   0085
186.000   0085
187.000   0085        $PROGRAMMER:
188.000   0086        $PR:
190.000   0086
191.000   0086            list PROGRAMMER:
192.000   0087                 PHONE;
193.000   0088            data LNAME:
194.000   0089                 FNAME:
195.000   0090                 PHONE;
196.000   0091            put PROGRAMMERS, list=(PROGRAMMER:PHONE);
197.000   0095
198.000   0095        end; <<end of ADD PROGRAMMER>>
            \
             ending-location
```

In these examples, the tests are requested by the TEST command just before executing the ADD PROGRAMMER command sequence:

Test Mode 1

This test mode displays the error message only when an error occurs; in this example, a duplicate key item error occurs.

```
> TEST 1,77,95     <---Execute instructions 77 thru 95 in test mode 1

> ADD PROGRAMMER

Enter programmer's last name: MARTIN      <---duplicate name

Enter programmer's first name: JOAN
Enter phone extension number: 3803

*ERROR: DUPLICATE KEY VALUE IN MASTER  (IMAGE 43,95,PROGRAMMERS)

+-D-A-T-A---F-I-L-E---D-U-M-P----------+<---data block for unsuccessful PUT
!
! PUT               COND: 43  STATUS: 43  RECNO: -1
! BASE: PROGB   SET: PROGRAMMERS
!
! POSN: LIST:                      DATA:
!  0     PROGRAMMER                 MARTIN          JOAN
!  30    PHONE                      3803
+--------------------------------------+

Enter programmer's last name: JONES     <---unique name; no test output

Enter programmer's first name: JAMES

Enter phone extension number: 3067

>
```

Test Mode 3

This test mode shows the same information as test mode 2 (the instruction address and the compiler code for every instruction), plus it shows the space used by the list and data registers, and the remaining processor work space.

```
> TEST 3,77,95                 # of entries in list register
                              /      # of words in data register
> ADD PROGRAMMER              /     /
        00000  000:000   LIST  DATA  CELL  WORK
        00087  032:007     1    15    64   256
        00088  032:005     2    17    64   256

Enter programmer's last name: FRANCIS
        00089  024:008     2    17    64   256

Enter programmer's first name: JAMES
        00090  024:009     2    17    64   256

Enter phone extension number: 4835
        00091  024:005     2    17    64   256
        00092  048:129     2    17    64   256
            /     \ /                 \     \
           /       compiler code       \     words left in work space
instruction location                    entries left in work space
```

Test Mode 4

In addition to the instruction location and compiler code issued by test mode 2, this mode displays instruction timings and the location of the stack pointers, Z, S, Q, and DL.

```
> TEST 4,77,95                                      stack pointers
                                                   /             \
> ADD PROGRAMMER                                  /               \
       00000  000:000  000000 000000 : Z      S      Q      D
       00087  032:007  000001 000001 : 07362 05612 05335 00092
       00088  032:005  000001 000002 : 07362 05612 05335 00092
                              \    /
                               \  /
                         instruction times
Enter programmer's last name: MAYOTTE
       00089  024:008  000016 000018 : 07362 05612 05335 00092

Enter programmer's first name: MARK
       00090  024:009  000015 000033 : 07362 05612 05335 00092

Enter phone extension number: 3303
       00091  024:005  000014 000047 : 07362 05612 05335 00092
       00092  048:129  000016 000063 : 09922 07735 05335 00092

> EXIT

END OF PROGRAM
```

Direct Test Output to File

The following example directs the test mode output to a file TEST. Test mode 1 is selected when the program is executed.

```
:BUILD TEST; REC=-80,,F,ASCII      <---build file for test output
:FILE TRANDUMP=TEST                <---equate TRANDUMP to that file
:RUN TRANSACT.PUB.SYS


TRANSACT/3000    HP32247A.00.01 - (C) Hewlett-Packard Co. 1982

SYSTEM NAME> MYPROG,,-1,77,95      <---send test mode 1 output to TRANDUMP

MYPROG  A00.00

 PASSWORD FOR PROGB>


*INFO: OPENED PROGB,3  (USER 23,-1)


> ADD PROGRAMMER                  <---command sequence at locations 77-95

Enter programmer's last name: MARTIN

Enter programmer's first name: JOAN

Enter phone extension number: 3803

*ERROR: DUPLICATE KEY VALUE IN MASTER  (IMAGE 43,95,PROGRAMMERS)

Enter programmer's last name: * CONTROL(Y) BREAK

> EXIT

END OF PROGRAM
```

To see the test mode output, run the EDITOR and display or list the contents of TEST.

Test Modes 22 through 25

Test modes 22 through 25 are very similar. For that reason, only test mode 25 is illustrated. The compiler code used for the example of test mode 25 is shown below. It uses three instructions that access a data base, a PUT, a REPLACE, and a DELETE. In the case of the REPLACE(CHAIN), two entries in the chain are replaced.

```
                 starting location
                /
453.000  0302        $PROGRAMMER:
454.000  0303        $PR:
455.000  0303            <<replace one programmer with another>>
456.000  0303
457.000  0303            <<set up and add entry for new name to PROGRAMMERS>>
458.000  0303
459.000  0303            list PROGRAMMER:
460.000  0304                 PHONE;
461.000  0305            data LNAME ("Enter new programmer's last name"):
462.000  0307                 FNAME ("Enter new programmer's first name"):
463.000  0309                 PHONE;
464.000  0310            put PROGRAMMERS, list=(PROGRAMMER:PHONE); <<add new>>
465.000  0314            set(update) list(PROGRAMMER);
466.000  0316            data LNAME ("Enter old programmer's last name"):
467.000  0318                 FNAME ("Enter old programmer's first name");
468.000  0320            set(key) list(PROGRAMMER);
469.000  0321            reset(stack) list; <<release space>>
470.000  0322
471.000  0322            <<update entries in PROG-AUTHOR>>
472.000  0322
473.000  0322            display "Updating entries in PROG-AUTHOR", line=2;
474.000  0324            list PROG-NAME: <<temp. storage for update>>
475.000  0325                 PROGRAMMER;
476.000  0326            replace(chain) PROG-AUTHOR,
477.000  0326                           list=(PROG-NAME:PROGRAMMER);
478.000  0330            reset(stack) list; <<release temp. storage>>
479.000  0331
480.000  0331            <<delete old entry in PROGRAMMERS>>
481.000  0331
482.000  0331            delete PROGRAMMERS, list=();
483.000  0334
484.000  0334        end; <<end of REPLACE PROGRAMMER>>
                \
                 ending location
```

Test Mode 25

This test mode, like test modes 22 through 24, displays the data block (DATA FILE DUMP) for instructions that access files or data sets. As part of the data block display, test mode 25 shows the contents of all the registers used by each data base or file operation. Note in the example below that the data block for REPLACE(CHAIN) is issued every time an entry is selected in the chain of the detail set PROG-AUTHOR.

```
> TEST 25,302,334

> REPLACE PROGRAMMER
        00303  032:007
        00304  032:005
        00305  040:008

Enter new programmer's last name: KING
        00307  040:009

Enter new programmer's first name: WENDY
        00309  024:005

Enter phone extension number: 3818
        00310  048:129
+-D-A-T-A---F-I-L-E---D-U-M-P----------+          <---data block for PUT
!
! PUT              COND: 0  STATUS: 0  RECNO: 21
! BASE: PROGB   SET: PROGRAMMERS
!                                                  <---contents of list &
! POSN: LIST:                        DATA:               data registers
!  0     PROGRAMMER                   KING            WENDY
!  30    PHONE                        3818
+--------------------------------------+
        00314  031:000
        00316  040:008

Enter old programmer's last name: CINTZ
        00318  040:009

Enter old programmer's first name: SIMON
        00320  198:007
        00321  208:254
        00322  081:028
        00323  080:000

Updating entries in PROG-AUTHOR
        00324  032:006
        00325  032:007
        00326  067:132
```

```
+-D-A-T-A---F-I-L-E---D-U-M-P----------+      <---data block for 1st REPLACE
!
!
! REPLACE(CHAIN)     COND: 0  STATUS: 0  RECNO: 1
! BASE: PROGB   SET: PROG-AUTHOR
!   KEY: PROGRAMMER   ARGUMENT: CINTZ           SIMON  <--key/argument regs
!
!        UPDATE:                     VALUE:
!          PROGRAMMER                   KING           WENDY <--update value
!
! POSN: LIST:                         DATA:             <--list/argument regs
!  0      PROG-NAME                     PROG1A
!  8      PROGRAMMER                    CINTZ          SIMON
+--------------------------------------+


+-D-A-T-A---F-I-L-E---D-U-M-P----------+      <---data block for 2nd REPLACE
!
! REPLACE(CHAIN)     COND: 0  STATUS: 0  RECNO: 2
! BASE: PROGB   SET: PROG-AUTHOR
!   KEY: PROGRAMMER   ARGUMENT: CINTZ           SIMON
!
!        UPDATE:                     VALUE:
!          PROGRAMMER                   KING           WENDY
!
! POSN: LIST:                         DATA:
!  0      PROG-NAME                     PROG2B
!  8      PROGRAMMER                    CINTZ          SIMON
+--------------------------------------+

2 RECORDS REPLACED
        00330  208:254
        00331  068:129
+-D-A-T-A---F-I-L-E---D-U-M-P----------+      <---data block for DELETE
!
! DELETE             COND: 0  STATUS: 0  RECNO: 14
! BASE: PROGB   SET: PROGRAMMERS
!   KEY: PROGRAMMER   ARGUMENT: CINTZ           SIMON
!
! POSN: LIST:                         DATA:
+--------------------------------------+
        00334  000:000


> EXIT
```

Note that this test mode only displays that part of the list and data registers included in a LIST= option of the data management statement.

Test Mode 34

This test mode is used to trace instructions that access VPLUS forms. The output from test mode 34 should always be sent to an alternate device from your terminal. Otherwise, the output interferes with the forms displayed on the screen.

The compiler code used for this example is shown below.

```
        starting location
       /
0035       ADD-CUSTOMER:
0035
0035           get(form) ADDFORM,
0035              init,
0035              list=(ACCOUNT:DATE),
0035              window=("Please enter a new customer"),
0035              f7=START-OF-PROGRAM,
0035              f8=END-OF-PROGRAM,
0035              autoread;
0051
0051       PUT-CUSTOMER:
0051
0051           set(key) list(ACCOUNT);       <<Set up key register>>
0052           find CUSTOMER, list=();   <<Check if customer exists>>
0055
0055           if STATUS <> 0 then       <<Customer already in base>>
0055 1           go to ADD-CUST-ERROR;
0058
0058           put CUSTOMER,
0058              list=(ACCOUNT:DATE),
0058              error=PUT-ERROR(*);      <<Process PUT verb error>>
0064
       \
        ending location
```

Before running the program VTEST in test mode -34, build a file, TEST, to receive the test data and equate TRANDUMP to that file:

```
   :BUILD TEST; REC=-80,,F,ASCII
   :FILE TRANDUMP=TEST
   :RUN TRANSACT.PUB.SYS

   SYSTEM NAME> VTEST,,-34,35,64      <--run VTEST in test mode 34 with
                                           test output sent to file TEST
```

The test output from file TEST looks like this:

```
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+ \
!                                          \
! PUT(FORM)       CODE: 0     FKEY: 0        \
! FORM: MENUNU                FILE: CUSTF  TF \ \
!                                               \
+--------------------------------------+       <--from previous form access
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+      /      statements
!                                           /
! UPDATE(FORM)    CODE: 0     FKEY: 1      /
! FORM:   MENU                FILE: CUSTF / /
!                                        /
+--------------------------------------+
       00035  160:131
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+       <--output from location 35
!
! GET(FORM)       CODE: 0     FKEY: 0          <--last key pressed is ENTER
! FORM:   ADDFORM             FILE:  CUSTFORM
!
! OFFSET: LIST:                   DATA:
!  0       ACCOUNT                 1113434343       \
!  10      FIRST-NAME              MARGARET          \
!  28      INITIAL                 S                  \
!  29      LAST-NAME               TRUEMAN             \
!  49      STREET-ADDR             524 East 79th Street  <--entered data
!  71      CITY                    New York            /
!  85      STATE                   NY                 /
!  87      ZIP                     10024             /
!  96      DATE                    07/21/82         /
!
+--------------------------------------+
       00051  198:000
       00052  065:137
       00055  011:001
       00058  048:137
       00064  004:000
       00035  160:131
```

```
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+        <--back to location 35
!
! GET(FORM)       CODE: 0     FKEY: 8           <--f8 pressed to exit
! FORM:    ADDFORM            FILE:    CUSTFORM
!
! OFFSET: LIST:                   DATA:
!  0         ACCOUNT               ...w...;b.                \
!  10        FIRST-NAME            .W...$...,...B,H..         \
!  28        INITIAL               S                           \
!  29        LAST-NAME             TRU..AN......   ...Z.        \
!  49        STREET-ADDR           Y..>..5"......,.X.?...        <--garbage
!  71        CITY                  .......3......               /
!  85        STATE                 >.                          /
!  87        ZIP                   .........                  /
!  96        DATE                  ..2....                   /
!
+--------------------------------------+
```

Test Mode 42

This test mode lists the contents of the list and data registers only when the list register is changed.

The compiler listing shown below is for two subcommands that are part of a LIST command sequence; this code is executed by entering LIST PROGRAMMER and LIST PROGRAM respectively.

```
                starting location
               /
   97.000  0019       $PROGRAMMER:
   98.000  0020       $PR:
   99.000  0020           <<list programmers>>
  100.000  0020
  101.000  0020           list PROGRAMMER:
  102.000  0021                PHONE;
  103.000  0022           output(serial) PROGRAMMERS,
  104.000  0022                          list=(PROGRAMMER:PHONE),
  105.000  0022                          sort=(PROGRAMMER),
  106.000  0022                          nocount;
  107.000  0028
  108.000  0028       end; <<end of LIST PROGRAMMER>>
  109.000  0029
  110.000  0029
  111.000  0029       $PROGRAM:
  112.000  0030       $P:
  113.000  0030           <<list programs>>
  114.000  0030
  115.000  0030           list PROG-NAME:
  116.000  0031                DESCRIPTION;
  117.000  0032           output(serial) PROGRAMS,
  118.000  0032                          list=(PROG-NAME:DESCRIPTION),
  119.000  0032                          sort=(PROG-NAME),
  120.000  0032                          nocount;
  121.000  0038
  122.000  0038       end; <<end of LIST PROGRAM>>
               \
                ending location
```

Note in the following test output that the current contents of the data register are never shown; only the previous contents. Thus the data register display in the test output from LIST PROGRAMMER contains garbage. Similarly, the data register display in the test output from LIST PROGRAM contains data from the previous command sequence.

```
> TEST 42,19,38
> LIST PROGRAMMER
        00020  032:007
+-L-I-S-T---D-U-M-P--------------------+    <--issued for LIST PROGRAMMER
! POSN: LIST:                        DATA:
!  0      PROGRAMMER                  .a.B.a..b.............B.H....
+--------------------------------------+
        00021  032:005
+-L-I-S-T---D-U-M-P--------------------+    <--issued for LIST PHONE
! POSN: LIST:                        DATA:
!  0      PROGRAMMER                  .a.B.a..b.............B.H....
!  30     PHONE                       ....
+--------------------------------------+
        00022  066:129

Programmer                      Phone Number
 CRESSMAN        PETE            3805
 ERCOLANI        JOE             4343
 KING            WENDY           3818
 LEDERMAN        ABE             3753
 VANN            KEITH           4046
        00028  000:000

> LIST PROGRAM
        00030  032:006
+-L-I-S-T---D-U-M-P--------------------+     <--issued for LIST PROG-NAME
! POSN: LIST:                        DATA:
!  0      PROG-NAME                   VANN
+--------------------------------------+
        00031  032:002
+-L-I-S-T---D-U-M-P--------------------+     <--issued for LIST DESCRIPTION
! POSN: LIST:                        DATA:
!  0      PROG-NAME                   VANN
!  8      DESCRIPTION                         KEITH        4046.........6.B..
!                                     ........"........X.?
+--------------------------------------+
        00032  066:130

Program Name Program Description
 CRUNCH        Compacts ASCII files.
 DISCOPY       Copies disc files.
 GTDATA        Generates random test data.
 PROJMAN       Project management using the critical path method.
 SGEN          Generates STREAM job files.
 TLIST         Lists the contents of a "STORE" tape.
 UNCRUNCH      Expands a file compacted by CRUNCH.
        00038  000:000


> EXIT
```

Test Modes 101 and 102

These test modes allow you to keep track of the list and data register size, and also whether recovery was needed. Test mode 101 displays test data at the end of every command sequence; test mode 102 only at the end of the program.

```
> TEST 101                          <---request test mode 101

> ADD PROGRAMMER                    <---start of command sequence

Enter programmer's last name: MARTIN

Enter programmer's first name: JOAN

Enter phone extension number: 3803

+-S-E-Q-U-E-N-C-E---D-U-M-P--+      <---current status of list/data regs
!                                         at end of this command sequence
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 17 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+

> ADD PROGRAM                       <---new command sequence

Enter program name: MYPROG

Program description: Test program for Manual

+-S-E-Q-U-E-N-C-E---D-U-M-P--+      <---status at end of second
!                                         command sequence
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 34 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+
> TEST 102                                  <---request test mode 102

> LIST PROGRAMMER

Programmer                    Phone Number
 CRESSMAN        PETE          3805
 ERCOLANI        JOE           4343
 KING            WENDY         3818
 LEDERMAN        ABE           3753
 MARTIN          JOAN          3803
 VANN            KEITH         4046

>LIST PROGRAM

Program Name Program Description
 CRUNCH       Compacts ASCII files.
 DISCOPY      Copies disc files.
 GTDATA       Generates random test data.
 MYPROG       Test program for Manual
 PROJMAN      Project management using the critical path method.
 SGEN         Generates STREAM job files.
 TLIST        Lists the contents of a "STORE" tape.
 UNCRUNCH     Expands a file compacted by CRUNCH.


> EXIT

+-R-U-N---D-U-M-P------------+          <---test output only issued
!                                             at end of program
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 34 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+

END OF PROGRAM
```

COMPILER ERROR MESSAGES

APPENDIX A

The Transact/3000 compiler generates two types of error messages. They appear on the compilation listing. The two types of error messages are the following:

(1) Errors that have resulted in the generation of no code or erroneous transaction code. Unless you have specified the XERR compiler control option, no code file is produced.

```
*** ERROR ***        ERROR-MESSAGE
```

(2) Conditions detected by the compiler that do not completely follow the TPL syntax rules. They are, however, correctable by the compiler in generating the transaction code file.

```
** WARNING **        ERROR-MESSAGE
```

ERROR-MESSAGE takes the following form:

```
^ (ERROR NUMBER) MESSAGE
```

where "^" is positioned under the location in the statement line where the compiler detected an error condition.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
-1	INVALID TERMINATOR	Compiler has detected an unexpected field termination character and expected one of the characters displayed between the square brackets . (A blank could be one of the expected characters.)
0	COMMAND LONGER THAN 16 CHARACTERS	Shorten the command label.
1	SUB-COMMAND LONGER THAN 16 CHARACTERS	Shorten the sub-command label.
4	INVALID VERB	Correct the verb name.
5	INVALID ITEM TYPE	The specified item type is not legal. Legal item types include X, U, 9, Z, P, I, J, K, R, E, or @.
6	MULTIPLE LABEL DEFINITION	Label identified has been previously defined. Change one label to a unique name.
7	INVALID MODIFIER	Correct or replace the modifier name.
8	EXPECTING A NUMERIC FIELD	Compiler expected a numeric field and has detected a null or non-numeric field.
9	INSTRUCTION BUFFER OVERFLOW	The compiler has generated more object code words than are permitted by that version of the compiler.
11	UNEXPECTED EOF IN TEXT FILE	Compiler has detected an EOF within a statement, probably caused by a missing statement semi-colon or a string terminating quotation mark.
12	NO MORE STACK SPACE FOR COMPILER TABLES	Segment the program or split it into two separate programs; compiler used all of the data stack.
13	FATAL ERROR: COMPILATION TERMINATED	Compiler has detected an unrecoverable error. See the previous error message on the compilation listing.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
14	INVALID OPTION	Correct or modify the option field.
15	EXPECTING ITEM NAME	Compiler expected the name of a data item and has encountered an invalid or null field.
16	ITEM NAME LONGER THAN 16 CHARACTERS	Shorten the data item name.
17	SET NAME LONGER THAN 16 CHARACTERS	Shorten the data set name.
18	INVALID SYSTEM NAME	Incorrect syntax in SYSTEM statement. Check syntax and correct it.
19	MULTIPLE SYSTEM DEFINITION	Previous system statement definition is still in effect. Multiple definition, or a missing END 'system-name' statement separating two programs in the same source text file can cause this error.
20	MULTIPLE BASE DEFINITION	Base name has been previously defined in SYSTEM statement; only the home base may be defined twice.
21	EXPECTING A COMMAND LABEL	Compiler expected a command label at this point in the program.
22	EXPECTING A SYSTEM DEFINITION	Compiler expects the system statement to be the first statement in the source text file. Only comment fields may precede the SYSTEM statement.
23	SYNTAX CHECKED BUT NOT COMPILED	Compiler has checked the identified syntax but has not generated the associated object code because of a previous error condition.
24	INVALID NUMBER	Identified numeric field contains an invalid number. Check the associated statement and option specifications.
25	MISSING TEXT	Compiler expected a field between the identified terminator and the previous one.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
29	UNEXPECTED TEXT BETWEEN DELIMITERS	Compiler expected two consecutive delimiters. The text between them is ignored.
30	SYNTAX ERROR	Compiler has detected a syntax error. Check the required syntax for the associated verb statement.
31	INVALID BASE NAME	Base name in the identified reference has not been declared in the system statement.
32	DECIMAL COUNT MUST BE LESS THAN TOTAL	Decimal place count is larger than the number of digits declared in the item definition.
33	EXPECTING A CHARACTER STRING	Compiler expected a character string within quotation marks at this point in the program.
34	LABEL LONGER THAN 32 CHARACTERS	Shorten the label.
36	INVALID PARAMETER FOR PROC CALL	An invalid proc parameter has been detected. Correct the parameter.
37	STORAGE BYTE COUNT TOO SMALL	The storage length specified in a DEFINE(ITEM) statement is less than the value calculated internally for that item. Storage length specified is ignored.
38	BASE DEFINITION MUST PRECEDE FILE DEFINITION	A file definition may not precede a base definition in the SYSTEM statement. Correct the statement.
39	DATA TYPE LENGTH NOT SUPPORTED	The storage length specified in a DEFINE(ITEM) statement is greater than the maximum size supported for that data type. Storage length specified is ignored.
40	MULTIPLE COMMAND DEFINITION	Command identified has been previously defined. Change one command label to a unique name.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
41	MULTIPLE SUB-COMMAND DEFINITION	The sub-command identified has been previously defined for the same command. Change one sub-command label to a unique name.
42	PASSWORD LONGER THAN 8 CHARACTERS	Shorten the password.
43	INVALID MODE	The mode must be one of the IMAGE DBOPEN modes (1 to 8).
45	EXPECTING A SET NAME	Compiler expected the name of a set and has encountered an invalid or null field.
46	MULTIPLE OPTION DEFINITION	The same option has been used more than once in the statement. This option is ignored.
47	MULTIPLE ITEM DEFINITION	The item identified has been previously defined. This definition is ignored.
48	MULTIPLE FILE DEFINITION	The file identified has been previously defined. This definition is ignored.
49	EXPECTING A FILE NAME	Compiler expected the name of a file and has encountered an invalid or null field.
50	INVALID FILE NAME	The file name identified is not a valid file name or back-reference. Correct the file name.
51	INPUT STRING LONGER THAN 80 CHARACTERS	The prompt-string specified exceeds the 80 character maximum for the input verb. Shorten the prompt-string.
52	EXPECTING A LABEL REFERENCE	Compiler expected a label name and has encountered an invalid or null field.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
53	TOO MANY COMPARE VALUES	The compiler has encountered a comparison with more than 100 values. Split the comparison into two or more IF statements.
54	MULTIPLE VALUES ONLY VALID FOR COMPARE EQUAL	Multiple compare values may only be specified if the test relationship is equality.
55	NON-PRINTING CHARACTER IN TEXT FILE IGNORED	The compiler has detected a non-printing character at the position indicated. The character has been ignored.
56	INVALID COMMAND LABEL	An invalid character has been detected in the first position of the command label. Correct the command label to begin with a character other than a "$" or "*".
57	FORMAT STATEMENT TOO LONG	The maximum format control block (500 words) has been exceeded. Reduce the number of display-fields and/or options in the FORMAT statement.
58	CONFLICTING OPTION IGNORED	This option cannot be specified, since a conflicting option has previously been specified. This option is ignored.
59	OBSOLETE SYNTAX	Compiler has detected obsolete, syntax which may be unsupported now. This refers to syntax that has subsequently been replaced by more complete features.
60	TOO MANY SORT ITEMS	More than 30 items have been specified for a FILE(SORT) operation. Reduce the number of SORT items.
61	LITERAL STRING TOO LONG	Character string exceeds 256 characters. Shorten the string or split it into several strings.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
62	ITEM REFERENCED TO ITSELF	A sub-item or data base synonym definition references itself. The definition is ignored.
63	FIELD ASSIGNMENT MUST BE 1 OR 2 CHARACTERS	Only "a" or "ab" is allowed for a field assignment, where "a" and "b" are any characters.
64	EXCEEDED 30 BLOCK LEVELS	Maximum nesting of IF/ELSE statements has been exceeded.
65	UNEXPECTED BLOCK TERMINATOR IGNORED	An unexpected DOEND statement has been encountered.
66	UNEXPECTED ELSE STATEMENT	The ELSE statement is invalid in this position. Correct the program.
67	SORT ERROR DURING LABEL CROSS-REFERENCE	When SORT was called to produce the cross-reference listing, an error occurred. The listing is produced unsorted.
68	SORT ERROR DURING DATA ITEM DEFINITIONS	When SORT was called to order the item definitions an error occurred. The listing is produced unsorted.
69	NO BASE(S) DECLARED	Correct the SYSTEM statement to include a data base declaration for the identified set name.
70	INVALID ELSE STATEMENT IGNORED	There was no previous IF statement without a terminating semi-colon.
71	EXPECTING AN ELSE CONDITIONED STATEMENT	The previous IF statement was not complete and an ELSE clause was expected.
72	INTERNAL LABEL - PLEASE REPORT THIS ERROR	Compiler has detected an error in its internal program control labels. The error has been caused by a system or compiler fault condition.
73	INCOMPLETE BLOCK STRUCTURE IN PRIOR SEQUENCE	In this command sequence, the compiler has had to force the correct termination of the block structure. Correct the program.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
74	VPLUS FORM NAME LONGER THAN 15 CHARACTERS	Shorten the form name.
75	INVALID VPLUS FORM NAME	Correct the format of the name to consist of only alphabetic and underscore characters.
76	TOO MANY ITEMS IN VPLUS FORM LIST	Shorten the VPLUS form to consist of fewer than 128 items.
77	EXCEEDED MAXIMUM NUMBER OF PROMPT STRINGS	More than 4096 prompt strings have been used in the program. Segment the program or reduce the number of prompt strings per segment.
78	INVALID COMMAND REFERENCE	Referenced command has not yet been defined.
79	SOURCE FILE READ ERROR	Examine the file display for a possible system problem.
81	EXPECTING AN OPTION	The syntax indicates that an option was expected; see verb description for more information.
91	PREVIOUS SEGMENT IS EMPTY-OPTION IGNORED	A !SEGMENT statement has been encountered and no code exists either since the last !SEGMENT statement or since the beginning of the program.
92	CODE FILE WRITE ERROR	Examine the file display; check any file equations for invalid access.
93	SEGMENT TABLE FULL	More than 63 !SEGMENT statements have been encountered.
94	DEFINITION CANNOT FOLLOW STATEMENT USAGE	A statement option usage that requires a prior definition has been detected.
95	INCORRECT NUMBER OF PARAMETERS FOR INTRINSIC	Check the intrinsic usage.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
96	INVALID INTRINSIC NAME	Either an invalid MPE intrinsic name or one that is not yet processable by the compiler has been detected. If the intrinsic is valid, remove the declaration and allow it to be resolved at run time.
97	EXPECTING AN UNTIL CLAUSE	A REPEAT statement requires an UNTIL clause.
98	'DOEND' FOR 'REPEAT DO' HAS ';' TERMINATOR	Remove the ';' following DOEND. The syntax is: REPEAT DO ... DOEND UNTIL ...;
100	DATA DICTIONARY REQUIRED BUT NOT AVAILABLE	Compiler requires information from the dictionary which it cannot open. Make the dictionary data base available or add the definitions to the SYSTEM statement.
102	TOO MANY ITEMS IN FILE LIST	More than 128 items have been used in a file list.
103	COMPUTATION LEVELS DO NOT MATCH	Parenthesis levels in a compound arithmetic expression do not match. Number of '[' do not match number of ']'.
111	DATA DICTIONARY: DATA BASE ERROR: ERROR MESSAGE	Dictionary data base error has occurred. See error message for appropriate action.
117	CANNOT OPEN INCLUDE FILE	Compiler cannot open the file named in an !INCLUDE control statement. Compilation continues using current source file.
118	TOO MANY INCLUDE FILES	More than five INCLUDE files have been nested (included from each in sequence).

NO.	MESSAGE	EXPLANATION AND/OR ACTION
119	INCOMPATIBLE INCLUDE FILE	Compiler has detected an incompatible INCLUDE file (not an ASCII file).
120	VPLUS FILE/FORM NOT FOUND IN DICTIONARY	The compiler has looked for information on a VPLUS forms file in the dictionary, but the definition does not exist. Add the definition to the SYSTEM statement.
121	ALIAS TABLE IS FULL	Too many item alias names and associated file names have been defined for the item. Reduce the number of alias names or associated file names.
122	MISSING "RECNO" OPTION	RECNO option is required for a verb using the DIRECT modifier.
123	LABEL LINKAGE FATAL ERROR	Compiler detected a bad linkage in the label resolution structure. Contact your local SE with a problem report.
124	DELIMITER STRING LONGER THAN 8 CHARACTERS	Too many delimiter characters are defined. Reduce the number to 8 or fewer.
125	UNEXPECTED UNTIL STATEMENT	The UNTIL statement is not valid in this position.

PROCESSOR ERROR MESSAGES

APPENDIX

B

The Transact/3000 Processor generates two types of error message: one type indicates actual errors, the other type provides information to the user, but does not indicate an actual error. Both types of message are explained in this appendix or in the appropriate reference manual for the indicated subsystem.

Error Messages

Error messages are displayed in the following format:

*ERROR: *error-message* (*error-info*)

Information Messages

Information messages have the same format as error messages, but are preceded by *INFO rather than *ERROR:

*INFO: *error-message* (*error-info*)

Information messages are conditions that the processor will tell the user about that are not errors. Also, messages that occur only in test modes are type INFO.

Error-Info

Whether appearing in an error or information message, *error-info* may contain up to five of the following fields:

(*type* *number* [,*code-location* [,PARM(*n*)] [,*file-name*]]) where:

type	is one the following:
USER	The error is caused by a user of a system and may usually be corrected by entering a different response.
PROG	The error is due to an error in the program and may usually be corrected by the programmer.
SYSTEM	The error is due to constraints of the system the program is running on and may be corrected by the operator.
TRAP	The error is due to an internal error in the processor and its occurrence should be called to the attention of the Systems Engineer.
	The following error types are derived from the indicated subsystem. The appropriate reference manual should be consulted for explanation of the error condition.
IMAGE	IMAGE data base error.
KSAM	KSAM utility error or file system error while operating on a KSAM file.
MPEF	MPE file system error.
VPLUS	VPLUS data entry utility error.
number	is the error number listed in this manual for type USER, PROG, SYSTEM or TRAP, or it is a number meaningful to the indicated subsystem.
code-location	is the internal location in the program at which the error occurred (reference the second column of numbers on the program compilation listing).
PARM(*n*)	*n* is the field number on multiple data entry fields at which the error was detected. All the following fields are ignored.
file-name	is the name of the data set or file that was involved in the error condition.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
1	ENTRY NOT NUMERIC	Data item type is integer, floating point, or numeric ASCII and a non-numeric character has been detected in the data entry field.
2	INPUT FIELD LONGER THAN *n*	Length of data entry exceeds the size *n* defined for the associated data item.
3	ORIGINAL RECORD HAS BEEN RESTORED	An error has occurred on updating an entry in a data set and the original entry has been restored. Probable cause for the error was a data entry value for which no associated master set entry exists.
4	NUMERIC INTEGER PART LONGER THAN *n*	Integer part of a decimal number exceeds the length *n* defined for the associated data item.
5	NUMERIC DECIMAL PART LONGER THAN *n*	Decimal part of a decimal number exceeds the length *n* defined for the associated data item.
6	MISSING COMMAND	A command option was entered without a command.
7	INVALID COMMAND/ OPTION: *command/option*	The command or command option entered is not valid.
8	INVALID/MISSING SUB-COMMAND: *subcommand*	The subcommand entered is not valid for the command specified, or no subcommand was entered for a command that requires one.
9	INVALID MODE	The mode used in opening the data base must be numeric.
10	INVALID SYSTEM NAME	System name must be one to six characters.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
11	NO SUCH SYSTEM	The processor cannot find or open code file named IPXXXXXX, where XXXXXX is the system name entered.
12	INVALID COMMAND PASSWORD	The command password to execute a command sequence is incorrect.
13	INVALID SUBCOMMAND PASSWORD	The sequence password to execute a subcommand sequence is incorrect.
14	INVALID TEST MODE PARAMETER	An invalid test mode or associated instruction range has been specified.
15	TEST MODE NOT AVAILABLE	The test modes have been disabled.
16	ATTEMPT TO ASSIGN NEGATIVE VALUE TO ITEM: *item-name*	The processor has detected an attempt to assign a negative value to a positive-type field.
17	INVALID ARITHMETIC FIELD FOR ITEM: *item-name*	The processor is attempting to execute an arithmetic operation using a data item defined as character, zoned or packed decimal (type=X,U,Z or P) and has detected an invalid data storage format for the item.
18	ENTRY CANNOT BE NEGATIVE	This item has been declared to always be positive.
19	INVALID LOGICAL CONNECTOR	Connector must be 'and, 'or' or 'to'.
20	INVALID PRECEDING RELATIONAL OPERATOR	Operator must be one of =, <>, <=, =>, <, >.
21	UNDELIMITED TEXT STRING	A string value must terminate with a quote.
22	INVALID PASSWORD FOR DATA BASE: *base-name*	User has entered an invalid password to the password prompt.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
23	OPENED *base-name*	Information message issued under test mode 1 when a data base is opened by Transact.
24	LOADED *procedure-name*	Information message issued under test mode 1 when a procedure is loaded by Transact.

PROGRAMMER ERRORS

NO.	MESSAGE	EXPLANATION AND/OR ACTION
1	ITEM NOT FOUND IN LIST REGISTER: *item-name*	Item name that is specified is not in the list register.
2	NO INPUT AVAILABLE	The processor has detected an attempt to use the input variable, but an INPUT statement has not been previously executed. Correct program.
3	UNDEFINED DATA ITEM: *item-name*	The processor is unable to resolve the data item's definition from the dictionary.
4	INVALID LIST START POSITION	Start list item-name does not occur before end list item-name in list register.
5	PARENT ITEM NOT FOUND IN LIST REGISTER FOR: *sub-item-name*	The current operation references a sub-item for which there is no associated parent item in the list register.
6	INVALID FILE FOR SORT OPERATION	A file used in a SORT operation must be defined with an access of SORT or R/W in the SYSTEM statement.
7	DATA BASE BUFFER NOT ON WORD BOUNDARY	The data buffer for a data base operation must start on a word boundary. If necessary, insert a one-character fill item before the first item of the data base list.
8	CANNOT FIND OR OPEN CODE FILE: *file-name*	Processor cannot find or open an 'IPXXXXXX' file that has been called programmatically rather than by the user.
9	SUB-ITEM NOT ALLOWED IN LIST REGISTER: *sub-item-name*	A sub-item may not be used in prompt or list statements.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
10	PROC DATA PARAMETER NOT ON WORD BOUNDARY	The data field for the item given in a data parameter for the PROC statement must begin on a word boundary. If necessary, insert a one byte fill item before the data item.
11	FILE BUFFER NOT ON WORD BOUNDARY	The data buffer for a file operation must begin on a word boundary. If necessary, insert a one character fill item before the first character of the file operation list.
12	ITEM NOT INITIALIZED: *item-name*	A reference has been made to an undefined data item.
13	ATTEMPT TO REFERENCE A MARKER ITEM: *item-name*	A marker item may be referenced only by a list pointer operation and list range options.
14	SOURCE CODE - INSTRUCTION IGNORED	The processor has detected an attempt to execute a statement marked as unexecutable by the compiler.
15	SORT KEY NOT WITHIN FILE ITEM LIST: *item-name*	An item given in the SORT item list of a file (SORT) operation must be contained in the item list of the last file operation, that is, the last FILE(OPEN) or FILE (WRITE) statement.
16	INVALID RETURN OPERATION	The processor is unable to execute the return, since there is no perform in effect.
17	EXCEEDED MAXIMUM PERFORMS	More than 10 nested PERFORM= blocks in a data base or file access statement; or more than 80 nested PERFORM statements have been defined in this program.
18	ARITHMETIC CONVERSION FOR ITEM: *item-name*	In an arithmetic operation, a conversion was attempted but it failed. Check data types in program; if necessary, contact your Systems Engineer.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
19	SORT BUFFER NOT ON WORD BOUNDARY	The buffer for a SORT operation does not begin on a word boundary. If necessary, insert a one-byte fill character before the SORT buffer.
20	INVALID/MISSING KSAM KEY	The key was either not defined or is not a valid key for this file.
21	LIST REGISTER IS EMPTY	An operation was attempted that required an item on the list register but the register was empty.
22	WORKSPACE REORGANIZED	This message informs the user that the processor had to attempt to find space by reorganizing the program workspace. This is reported under test mode 123 only.
23	ITEM NOT FOUND IN VPLUS FORM: *item-name*	An item within a list range on a VPLUS operation is not defined in the VPLUS buffer. Correct definition on the SYSTEM statement, or in the dictionary.
24	VPLUS BUFFER CONVERSION FOR ITEM: ITEM-NAME	An error occurred in translating ASCII from a VPLUS screen to an internal format. Check the definition of the item in program and screen. Run program in test mode 1 to get more information on the conversion error.
25	KEY REGISTER IS EMPTY	An operation was attempted that required a key item but none was defined.
26	CHECK FILE IS NOT A DATA SET	The file name for a check or checknot is not a data set.
27	SUB-ITEM FIELD EXTENDS BEYOND DATA REGISTER	An operation has been attempted that requires the data register to be extended. Use the data parameter of the SYSTEM statement to increase data register space.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
28	FILE BUFFER EXTENDS BEYOND DATA REGISTER	A file operation requires more data register space. Use the data parameter of the SYSTEM statement to increase data register space.
29	UNDEFINED BRANCH - SEE COMPILATION	An attempt has been made to use an undefined label.
30	DELETE NOT ALLOWED ON MPE FILE	A delete operation is not permitted on an MPE sequential file. File must be rewritten with unrequired records omitted.
31	ITEM STACK FULL	The number of data items has exceeded the default count or the number defined by *data-count* in the SYSTEM statement. Increase list register size with *data-count*.
32	LIST REGISTER FULL	IMAGE List register (2048 Bytes) -not Transact's list register - is full. Change program to: split the data base or split the display operations into two separate sequential steps, each of which retrieves some of the data items.
33	DATA REGISTER FULL	The data register size has exceeded the default size or the size defined by *data-length* in the SYSTEM statement. Increase data register size with *data-length*.
34	WORKSPACE FULL	Internal processor work space is full. Increase *work-length* in the SYSTEM statement.
35	WORKSPACE ENTRIES FULL	Internal processor work space is full. Increase *work-count* in the SYSTEM statement.
36	LEVEL STACK FULL	More than 10 processing levels have been activated in the program. Eliminate one or more levels.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
37	CALL TO UNLOADED PROCEDURE	The processor has detected an attempt to execute a PROC statement which references a procedure which the processor was unable to load.
38	LOCAL SEGMENT ITEM IN LIST: *item-name*	The processor has detected a local item in the list register during a transfer to a new segment. This error is not detected if the program was compiled with the OPTS option. Remove the item from the list register before the segment exit.
39	LOCAL SEGMENT ITEM IN MATCH REGISTER: *item-name*	The processor has detected a local item in the match register during a transfer to a new segment. This is not reported if the program was compiled with the OPTS option. Remove the local item from the match register before the segment exit.
40	LOCAL SEGMENT ITEM IN UPDATE REGISTER: *item-name*	The processor has detected a local item in the update register during a transfer to a new segment. This is not reported if the program was compiled with the OPTS option. Remove the local item from the update register before the segment exit.
41	INCOMPATIBLE CODE FILE	The code file is not compatible with the version of the processor currently executing. Recompile with correct version.
42	INSUFFICIENT STACK FOR WORK SPACE RELOAD	Processor needs more data stack for a work space reorganization. One solution is to segment the program. Transact uses the maximum stack space by default.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
43	INSUFFICIENT STACK FOR SYSTEM LOAD	The data stack required for the program is greater than the maximum specified for the processor. You must either segment the program or break it into smaller programs that are called with a CALL statement.
44	PRINT REGISTER LENGTH TOO LONG	The processor has detected an attempt to print a line of more than 512 characters.
45	INSUFFICIENT STACK FOR VPLUS BUFFER	Processor requires more data stack for a VPLUS buffer than is available. You should segment your program, or break it into smaller callable programs.
46	DECIMAL DIVIDE BY ZERO	Processor has detected a divide by zero in a packed decimal computation.
47	DECIMAL OVERFLOW	Processor has detected an overflow in a packed decimal computation.
48	EXTENDED PRECISION DIVIDE BY ZERO	Processor has detected a divide by zero in a double precision floating point computation.
49	EXTENDED PRECISION UNDERFLOW	Processor has detected an underflow in a double precision floating point computation.
50	EXTENDED PRECISION OVERFLOW	Processor has detected an overflow in a double precision floating point computation.
51	INTEGER OVERFLOW	Processor has detected an overflow in an integer arithmetic operation.
52	FLOATING POINT OVERFLOW	Processor has detected an overflow in a single precision floating point operation.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
53	FLOATING POINT UNDERFLOW	Processor has detected an underflow in a single precision floating point operation.
54	INTEGER DIVIDE BY ZERO	Processor has detected a divide by zero in an integer arithmetic computation.
55	FLOATING POINT DIVIDE	Processor has detected a divide by zero in a single precision floating point computation.
56	ARITHMETIC TRAP IN EXTERNAL PROCEDURE	Processor has detected an arithmetic trap in an external procedure called by a PROC statement.
57	OVERLAY CALLED FOR SEGMENT *n*	Informs the user that the processor has called on overlay for segment *n*. This is only reported in test mode 121.
58	NO VPLUS FORM AVAILABLE FOR UPDATE	UPDATE(FORM) was issued, but no form was displayed with a prior PUT(FORM) or GET(FORM).
59	ARGUMENT REGISTER OVERFLOW	Value has been entered into the argument register that exceeds 80 bytes in length.
60	INVALID OPERATION FOR FILE TYPE	Operation specified in program is invalid for the given file type.
61	INVALID/MISSING IMAGE KEY	The key item was either not set up or is not a valid key item for the given data set.
62	VPLUS FORM NOT FOUND	The VPLUS form designated as next form was not defined in the Transact program or the dictionary.
63	CALL FUNCTION HAS INVALID PASSWORD	The password specified in a CALL statement is invalid for any data bases to be opened in called program.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
64	INVALID PASSWORD FOR DATA BASE: *base-name*	The password specified for a data base in the SYSTEM statement is invalid.
66	ITEM NAME NOT DEFINED: *item-name*	Item whose name was set up in the data register for a VPLUS window option is not defined in the program.
67	VPLUS FORM IS NOT CURRENT: *form-name*	A VPLUS statement uses the CURRENT option when the named form is not the current form.
68	SORT KEY NOT IN SORT FILE: *item-name*	In a FIND or OUTPUT statement with a SORT= option, an item specified as a sort key is not in the sort file.
69	PARENT OF SORT KEY NOT IN SORT FILE FOR: *item-name*	In a FIND or OUTPUT statement with a SORT= option, the parent of a child item specified as a sort key is not in the sort file.
70	ATTEMPTED VPLUS OPERATION WHILE VPLS OPTION SET	A SET(OPTION) VPLS statement is in effect. You cannot execute a Transact statement that operates on VPLUS forms until you execute a RESET(OPTION) VPLS statement.
71	DATE/TIME EDIT MASK OVERFLOW	The application of an edit mask to a $TIME or $TODAY variable results in a string longer than 64 characters.
72	DATA REGISTER DOES NOT START ON WORD BOUNDARY	The data name specified in the DATA= option of the CALL statement does not start on a word boundary. Add filler byte to data register before adding this item.
73	UNABLE TO CLOSE VPLUS PRINTFILE	The CLOSE $FORMLIST statement failed because no forms file is open or because a SET(OPTION) VPLS was executed.

SYSTEM ERRORS

NO.	MESSAGE	EXPLANATION AND/OR ACTION
1	SORT INITIALIZATION	Error in call to System SORT Utility. The probable cause is insufficient disc space for SORT scratch file.
2	SORT FILE WRITE	Error on releasing record to the System SORT Utility. If cause is not apparent to the resident System Programmer then contact the Systems Engineer.
3	SORT OUTPUT	Error on requesting record from System SORT Utility. If cause is not apparent to the resident System Programmer then contact the Systems Engineer.
4	SORT END	Error in System SORT Utility during exit procedures. If cause is not apparent to the resident System Programmer then contact the Systems Engineer.
5	CANNOT OPEN PRINT FILE	The processor is unable to open the print file.
6	CANNOT ACCESS DATA DICTIONARY	The processor is unable to access the Data Dictionary.
7	CODE FILE READ	The processor detected a read error while reading the code file.
8	DISC SPACE NOT AVAILABLE FOR SORT FILE	Could not create scratch space for SORT operation.
9	PRINT FILE ACCESS	The print device is unavailable. Check FILE statement or call system operator.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
10	TRANOUT FILE ACCESS	A write operation to the TRANOUT file was unsuccessful. This should occur only if the TRANOUT file was equated to a system file or device.
11	CANNOT OPEN CALLED SYSTEM CODE FILE	Transact processor could not open the code file for a system name in a CALL statement.
12	CANNOT OPEN DATA BASE: *base-name*	Transact processor cannot open a data base because it does not exist or an incompatible open mode was specified.
13	CANNOT LOAD PROCEDURE: *procedure-name*	Transact processor cannot load the specified procedure. Usually occurs when a procedure cannot be found in a group, account, or system SL.

TRAPS

NO.	MESSAGE	EXPLANATION AND/OR ACTION
1	INTEGER STACK FULL	Internal stack is full. Please notify the Systems Engineer.
2	INTEGER STACK EMPTY	Internal stack is empty. Please notify the Systems Engineer.
3	WORKSPACE EMPTY	Internal processor work space is empty. Please notify Systems Engineer.
4	UNIMPLEMENTED CODE/ OPERAND ENCOUNTERED	Probable error in processor or compiler. Contact the Systems Engineer.
5	EMPTY CODE FILE	The processor detected an empty code file.
6	UNEXPECTED EOF IN CODE FILE	The processor encountered an unexpected end-of-file while reading the code file. Please notify the Systems Engineer.
7	ARITHMETIC CONVERSION FOR TABLE LITERAL	A program constant cannot be converted to the binary equivalent as the current statement requires. This usually means the code file is corrupted. If recompilation does not correct this, contact your Systems Engineer.
8	BROKEN WORK SPACE CHAIN	Processor has detected a break in a work space link list. Please report the error condition to the Systems Engineer.
9	ARITHMETIC TRAP	Processor has detected an arithmetic trap condition in its internal processing. Please report the error condition and the internal address (given as %*n*.%*nnnnn*) to the Systems Engineer.

NO.	MESSAGE	EXPLANATION AND/OR ACTION
10	OUT OF RANGE PCODE ADDRESS	Processor has detected a PCODE address out of the loaded range of transaction codes. Please report the error condition and the PCODE address (given as %*nnn*.%*nnnnn*) to the Systems Engineer.
11	DISPLAY FORMAT LEVEL OVERFLOW	Processor has detected an overflow in managing format levels. Please report the error condition to the Systems Engineer.
12	DISPLAY FORMAT LEVEL UNDERFLOW	Processor has detected an underflow in managing format levels. Please report the error condition to the Systems Engineer.

FLOW CHARTS OF FILE AND DATA BASE OPERATIONS

The flow charts shown in this appendix illustrate which file or data base procedures are called when a Transact verb performs a file or data base operation. Flow charts are given for the following verbs:

DELETE	- for an IMAGE data set, or a KSAM file operation
FIND	- for an IMAGE data set, or a KSAM or MPE file operation
GET	- for an IMAGE data set, or a KSAM, MPE, or VPLUS file operation
OUTPUT	- for an IMAGE data set, or a KSAM or MPE file operation
PATH	- for an IMAGE data set, or a KSAM file operation
PUT	- for an IMAGE data set, or a KSAM, MPE, or VPLUS file operation
REPLACE	- for an IMAGE data set, or a KSAM or MPE file operation
SET	- for a VPLUS file operation
UPDATE	- for an IMAGE data set, or a KSAM, MPE, or VPLUS file operation

DELETE Charts

Execution of a DELETE verb for an IMAGE data set access results in:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                            CURRENT           *
*    |                 |                               DIRECT            *
*  STATUS? -> yes <- STATUS?                           no modifier       *
*    no         |      no                              PRIMARY           *
*    |          |      |                                  |              *
*  DBFIND       |   DBCLOSE                               |              *
*    |          |      |                                  |              *
*     ------->  | <-----                                  |              *
*               |                                         |              *
*   -----> [DBLOCK]                                    [DBLOCK]          *
*  |            |                                         |              *
*  |          DBGET                                     DBGET            *
*  |            |                                         |              *
*  |  (mode=2 for SERIAL   )                    (mode=1 for CURRENT )    *
*  |  (     3     RSERIAL )                    (     4     DIRECT  )     *
*  |  (     5     CHAIN   )                    (     7     no mod. )     *
*  |  (     6     RCHAIN  )                    (     8     PRIMARY )     *
*  |            |                                         |              *
*  |            |                                         |              *
*  |          EOF/EOC? yes --------------->               |              *
*  |            |                         |               |              *
*  |            |                         |               |              *
*  |  <--no  Selected?(MATCH?)            |               |              *
*  |            yes                       |               |              *
*  |            |                         |               |              *
*  |         PERFORM? yes ----> label     |         PERFORM? yes-->label *
*  |             no               |       |               no        |    *
*  |            | <---------------        |               | <--------    *
*  |            |                         |               |              *
*  |         DBDELETE                     |            DBDELETE          *
*  |            |                         |               |              *
*  |        [DBUNLOCK]                    |           [DBUNLOCK]         *
*  |            |                         |               |              *
*  |         STATUS? yes ---------------->|               |              *
*  |            no                        |               |              *
*  |            |                         |               |              *
*   <----------                                                          *
*                                                                        *
*                                                                        *
 ************************************************************************
```

Execution of a DELETE verb for a KSAM file results in the following:

```
 *************************************************************************
*                                                                         *
*   CHAIN            SERIAL                                               *
*  RCHAIN           RSERIAL                  no modifier    CURRENT       *
*    |                 |                     PRIMARY        DIRECT        *
*    |                 |                        |             |           *
*  [FOPEN]          [FOPEN]                  [FOPEN]        [FOPEN]       *
*    |                 |                        |             |           *
*    |                 |                        |             |           *
*  STATUS? -> yes <- STATUS?                    |             |           *
*    no         |      no                       |             |           *
*    |          |      |                        |             |           *
*  FFINDBYKEY   |   FPOINT                      |             |           *
*    |          |      |                        |             |           *
*    |<----------------|<---- <------           |             |           *
*    |          |      |     |       |          |             |           *
*  [FLOCK]      |   [FLOCK]  |       |       [FLOCK]       [FLOCK]        *
*    |          |      |     |       |          |             |           *
*    |          |      |     |       |          |             |           *
*  FREAD        |   FREADC   |       |        FREAD         FREADDIR      *
*    |          |      |     |       |          |             |           *
*     -------->|<------      |       |           ----->  <-----           *
*              |             |       |                  |                 *
*   <----yes EOF/EOC?        |       |                  |                 *
*  |           no            |       |                  |                 *
*  |           |             |       |                  |                 *
*  |    Selected?(MATCH?) no->       |                  |                 *
*  |          yes                    |                  |                 *
*  |           |                     |                  |                 *
*  |        PERFORM? yes -> label    |             PERFORM? yes--->label  *
*  |           no             |      |                 no             |   *
*  |           | <------------       |                  | <-----------    *
*  |           |                     |                  |                 *
*  |        FREMOVE                  |               FREMOVE              *
*  |           |                     |                  |                 *
*  |       [FUNLOCK]                 |              [FUNLOCK]             *
*  |           |                     |                  |                 *
*  |        STATUS? no ------------->                   |                 *
*  |          yes                                                         *
*  |           |                                                          *
*  |           |                                                          *
*    --------->|                                                          *
*              |                                                          *
*                                                                         *
 *************************************************************************
```

FIND Charts

Execution of the FIND verb for an IMAGE data set access results in:

```
 ************************************************************************
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                       CURRENT                *
*    |                 |                          DIRECT                 *
*  STATUS? -> yes <- STATUS?                      no modifier            *
*   no          |      no                         PRIMARY                *
*    |          |       |                            |                   *
*  DBFIND       |    DBCLOSE                         |                   *
*    |          |       |                            |                   *
*     -------> | <-----                              |                   *
*   -------> DBGET                                 DBGET                 *
*  |           |                                     |                   *
*  |  (mode=2 for SERIAL  )                  (mode=1 for CURRENT )       *
*  |  (     3     RSERIAL )                  (     4     DIRECT  )       *
*  |  (     5     CHAIN   )                  (     7     no mod. )       *
*  |  (     6     RCHAIN  )                  (     8     PRIMARY )       *
*  |           |                                          |              *
*  |         EOF/EOC? yes ------------------>             |              *
*  |          no                             |            |              *
*  |           |                             |            |              *
*  |<---no  Selected?(MATCH?)                |            |              *
*  |          yes                            |            |              *
*  |           |                             |            |              *
*  |         SORT? no--->PERFORM? yes-->label|            |              *
*  |          yes           no           |   |                           *
*  |           |            |            |   |         PERFORM? yes -->  *
*  |        PERFORM? no --->|            |   |             no          | *
*  |           yes          |            |   |             |           | *
*  |           |            |            |   |             |      label *
*  |        Write to        |            |   |             | <-------    *
*  |        Sort File       |            |   |             |             *
*  |           | <---------- <-----------    |             |             *
*   <----no STATUS?                          |                           *
*             yes                            |                           *
*              |                             |                           *
*              | <---------------------------                            *
*            SORT? no----------------->                                  *
*              yes                     |                                 *
*              |                       |                                 *
*            PERFORM? no ------------->|                                 *
*              yes                     |                                 *
*              |                       |                                 *
*            Sort Sort File            |                                 *
*              |                       |                                 *
*        ----> |                       |                                 *
*       |   Read Sort Record           |                                 *
*       |      |                       |                                 *
*       |    EOF? yes ---------------->|                                 *
*       |     no                       |                                 *
*       |      |                       |                                 *
*      <-- perform routine          end                                  *
*                                                                        *
 ************************************************************************
```

Execution of a FIND verb for a KSAM file results in the following:

```
 ************************************************************************
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                                    CURRENT   *
*    |                 |                          no modifier  DIRECT    *
*  [FOPEN]          [FOPEN]                       PRIMARY        |       *
*    |                 |                             |           |       *
*  STATUS? -> yes <- STATUS?                         |           |       *
*    no         |      no                            |           |       *
*    |          |       |                            |           |       *
*  FFINDBYKEY   |   FPOINT                           |           |       *
*    |          |       |                            |           |       *
*  ->|---------|-----> |                             |           |       *
*  | |          |       |                            |           |       *
*  |FREAD ---->|<--- FREADC                        FREAD     FREADDIR    *
*  |           |                                     |           |       *
*  |         EOF/EOC? yes ------------------>        |           |       *
*  |           no                           |        |           |       *
*  |           |                            |        |           |       *
*  |<---no  Selected?(MATCH?)               |        ----> <----         *
*  |          yes                           |            |               *
*  |           |                            |            |               *
*  |         SORT? no--->PERFORM? yes-->label|            |               *
*  |          yes           no          |   |                            *
*  |           |            |           |   |         PERFORM? yes -->  *
*  |        PERFORM? no --->|           |   |             no          |  *
*  |           yes          |           |   |             |           |  *
*  |           |            |           |   |             |       label  *
*  |        Write to        |           |   |             | <-------     *
*  |        Sort File       |           |   |             |              *
*  |           |            |           |   |             |              *
*  |           | <---------- <-----------   |             |              *
*   <----no STATUS?                         |                            *
*             yes                           |                            *
*              |                            |                            *
*              | <---------------------------                            *
*              |                                                         *
*            SORT? no----------------->                                  *
*              yes                    |                                  *
*              |                      |                                  *
*            PERFORM? no ------------->|                                  *
*              yes                    |                                  *
*              |                      |                                  *
*          sort Sort File             |                                  *
*              |                      |                                  *
*        ----> |                      |                                  *
*       |    read Sort Record         |                                  *
*       |      |                      |                                  *
*       |     EOF? yes --------------->|                                  *
*       |      no                     |                                  *
*       |      |                      |                                  *
*       <-- perform routine          end                                 *
*                                                                        *
 ************************************************************************
```

FIND Charts

Execution of a FIND verb for an MPE file results in the following:

```
 ************************************************************************
*                                                                         *
*           CHAIN                                                         *
*           RCHAIN                                                        *
*           SERIAL                                DIRECT     no modifier  *
*           RSERIAL                               CURRENT    PRIMARY      *
*              |                                      |           |       *
*            [FOPEN]                              [FOPEN]    [FOPEN]      *
*              |                                      |           |       *
*           STATUS? yes-->                            |           |       *
*              no         |                           |           |       *
*              |          |                           |           |       *
*           FCONTROL      |                           |           |       *
*              | <--------                            |           |       *
*   -------> FREAD                               FREADDIR      FREAD      *
*  |           |                                      |           |       *
*  |           |                                      |           |       *
*  |        EOF/EOC? yes ------------------>          |           |       *
*  |           no                           |         |           |       *
*  |           |                            |         |           |       *
*  |<---no  Selected?(MATCH?)               |         ----> <----         *
*  |          yes                           |             |               *
*  |           |                            |             |               *
*  |        SORT? no--->PERFORM? yes-->label|             |               *
*  |          yes          no           |   |             |               *
*  |           |           |            |   |        PERFORM? yes -->     *
*  |        PERFORM? no --->|           |   |             no         |    *
*  |           yes          |           |   |             |          |    *
*  |           |            |           |   |             |      label    *
*  |        Write to        |           |   |             | <--------     *
*  |        Sort File       |           |   |             |               *
*  |           |            |           |   |             |               *
*  |           | <---------- <-----------   |             |               *
*   <----no STATUS?                         |                             *
*             yes                           |                             *
*              |                            |                             *
*              | <----------------------------                            *
*              |                                                          *
*            SORT? no----------------->                                   *
*             yes                      |                                  *
*              |                       |                                  *
*            PERFORM? no ------------->|                                  *
*             yes                      |                                  *
*              |                       |                                  *
*          sort Sort File              |                                  *
*              |                       |                                  *
*        ----> |                       |                                  *
*       |    read Sort Record          |                                  *
*       |      |                       |                                  *
*       |     EOF? yes --------------->|                                  *
*       |      no                                                         *
*       |      |                                                          *
*      <-- perform routine                                                *
*                                                                         *
 ************************************************************************
```

Execution of the GET verb for an IMAGE data set access results in:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                        CURRENT               *
*    |                 |                           DIRECT                *
*  STATUS? -> yes <- STATUS?                      no modifier            *
*    no         |      no                          PRIMARY               *
*    |          |      |                              |                  *
*  DBFIND       |    DBCLOSE                          |                  *
*    |          |      |                              |                  *
      -------> | <-----                               |                  *
*               |                                     |                  *
*   ----->    DBGET                                  DBGET               *
*  |            |                                     |                  *
*  |  (mode=2 for SERIAL   )                  (mode=1 for CURRENT )      *
*  |  (     3     RSERIAL  )                  (     4     DIRECT  )      *
*  |  (     5     CHAIN    )                  (     7     no mod. )      *
*  |  (     6     RCHAIN   )                  (     8     PRIMARY )      *
*  |            |                                     |                  *
*  |            |                                     |                  *
*  |        EOF/EOC? yes----->error                   |                  *
*  |            no                                    |                  *
*  |            |                                     |                  *
*  |            |                                     |                  *
*   <---no  Selected?(MATCH?)                         |                  *
*             yes                                     |                  *
*               |                                     |                  *
*               |                                     |                  *
*                                                                        *
*                                                                        *
 ************************************************************************
```

EOF = End of File
EOC = End of Chain

GET Charts

Execution of the GET verb for KSAM access results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*            CHAIN            SERIAL                                     *
*           RCHAIN           RSERIAL        No modifier     CURRENT      *
*             |                 |            PRIMARY        DIRECT       *
*           [FOPEN]          [FOPEN]            |              |         *
*             |                 |               |              |         *
*           STATUS? -> yes <- STATUS?        [FOPEN]        [FOPEN]      *
*             no        |       no              |              |         *
*             |         |       |               |              |         *
*           FFINDBYKEY  |     FPOINT            |              |         *
*             |         |       |               |              |         *
*             |         |       |               |              |         *
*    -----> FREAD <----- ---->FREADC <----      |              |         *
*   |         |                 |         |     |              |         *
*   |         |                 |         |     |              |         *
*   |      EOF/EOC?--> yes <--EOF/EOC?    |     |              |         *
*   |        no         |       no        |     |              |         *
*   |         |         |       |         |     |              |         *
*   |         |       error     |         |   FREAD        FREADDIR      *
*   |         |                 |         |     |              |         *
*   <---no-Selected?        Selected?no-->      |              |         *
*          (MATCH)           (MATCH)            |              |         *
*            yes               yes              |              |         *
*             |                 |               |              |         *
*             |                 |               |              |         *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
**************************************************************************
```

Execution of a GET verb for an MPE file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*           SERIAL                  no modifier     CURRENT              *
*           RSERIAL                 PRIMARY         DIRECT               *
*           CHAIN                      |              |                  *
*           RCHAIN                     |              |                  *
*              |                       |              |                  *
*           [FOPEN]                 [FOPEN]        [FOPEN]               *
*              |                       |              |                  *
*           STATUS? yes--->            |              |                  *
*              no         |            |              |                  *
*              |          |            |              |                  *
*           FCONTROL      |            |              |                  *
*              | <--------             |              |                  *
*   -------> FREAD                   FREAD        FREADDIR               *
*  |           |                       |              |                  *
*  |           |                       |              |                  *
*  |       EOF/EOC? yes----->error     |              |                  *
*  |           no                      |              |                  *
*  |           |                       |              |                  *
*  |           |                                                         *
*   <----no Selected?(MATCH?)                                            *
*             yes                                                        *
*              |                                                         *
*              |                                                         *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
 ************************************************************************
```

GET Charts

Execution of GET(FORM) for a VPLUS form results in the following:

```
 *************************************************************************
*  [VOPENTERM]<----[VOPENFORMF]<----[VCLOSEFORMF]                         *
*       |                                                                 *
* WINDOW option? yes-->item? yes-->                                       *
*       no              no        |         A                             *
*       |                |  VGETFIELDINFO  |                              *
*       |                |        |     CLEAR,                            *
*       | <------------------------     APPEND,                           *
*    CURRENT option? yes-->             FREEZE? yes ---> set FREEZAPP     *
*       no                |                no            to 0,1, or 2     *
*       |                 |                |                     |        *
* prior SET of form? yes->|                | <--------------------        *
*       no                |              FKEY=? yes ---> set item-name    *
*       |                 |                no                    |        *
*  VGETNEXTFORM           |                | <--------------------        *
*       | <---------------               Fn=label? yes --> if Fn pressed  *
*     INIT? yes---> VINITFORM              no              go to label    *
*       no              |                  |                     |        *
*       | <-------------                   | <--------------------        *
*   WINDOW= option? yes-->item? yes -->   exit                            *
*       no                 no        |                                    *
*       |                   |    VSETERROR                                *
*       | <------- VPUTWINDOW        |                                    *
*       | <----------------------------                                   *
*   FEDIT option? yes --->VFIELDEDITS                                     *
*       no                     |                                          *
*       | <--------------------                                           *
*   VSHOWFORM                                                             *
*   VREADFIELDS <-----------------------------------------                *
*       |                                                |                *
*   fkey pressed? yes-->AUTOREAD? yes-->set autoread     |                *
*       no                 no                |           |                *
*       |                   |           VREADFIELDS      |                *
*       |                   A                |           |                *
*       |<-----------------------------------            |                *
* <-yes STATUS?                                          |                *
* |     no                                               |                *
* | VFIELDEDITS <----------------------------            |                *
* |     |                                    |           |                *
* |  errors? yes ---------> VERRMSG          |           |                *
* |     no                     |             |           |                *
*  ---->|                   VPUTWINDOW       |           |                *
*   VPUTWINDOW                 |             |           |                *
*       |                   VSHOWFORM -------)----------->                *
*  VFINISHFORM                               |                            *
*       |                                    |                            *
*  VGETBUFFER & move to list                 |                            *
*       |                                    |                            *
*   conversion error? yes ---> VSETERROR     |                            *
*       no                     VREADFIELDS   |                            *
*       |                      VPUTWINDOW --->                            *
*    STATUS? yes -->VFIELDEDITS                                           *
*       no               |                                                *
*       | <--------------                                                 *
*       A                                                                 *
 *************************************************************************
```

Execution of the OUTPUT verb for an IMAGE data set access results in:

```
 *************************************************************************
*                                                                         *
*                                                                         *
*   CHAIN            SERIAL                                               *
*  RCHAIN           RSERIAL                            CURRENT            *
*    |                 |                               DIRECT             *
*  STATUS? -> yes <- STATUS?                           no modifier        *
*    no        |       no                              PRIMARY            *
*    |         |       |                                  |               *
*  DBFIND      |    DBCLOSE                               |               *
*    |         |       |                                  |               *
*     -------> | <-----                                   |               *
*              |                                          |               *
*   ----->   DBGET                                     DBGET              *
*  |           |                                          |               *
*  |  (mode=2 for SERIAL  )                     (mode=1 for CURRENT )     *
*  |  (     3     RSERIAL )                     (     4     DIRECT  )     *
*  |  (     5     CHAIN   )                     (     7     no mod. )     *
*  |  (     6     RCHAIN  )                     (     8     PRIMARY )     *
*  |           |                                          |               *
*  |           |                                          |               *
*  |        EOF/EOC yes ---------------->              PERFORM? yes-->    *
*  |           no                        |                no          |   *
*  |           |                         |                |         label *
*  |           |                         |                |           |   *
*  | <--no  Selected?(MATCH?)            |                | <---------    *
*  |          yes                        |                |               *
*  |           |                         |                |               *
*  |        PERFORM? yes ----> label     |              display           *
*  |           no                |       |                |               *
*  |           | <---------------        |                |               *
*  |         SORT? yes ----> write to    |                |               *
*  |           no            Sort File   |                |               *
*  |           |                 |       |                |               *
*  |        display              |       |                |               *
*  |           | <---------------        |                                *
*  <----- no STATUS                      |                                *
*              yes                       |                                *
*              | <-----------------------                                 *
*              |                                                          *
*             SORT? yes --> sort                                          *
*              no           Sort File                                     *
*              |              | <-----------------                        *
*              |            read                  |                       *
*              |            Sort File             |                       *
*              |              |                   |                       *
*              | <------ yes EOF? no--->display -->                       *
*              |                                                          *
*              |                                                          *
*                                                                         *
 *************************************************************************
```

EOF = End of File
EOC = End of Chain

OUTPUT Charts

Execution of the OUTPUT verb for a KSAM file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                 no modifier          DIRECT  *
*    |                 |                     PRIMARY            CURRENT  *
*  [FOPEN]          [FOPEN]                     |                  |     *
*    |                 |                     [FOPEN]            [FOPEN]  *
*  STATUS? -> yes <- STATUS?                    |                  |     *
*    no         |      no                       |                  |     *
*    |          |       |                       |                  |     *
*  FFINDBYKEY  |     FPOINT                     |                  |     *
*  ->|---------------->|
*  | |          |       |                       |                  |     *
*  |FREAD      |     FREADC                   FREAD            FREADDIR *
*  | |          |       |                       |                  |     *
*  |  -------> | <-----                          ---------> <-------     *
*  |           |                                          |              *
*  |        EOF/EOC yes --------------->               PERFORM? yes-->   *
*  |           no                       |                 no          |  *
*  |           |                        |                 |       label *
*  |           |                        |                 |           |  *
*  | <--no  Selected?(MATCH?)           |                 | <---------   *
*  |          yes                       |                 |              *
*  |           |                        |                 |              *
*  |        PERFORM? yes ----> label    |              display           *
*  |           no                |      |                 |              *
*  |           | <---------------       |                 |              *
*  |           |                        |                 |              *
*  |         SORT? yes ---> write to    |                 |              *
*  |           no         Sort File     |                 |              *
*  |           |                |       |                 |              *
*  |        display             |       |                 |              *
*  |           |                |       |                                *
*  |           | <--------------        |                                *
*  <----- no STATUS?                    |                                *
*              yes                      |                                *
*              | <-----------------------                                *
*              |                                                         *
*             SORT? yes --> sort                                         *
*              no         Sort File                                      *
*              |           | <-----------------                          *
*              |          read                 |                         *
*              |         Sort File             |                         *
*              |           |                   |                         *
*              | <----- yes EOF? no-->display -->                        *
*              |                                                         *
*              |                                                         *
*                                                                        *
*                                                                        *
*                                                                        *
 ************************************************************************
```

Execution of an OUTPUT verb for an MPE file results in the following:

```
 *************************************************************************
*                                                                        *
*           SERIAL                                                       *
*           RSERIAL                        no modifier      DIRECT       *
*           CHAIN                           PRIMARY         CURRENT      *
*           RCHAIN                             |              |          *
*              |                               |              |          *
*           [FOPEN]                         [FOPEN]         [FOPEN]      *
*              |                               |              |          *
*           STATUS? yes -->                    |              |          *
*             no           |                   |              |          *
*              |           |                   |              |          *
*           FCONTROL       |                   |              |          *
*              | <---------                    |              |          *
*   -------> FREAD                          FREAD         FREADDIR       *
*  |           |                               |              |          *
*  |           |                                -----> <------           *
*  |           |                                      |                  *
*  |        EOF/EOC yes --------------->           PERFORM? yes-->       *
*  |          no                        |             no          |      *
*  |           |                        |             |        label     *
*  |           |                        |             |           |      *
*  | <--no  Selected?(MATCH?)           |             | <---------       *
*  |         yes                        |             |                  *
*  |           |                        |             |                  *
*  |        PERFORM? yes ----> label    |           display              *
*  |          no                 |      |             |                  *
*  |           | <---------------       |             |                  *
*  |         SORT? yes ---> write to    |             |                  *
*  |          no           Sort File    |             |                  *
*  |           |                |       |             |                  *
*  |        display             |       |             |                  *
*  |           |                |       |                                *
*  |           | <--------------        |                                *
*  <----- no STATUS                     |                                *
*             yes                       |                                *
*              | <-----------------------                                *
*              |                                                         *
*             SORT? yes --> sort                                         *
*             no            Sort File                                    *
*              |              | <-----------------                       *
*              |            read                  |                      *
*              |            Sort File             |                      *
*              |              |                   |                      *
*              | <----- yes EOF? no--> display -->                       *
*              |                                                         *
*              |                                                         *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
 *************************************************************************
```

PATH Charts

Execution of a PATH verb for an IMAGE data set access results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                  |                                     *
*                                  |                                     *
*                               DBFIND                                   *
*                                  |                                     *
*                                  |                                     *
*                                                                        *
*                                                                        *
 ************************************************************************
```

Execution of a PATH verb for a KSAM file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                  |                                     *
*                                  |                                     *
*                              FFINDBYKEY                                *
*                                  |                                     *
*                                  |                                     *
*                                                                        *
*                                                                        *
 ************************************************************************
```

Execution of a PUT verb for an IMAGE data set results in the following:

```
 ***********************************************************************
*                                                                       *
*                                                                       *
*                                                                       *
*            [DBLOCK]                                                   *
*                |                                                      *
*              DBPUT   (mode=1)                                         *
*                |                                                      *
*           [DBUNLOCK]                                                  *
*                |                                                      *
*                |                                                      *
*                                                                       *
 ***********************************************************************
```

Execution of a PUT verb for a KSAM or MPE file results in the following:

```
 ***********************************************************************
*                                                                       *
*                 [FOPEN]                                               *
*                    |                                                  *
*                    |                                                  *
*                 [FLOCK]                                               *
*                    |                                                  *
*                  FWRITE                                               *
*                    |                                                  *
*                [FUNLOCK]                                              *
*                    |                                                  *
*                    |                                                  *
*                                                                       *
 ***********************************************************************
```

PUT Charts

Execution of a PUT(FORM) verb on a VPLUS form results in the following:

```
 ************************************************************************
*   [VOPENTERM]<-----[VOPENFORMF]<-----[VCLOSEFORMF]                     *
*       |                                                                *
*    WINDOW option? yes--> item?--> yes                                  *
*       no                  no       |                                   *
*       |                   |     VGETFIELDINFO                          *
*       |                   |        |                                   *
*       | <--------------------------     A                              *
*    CURRENT option? yes--                |                              *
*       no                |             CLEAR,                           *
*       |                 |             APPEND, yes--->set FREEZAPP      *
*    prior SET            |             FREEZE?         to 0,1,or 2      *
*    of form? yes ------->|                no                |           *
*       no                |                | <---------------            *
*       |                 |               FKEY=? yes--->set item-name    *
*  VGETNEXTFORM           |                no                |           *
*       | <---------------                 |<---------------             *
*       |                                 Fn=label? yes--->if Fn pressed *
*     INIT? no->items to transfer? yes     no              go to label   *
*      yes         no              |       |                  |          *
*       |          |          initialize   | <---------------            *
*       |       Current? no   buffer to   exit                           *
*    VINITFORM    yes     |     blanks                                   *
*       |          |      |        |                                     *
*    VGETBUFFER <--       |        |                                     *
*       | <-----------------------                                       *
*    LIST=() and not current? yes                                        *
*       no                     |                                         *
*       |                      |                                         *
*     Move LIST                |                                         *
*     to buffer                |                                         *
*       |                      |                                         *
*    VPUTBUFFER                |                                         *
*       | <--------------------                                          *
*     WINDOW option? yes--> item? yes--> VSETERROR                       *
*       no                  no               |                           *
*       |                   |                |                           *
*       | <------------- VPUTWINDOW          |                           *
*       | <--------------------------------------------------------      *
*    FEDIT option? yes ---> VFIELDEDITS                       |          *
*       no                      |                             |          *
*       | <--------------------                               |          *
*   VSHOWFORM                                                 |          *
*       |                                                     |          *
*   WAIT or Fn=                                               |          *
*   label option? yes --> STATUS option? no-----> VREADFIELDS |          *
*       no                      yes                   |       |          *
*       |                        |                WAIT=fn &   |          *
*       | <----------------------                 wrong key   |          *
*       | <-------------------------------- no<-- pressed?    |          *
*   VFINISHFORM                                      yes      |          *
*       |                                             |       |          *
*       A                                     VPUTWINDOW ---->           *
 ************************************************************************
```

Execution of the REPLACE verb for IMAGE data set access results in:

```
 *************************************************************************
*                                                                        *
*   CHAIN            SERIAL                                              *
*  RCHAIN           RSERIAL                           CURRENT            *
*    |                 |                              DIRECT             *
*  STATUS? -> yes <- STATUS?                         no modifier         *
*    no        |       no                             PRIMARY            *
*    |         |       |                                  |              *
*  DBFIND      |    DBCLOSE                               |              *
*    |         |       |                                  |              *
*     -------> | <-----                                   |              *
*              |                                          |              *
*   -----> [DBLOCK]                                  [DBLOCK]            *
*  |           |                                          |              *
*  |         DBGET                                     DBGET             *
*  |           |                                          |              *
*  |  (mode=2 for SERIAL  )                    (mode=1 for CURRENT )     *
*  |  (     3     RSERIAL )                    (     4     DIRECT  )     *
*  |  (     5     CHAIN   )                    (     7     no mod. )     *
*  |  (     6     RCHAIN  )                    (     8     PRIMARY )     *
*  |           |                                          |              *
*  |        EOF/EOC? yes -------------->                  |              *
*  |           no                       |                 |              *
*  |           |                        |                 |              *
*  |   <-no Selected?(MATCH?)           |                 |              *
*  |   |      yes                       |                 |              *
*  |   |       |                        |                 |              *
*  |   |    PERFORM? yes ---->          |            PERFORM? yes -->    *
*  |   |       no             |         |                no          |   *
*  |   |       | <--------- label       |                 | <------ label *
*  |   |       |                        |                 |              *
*  |   |  get update values             |          get update values     *
*  |   |       |                        |                 |              *
*  |   |    UPDATE? yes -->             |             UPDATE? yes ->     *
*  |   |      no           |            |               no          |    *
*  |   |       |           |            |                 |         |    *
*  |   |     DBPUT      DBUPDATE        |              DBPUT    DBUPDATE *
*  |   |       |           |            |                 |         |    *
*  |   | DBGET (mode=4)    |            |           DBGET(mode=4)   |    *
*  |   |       |           |            |                 |         |    *
*  |   |   DBDELETE        |            |            DBDELETE       |    *
*  |   |       |           |            |                 |         |    *
*  |    -----> | <---------             |                 | <--------    *
*  |      [DBUNLOCK]                    |           [DBUNLOCK]           *
*  |           |                        |                 |              *
*   <---no STATUS?                      |                 |              *
*             yes                       |                 |              *
*              |                        |                 |              *
*              | <----------------------                                 *
*              |                                                         *
 *************************************************************************
```

EOF = End of File
EOC = End of Chain

REPLACE Charts

Execution of a REPLACE verb for a KSAM file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*        SERIAL     CHAIN             no modifier        CURRENT         *
*        RSERIAL    RCHAIN              PRIMARY          DIRECT          *
*           |         |                      |              |            *
*        [FOPEN]   [FOPEN]                [FOPEN]         [FOPEN]        *
*           |         |                      |              |            *
*           |         |                      |              |            *
*   <-yesSTATUS?   STATUS?yes>               |              |            *
*  |        no        no      |              |              |            *
*  |        |         |       |              |              |            *
*  |     FPOINT  FFINDBYKEY   |              |              |            *
*  |        |        |        |              |              |            *
*   ------->|        |<-------               |              |            *
*           |        |                       |              |            *
*            ---><---                        |              |            *
*   -------->[FLOCK]                      [FLOCK]         [FLOCK]        *
*  |            |                            |              |            *
*  |            |                            |              |            *
*  |        FREAD (CHAIN/RCHAIN)           FREAD         FREADDIR        *
*  |           or                            |              |            *
*  |        FREADC (SERIAL/RSERIAL)          |              |            *
*  |            |                            |              |            *
*  |            |                             ------> <-----             *
*  |         EOF/EOC? yes ---------->                |                   *
*  |           no                       |            |                   *
*  |            |                       |            |                   *
*  |  <--no Selected?(MATCH?)           |            |                   *
*  | |         yes                      |            |                   *
*  | |          |                       |            |                   *
*  | |      PERFORM? yes ---->          |         PERFORM? yes ---->     *
*  | |          no            |         |            |             |     *
*  | |          |           label       |            |           label   *
*  | |          |             |         |            |             |     *
*  | |          | <-----------          |            | <-----------      *
*  | |    get update value              |        get update value        *
*  | |          |                       |            |                   *
*  | |      FUPDATE                     |         FUPDATE                *
*  | |          |                       |            |                   *
*  |  --------> |                       |            |                   *
*  |       [FUNLOCK]                    |       [FUNLOCK]                *
*  |            |                       |            |                   *
*  <------ no STATUS?                   |            |                   *
*              yes                      |            |                   *
*               |                       |                                *
*               | <------------------                                    *
*                                                                        *
*                                                                        *
 ************************************************************************
```

Execution of a REPLACE verb for an MPE file results in the following:

```
 ************************************************************************
*                                                                        *
*          SERIAL                                                        *
*          RSERIAL                                                       *
*          CHAIN                     no modifier          CURRENT        *
*          RCHAIN                      PRIMARY            DIRECT         *
*              |                           |                  |          *
*           [FOPEN]                     [FOPEN]            [FOPEN]       *
*              |                           |                  |          *
*          STATUS? no -->                  |                  |          *
*            yes         |                 |                  |          *
*              |         |                 |                  |          *
*          FCONTROL      |                 |                  |          *
*              | <--------                 |                  |          *
*   ------->[FLOCK]                     [FLOCK]            [FLOCK]       *
*  |           |                           |                  |          *
*  |           |                           |                  |          *
*  |            FREAD                    FREAD            FREADDIR       *
*  |           |                           |                  |          *
*  |           |                            --------> <-------           *
*  |           |                                     |                   *
*  |            EOF? yes --------------->            |                   *
*  |             no          |                       |                   *
*  |           |             |                       |                   *
*  |  <--no Selected?(MATCH?)                |       |                   *
*  | |          yes                      |           |                   *
*  | |         |                         |           |                   *
*  | |     PERFORM? yes ---->        |            PERFORM? yes ---->     *
*  | |         no           |        |           |              |        *
*  | |         |          label      |           |            label      *
*  | |         |            |        |           |              |        *
*  | |         | <-------------      |           | <-----------          *
*  | |    get update value           |      get update value             *
*  | |         |                     |           |                       *
*  | |      FUPDATE                  |        FUPDATE                    *
*  | |         |                     |           |                       *
*  |  ------->  |                     |           |                       *
*  |       [FUNLOCK]                 |       [FUNLOCK]                   *
*  |          |                      |           |                       *
*   <--no STATUS?                    |           |                       *
*            yes                     |           |                       *
*             |                      |                                   *
*             | <--------------------                                    *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
 ************************************************************************
```

SET Charts

Execution of a SET(FORM) verb for a VPLUS form results in the following:

```
 **********************************************************************
*         [VOPENFORMF]<-----[VCLOSEFORMF]                              *
*             |                                                        *
*        WINDOW option? yes --> item? yes --> VGETFIELDINFO            *
*            no                 no                 |                   *
*             |                  |                 |                   *
*             | <-----------------------------------                   *
*       form to SET same                                               *
*       as current form?  yes ------------------------->               *
*            no                                        |               *
*             |                                        |               *
*           CLEAR,                            Has current form been    *
*           APPEND,                           shown on screen? no      *
*           FREEZE? yes ------> set FREEZAPP           |        |      *
*            no                 to 0,1,or 2          yes        |      *
*             |                     |                  |        |      *
*             | <--------------------------------------         |      *
*          VGETNEXTFORM                                         |      *
*             | <------------------------------------------------      *
*             |                                                        *
*           INIT option? yes---> VINITFORM                             *
*            no                     |                                  *
*             |                     |                                  *
*             | <-------------------                                   *
*           WINDOW option? yes---> item? yes ----> VSETERROR           *
*            no                     no                |                *
*             |                      |                |                *
*             | <-------------- VPUTWINDOW            |                *
*             | <--------------------------------------                *
*      items to transfer?  yes --> VGETBUFFER                          *
*            no                        |                               *
*             |                 move Transact                          *
*             |                items to buffer                         *
*             |                        |                               *
*             |                    VPUTBUFFER                          *
*             |                        |                               *
*             | <-----------------------                               *
*          CLEAR,                                                      *
*          APPEND,                                                     *
*          FREEZE? yes -------> set FREEZAPP                           *
*            no                 to 0,1, or 2                           *
*             |                     |                                  *
*             | <--------------------                                  *
*          FEDIT option? yes ---> VFIELDEDITS                          *
*            no                     |                                  *
*             | <--------------------                                  *
*            exit                                                      *
 **********************************************************************
```

Execution of an UPDATE verb for an IMAGE data set access results in:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*           [DBLOCK]                                                     *
*              |                                                         *
*           DBUPDATE (mode=1)                                            *
*              |                                                         *
*           [DBUNLOCK]                                                   *
*              |                                                         *
*              |                                                         *
*                                                                        *
 ************************************************************************
```

Execution of an UPDATE verb for a KSAM file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*           [FLOCK]                                                      *
*              |                                                         *
*           FUPDATE                                                      *
*              |                                                         *
*           [FUNLOCK]                                                    *
*              |                                                         *
*              |                                                         *
*                                                                        *
 ************************************************************************
```

Execution of an UPDATE verb for an MPE file results in the following:

```
 ************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*           [FLOCK]                                                      *
*              |                                                         *
*           FUPDATE                                                      *
*              |                                                         *
*           [FUNLOCK]                                                    *
*              |                                                         *
*              |                                                         *
*                                                                        *
 ************************************************************************
```

UPDATE Charts

Execution of an UPDATE(FORM) verb for a VPLUS form results in:

```
 *************************************************************************
*       requested form same as current? no-------> error                  *
*           yes                                                           *
*            |                                                            *
*        [VOPENTERM]                                                      *
*            |                                                            *
*        WINDOW option? yes --> item? yes --> VGETFIELDINFO               *
*            no                 no                  |                     *
*            |                  |                   |                     *
*            | <-----------------------------------                       *
*          INIT option? yes ---->VINITFORM                                *
*            no                     |                                     *
*            | <---------------------                                     *
*        items to transfer?  yes --> VGETBUFFER --> move Transact         *
*            no                                     items to buffer       *
*            |                                            |               *
*            |                                       VPUTBUFFER           *
*            |                                            |               *
*            | <-----------------------------------------                 *
*         WINDOW option? yes ---> item? yes ---> VSETERROR                *
*           no                     no                  |                  *
*            |                     |                   |                  *
*            | <--------------- VPUTWINDOW <-----------                   *
*            | <---------------------------------------------------       *
*         FEDIT option?  yes --> VFIELDEDITS                       |      *
*            no                     |                              |      *
*            | <---------------------                              |      *
*         VSHOWFORM                                                |      *
*            |                                                     |      *
*       WAIT or FN=                                                |      *
*   label option?  yes-> STATUS? no ---> VREADFIELDS               |      *
*            no          yes                 |                     |      *
*            |            |               WAIT=fn                  |      *
*            | <-----------             & wrong key                |      *
*            |                           pressed? yes -->VPUTWINDOW ->|   *
*            |                               no                           *
*            |                               |                            *
*            | <--------VPUTWINDOW <----------                            *
*        VFINISHFORM                                                      *
*            |                                                            *
*         CLEAR,                                                          *
*        APPEND,                                                          *
*        FREEZE,? yes ----> set FREEZAPP to 0, 1, or 2                    *
*            no                       |                                   *
*            | <-------------------------                                 *
*          FKEY option? yes-------> set item-name                         *
*            no                        |                                  *
*            | <--------------------------                                *
*        Fn=label option? yes -----> if Fn pressed, go to label           *
*            no                               |                           *
*            | <----------------------------------                        *
*           exit                                                          *
 *************************************************************************
```

INTRINSICS ALLOWED IN DEFINE(INTRINSIC)

APPENDIX D

The intrinsics listed in this appendix may be specified in a DEFINE(INTRINSIC) statement.

ACTIVATE
ALTDSEG
ARITRAP
ASCII

BINARY

CALENDAR
CAUSEBREAK
CLOCK
CLOSELOG
COMMAND
CREATE
CREATEPROCESS
CTRANSLATE

DASCII
DATELINE
DBBEGIN
DBCLOSE
DBCONTROL
DBDELETE
DBEND
DBERROR
DBEXPLAIN
DBFIND
DBGET
DBINARY
DBINFO
DBLOCK
DBMEMO
DBOPEN
DBPUT
DBUNLOCK
DBUPDATE
DEBUG
DMOVIN
DMOVOUT

FATHER
FCHECK
FCLOSE
FCONTROL
FDELETE
FERRMSG
FFILEINFO
FFINDBYKEY
FFINDN
FGETINFO
FGETKEYINFO
FINDJCW
FLOCK
FMTCALENDAR
FMTCLOCK
FMTDATE
FOPEN
FPOINT
FREAD
FREADBACKWARD
FREADBYKEY
FREADC
FREADDIR
FREADLABEL
FREADSEEK
FREEDSEG
FREELOCRIN
FRELATE
FREMOVE
FRENAME
FSETMODE
FSPACE
FUNLOCK
FUPDATE
FWRITE
FWRITEDIR
FWRITELABEL

GENMESSAGE
GETDSEG
GETJCW
GETLOCRIN
GETORIGIN
GETPROCID
GETPROCINFO

KILL

LOCKGLORIN
LOCKLOCRIN
LOCRINOWNER

MAIL
MYCOMMAND

OPENLOG

PAUSE
PRINT
PRINTFILEINFO
PRINTOP
PRINTOPREPLY
PUTJCW

QUIT
QUITPROG

READ
READX
RECEIVEMAIL
RESETCONTROL
RESETDUMP

SEARCH
SENDMAIL

Intrinsics Allowed in DEFINE(INTRINSIC)

SETDUMP	VGETFIELD	VPUTLONG
SETJCW	VGETFIELDINFO	VPUTREAL
STACKDUMP	VGETFILEINFO	VPUTWINDOW
SUSPEND	VGETFORMINFO	VREADBATCH
	VGETINT	VREADFIELDS
TERMINATE	VGETKEYLABELS	VSETERROR
TIMER	VGETLONG	VSETKEYLABEL
	VGETNEXTFORM	VSETKEYLABELS
UNLOCKGLORIN	VGETREAL	VSHOWFORM
UNLOCKLOCRIN	VINITFORM	VUNLOADFORM
	VLOADFORMS	VWRITEBATCH
VCLOSEBATCH	VOPENBATCH	
VCLOSEFORMF	VOPENFORMF	WHO
VCLOSETERM	VOPENTERM	WRITELOG
VERRMSG	VPOSTBATCH	
VFIELDEDITS	VPRINTFORM	XARITRAP
VFINISHFORM	VPUTBUFFER	XCONTRAP
VGETBUFFER	VPUTDINT	XLIBTRAP
VGETDINT	VPUTFIELD	XSYSTRAP
	VPUTINT	

OPTIMIZING TRANSACT APPLICATIONS

APPENDIX E

This appendix provides suggestions for optimizing the run-time efficiency of Transact applications, a topic of interest to experienced Transact programmers responsible for large applications. The fine-tuning of *individual* programs is a very application-dependent problem, but the guidelines presented in this appendix should help you make some of the trade-offs. Material focuses on minimizing stack space and maximizing processing speed.

The first part of this appendix shows how to use Transact compiler statistics and test modes to assess your program's run-time use of the HP 3000 data stack. It also describes how certain program structure and language options affect the size of the data stack.

The last part of the appendix presents several program structure and coding suggestions for increasing Transact program run-time speed.

DATA STACK OPTIMIZATION

Data stacks for three different Transact program structures are discussed:

- Nonsegmented program
- Segmented program
- Main program with several CALLed subprograms
  - Implemented without the SWAP option
  - Implemented with the SWAP option

In each case, the entities found on the data stack at run time are identified and mapped to the compiler listing produced with the STATistics option. First, however, the general characteristics of the data stack and the compiler listing are defined.

The Run-Time Stack

The size and composition of the run-time stack vary with:

- VPLUS utilization
- the number and size of program segments
- how subprograms are designed and whether swapping is used during processing
- Transact processor register utilization

Figure E-1 provides a profile of the data stack, including a breakdown of the table register components. The stack profile is for a *nonsegmented program*, but it illustrates components that may occur on the data stack regardless of program structure.

Use test mode 4 to initially determine the stack requirements of your program or specific portions of it. Use the information provided below for individual components to selectively change their size, if desired.

The data stack components are defined as follows:

PCBX: Process Control Block Extension, a control area for MPE. The size of this area is operating system-dependent, but could be reduced slightly by running Transact with the NOCB option. This option should *only* be used to avoid stack overflow on a short-term basis. In the long run, applications should be structured and optimized so that use of the NOCB option is not necessary.

VPLUS INFO: an area that appears on the data stack if your Transact program uses VPLUS forms files. This area is used by the VPLUS subsystem. You can minimize the size of this area by using fast forms files.

TRANSACT OUTER BLOCK and TRANSACT PROCESSOR CONTROL BLOCK: areas containing data and pointers for Transact processor control. The size of these areas is version-dependent and installation-dependent.

DATA REGISTER: the Transact data register. (Section 4 explains how this register works.) The default size of this area is 1024 words. The *data-length* parameter of the DATA= option in the SYSTEM statement can be used to control the size of this area. Use test mode 3 or 102 to determine values to specify for the DATA= option. The data register for segmented programs or programs using CALLs must be large enough to accommodate all segments or subprograms. If one part of the application requires much more data register space than any other parts, invoke it using MPE's process handling feature.

```
    _____________________________
    |                           |
    |          PCBX             |
    |                           |
    |---------------------------|  <-- DL
    |                           |
    |       VPLUS INFO          |
    |                           |
    |---------------------------|  <-- DB    /------>  |--------------------------|
    |   TRANSACT OUTER BLOCK    |           /          |                          |
    |---------------------------|  <-- Q   /           |          BASES           |
    |   TRANSACT PROCESSOR      |         |            |--------------------------|
    |      CONTROL BLOCK        |         |            |   VPLUS COMAREA; VPLUS,  |
    |---------------------------|         |            |   KSAM, MPE, AND DATA    |
    |                           |         |            |     SET FILE INFO        |
    |      DATA REGISTER        |         |            |--------------------------|
    |---------------------------|         |            |       PROCEDURES         |
    |                           |         |            |--------------------------|
    |       TABLE REGISTER      |-------->|            |        COMMANDS          |
    |                           |         |            |--------------------------|
    |---------------------------|         |            |       SUBCOMMANDS        |
   *|       TABLE INDEX         |         |            |--------------------------|
    |---------------------------|         |            |                          |
   *|       TABLE LENGTH        |         |            |         ITEMS            |
    |---------------------------|         |            |                          |
    |                           |         |            |--------------------------|
    |       CODE REGISTER       |         |            |                          |
    |                           |         |            |      TEXT STRINGS        |
    |---------------------------|         |            |                          |
  **|      ITEM REGISTER        |         |            |--------------------------|
    |---------------------------|         |            |                          |
  **|      DATA INDEX           |         |            |     CONTROL STRINGS      |
    |---------------------------|         |            |                          |
  **|      DATA LENGTH          |         |            |--------------------------|
    |___________________________|         |            |                          |
    |                           |          \           |       WORK SPACE         |
    ~                           ~           \          |                          |
    |                           |  <-- S     \------>  |--------------------------|
    ~                           ~
    |___________________________|  <-- Z

 COMPONENTS IN THE DATA STACK              ENTITIES IN TABLE COMPONENTS

  *   Used to manage the TABLE REGISTER
 **   Used to manage the DATA REGISTER
```

Figure E-1. Data Stack Layout for Nonsegmented Transact Program

TABLE REGISTER: an area used to manage files, PROC calls, built-in and programmer-defined commands, sub-commands, data items and strings. The entities of this register are identified in the right-hand diagram of Figure E-1. Definitions of these entities and optimization suggestions are provided later in this section.

VPLUS forms files have a significant effect on the size of the TABLE REGISTER. If an application requires many VPLUS forms, you can conserve stack space by:

- using a CALL structure rather than a segmented program structure
- specifying only forms used by the main program and each subprogram in the SYSTEM statement of the main program and each subprogram. If only a forms file name is specified in a SYSTEM statement, Transact allocates TABLE REGISTER space for each form in the file and for all items associated with each form.

TABLE INDEX and TABLE LENGTH: areas used to manage the TABLE REGISTER. These areas consist of indexes and lengths, respectively, that correspond to entities of the TABLE REGISTER.

CODE REGISTER: an area containing instruction code data.

ITEM REGISTER, DATA INDEX, and DATA LENGTH: areas used to manage the DATA REGISTER component. The default size of each of these areas is 128 words. The DATA= option of the SYSTEM statement can be used to control the size of these areas. Use test mode 3 or 102 to determine a value to specify in the *data-count* parameter of the DATA= option.

DL, DB, Q, S, and Z: stack pointers. Transact requires 4K of the space between DATA LENGTH and S. SORT, laser printer, and other subsystems have stack requirements between S and Z. Use test mode 4 to locate stack pointers DL, Q, S, and Z for various portions of your program.

The TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components manage the following entities. In general, as the number of these entities used by your Transact program increases, so does the table register space required:

BASES: Image data bases.

VPLUS COMAREA; VPLUS, KSAM, MPE, AND DATA SET FILE INFO: forms files, forms, MPE and KSAM files, and data sets.

PROCEDURES: calls to user procedures or system intrinsics.

COMMANDS: built-in commands and command qualifiers. The 11 built-in commands (e.g., PRINT, SORT, REPEAT, EXIT) require 65 words of stack space. Programmer-defined commands increase these needs.

SUBCOMMANDS: programmer-defined subcommands.

ITEMS: data items defined with the DEFINE statement or defined in the Data Dictionary. This area can be optimized by using the DEFINE(ITEM) statement with the OPT option as well as compiling with the OPTI compiler option. Space is allocated for *all* data item textual names unless these options are invoked. Refer to syntax option 3 under DEFINE in section 6 and to OPTI in section 5.

TEXT STRINGS: literal ASCII strings. The stack requirements increase with the number of MOVE and DISPLAY statements with literals, WINDOW= options for VPLUS, etc. A way to optimize the TEXT STRINGS component is to keep all application messages in a message file instead of embedding them in the code. This practice both saves stack space and allows for easy message customization. Also consider keeping messages in forms files instead of using the WINDOW= option of the VPLUS verbs.

CONTROL STRINGS: internal representations of DISPLAY and FORMAT statements and complex arithmetic expressions.

WORK SPACE: work area used for sort items and match, update, input, key, and argument registers. By default, 256 words are allocated for the work space portion of the TABLE REGISTER and 64 words for the work space portions of the TABLE INDEX and TABLE LENGTH components.

The WORK= option of the SYSTEM statement can be used to control the size of the work space areas. Run test mode 3 or 102 to determine the requirements for your program. To override the defaults, specify a *work-length* value for the work space portion of the TABLE REGISTER and a *work-count* value for the work space portions of the INDEX and TABLE LENGTH registers.

Do not underestimate WORK SPACE requirements, because the recovery procedure invoked to re-use work spaces increases processing time. Maximize the usefulness of test mode 3 or 102 results by ensuring that all program options and branches are exercised several times.

Segmented programs and programs using the CALL statement have additional data stack components:

- Segmented programs use the data stack for keeping track of where the segments are located on disk and storing segment offsets. Code registers for a root segment and the current segment are also required.
- Programs containing CALLs without the SWAP option use the data stack to control both the main program and the current subprogram.
- Programs containing CALLs that use the SWAP option require data stack components very similar to those that do not use this option, but they do not all need to be present simultaneously on the stack.

Although these three structures require additional data stack components, they require less total stack space than a nonsegmented program if they contain more than two segments. The data stack requirements of each structure are described in detail later in this section.

Compiler Statistics

Figure E-2 illustrates the compiler listing produced when the STATistics option is in effect during compilation. The format shown is for *nonsegmented* programs, but is virtually the same for the other three structures being examined in this appendix. The fields are defined as follows:

COMPILE TIME STATISTICS

STACK=	x	The number of words the Transact compiler put on its data stack during compilation.
TABLE=	x	The portion of the data stack used for table space during compilation, in words.

RUN TIME STATISTICS

PCODE=	x	The number of words of instruction code data in the current segment, plus each segment compiled before it.
SCODE=	x	The number of words of instruction code for a particular segment, main program, or subprogram.

PARTIAL TABLE REGISTER

BASE=	x, y	The number of words that the TABLE REGISTER, the TABLE INDEX, and the TABLE LENGTH components require. Refer to Figure E-3 to map these compiler notations to the entities in these components. Note that the x values pertain to TABLE INDEX and TABLE LENGTH and that the y values pertain to TABLE REGISTER.
FILE=	x, y	
SET=	x, y	
PROC=	x, y	
$$CMD=	x, y	
$CMD=	x, y	
ITEM=	x, y	
STRNG=	x, y	
CNTRL=	x, y	
	------	
	x, y	The total number of words in the PARTIAL TABLE REG. SUMMARY.

FINAL TABLE REG. SUMMARY

WORK AREA=	x, y	The number of words in the WORK SPACE portions of the TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components. The x value reflects the work space in the TABLE INDEX and TABLE LENGTH components, and the y value reflects the work space in the TABLE REGISTER.
TABLE REG.=	y	The total number of words that the TABLE REGISTER occupies. This value is the sum of the y values in the PARTIAL TABLE REG. SUMMARY and the y value in WORK AREA.
TABLE INDX=	x	The total number of words that the TABLE INDEX requires This value is the sum of the x values in the PARTIAL TABLE REG. SUMMARY and the x value in WORK AREA.
TABLE LEN.=	x	The total number of words that the TABLE LENGTH needs. This value is the same as that for TABLE INDX=.

RUN TIME STACK SUMMARY

DATA REG.=	x	The number of words in the DATA REGISTER component.
TABLE REG.=	x	The number of words in the TABLE REGISTER component.
TABLE INDX=	x	The number of words in the TABLE INDEX component.
TABLE LEN.=	x	The number of words in the TABLE LENGTH component.
ROOT SEG.=	x	The number of words in the CODE REGISTER component.
ITEM REG.=	x	The number of words in the ITEM REGISTER component.
DATA INDEX=	x	The number of words in the DATA INDEX component.
DATA LEN.=	x	The number of words in the DATA LENGTH component.
	------	
	x	The total number of words in the RUN TIME STACK SUMMARY.

The following format differences occur when the program is segmented or uses CALLs:

- For segmented programs, the listing includes a PARTIAL TABLE REG. SUMMARY for each segment. The FINAL TABLE REG. SUMMARY and the RUN TIME STACK SUMMARY contain information that applies to the largest segment.

- For programs using the CALL statement with or without the SWAP option, the information shown in Figure E-2 is provided for the main program compilation and for each subprogram compilation.

```
*****COMPILE TIME STATISTICS****
     STACK=          x
     TABLE=          x

*******RUN TIME STATISTICS******
     PCODE=          x
     SCODE=          x

***PARTIAL TABLE REG. SUMMARY***
      BASE=    x,    y
      FILE=    x,    y
       SET=    x,    y
      PROC=    x,    y
     $$CMD=    x,    y
      $CMD=    x,    y
      ITEM=    x,    y
     STRNG=    x,    y
     CNTRL=    x,    y
          ------------
               x,    y

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    x,    y
          ------------
TABLE REG.=          y
TABLE INDX=          x
TABLE LEN.=          x

*****RUN TIME STACK SUMMARY*****
DATA  REG.=          x
TABLE REG.=          x
TABLE INDX=          x
TABLE LEN.=          x
ROOT  SEG.=          x
ITEM  REG.=          x
DATA INDEX=          x
DATA  LEN.=          x
          ------------
                     x

CODE FILE STATUS: REPLACED

0 COMPILATION ERRORS
PROCESSOR TIME=xx:xx:xx
ELAPSED TIME=xx:xx:xx
```

Figure E-2. Transact Compiler Statistics

```
                                               ____TABLE INDEX and TABLE
  ________________________                    |    LENGTH entities
  |         PCBX         |                    |
  |                      |                    |  __TABLE REGISTER entities
  |----------------------|                    | |
  |                      |                    | |
  |      VPLUS INFO      |                    v v
  |                      |
  |----------------------|                       ---------------------------
  | TRANSACT OUTER BLOCK |                       |                         |
  |----------------------|               BASE=x,y|         BASES           |
  | TRANSACT PROCESSOR   |                       |-------------------------|
  |    CONTROL BLOCK     |                       | VPLUS COMAREA; VPLUS,   |
  |----------------------|               FILE=x,y| KSAM, MPE, AND DATA     |
  |                      |                SET=x,y|    SET FILE INFO        |
 *|    DATA REGISTER     |DATA REG.=x            |-------------------------|
  |----------------------|               PROC=x,y|       PROCEDURES        |
  |                      |                       |-------------------------|
  |     TABLE REGISTER   |              $$CMD=x,y|        COMMANDS         |
  |                      |TABLE REG.=x           |-------------------------|
  |----------------------|               $CMD=x,y|       SUBCOMMANDS       |
**|     TABLE INDEX      |TABLE INDX=x           |-------------------------|
  |----------------------|                       |                         |
**|     TABLE LENGTH     |TABLE LEN.=x           |          ITEMS          |
  |----------------------|               ITEM=x,y|                         |
  |                      |                       |-------------------------|
  |     CODE REGISTER    |ROOT SEG.=x            |                         |
  |                      |              STRNG=x,y|       TEXT STRINGS      |
  |----------------------|                       |                         |
 *|    ITEM REGISTER     |ITEM REG.=x            |-------------------------|
  |----------------------|                       |                         |
 *|    DATA INDEX        |DATA INDEX=x           |     CONTROL STRINGS     |
  |----------------------|              CNTRL=x,y|                         |
 *|    DATA LENGTH       |DATA LEN.=x            |-------------------------|
  |______________________|                       |                         |
  |                      |          WORK AREA=x,y|       WORK SPACE        |**
  ~                      ~                       |                         |
  |                      |                       |-------------------------|
  ~                      ~
  |______________________|
```

COMPONENTS IN THE DATA STACK *ENTITIES IN TABLE COMPONENTS*

* Can be changed by using the DATA= option of the SYSTEM statement
** Can be changed by using the WORK= option of the SYSTEM statement

Figure E-3. Compiler Statistics Fields and Data Stack Components

Nonsegmented Programs

Nonsegmented programs generally execute faster than segmented programs, since the processor does not have to overlay information on the data stack when switching from segment to segment.

Figure E-4 illustrates the compiler listing produced when a nonsegmented program was compiled with the STAT option. Figures E-5 and E-6 map the compiler statistics to individual components and entities in the run-time data stack.

```
*****COMPILE TIME STATISTICS****
     STACK=      23368
     TABLE=      14482

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=       3765

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=   12,  176
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   82, 1047
     STRNG=  195, 2192
     CNTRL=  116,  916
          ------------
             455, 4950

****FINAL TABLE REG. SUMMARY****
WORK  AREA=   30,  100
          ------------
TABLE REG.=       5050
TABLE INDX=        485
TABLE LEN.=        485

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
TABLE REG.=       5050
TABLE INDX=        485
TABLE LEN.=        485
ROOT  SEG.=       3765
ITEM  REG.=         30
DATA INDEX=         30
DATA  LEN.=         30
          ------------
                 10075

CODE FILE STATUS: REPLACED

0 COMPILATION ERRORS
PROCESSOR TIME=00:01:43
ELAPSED TIME=00:02:15
```

Figure E-4. Compiler Statistics for Nonsegmented Program

```
 ___________________________
|                           |
|           PCBX            |
|                           |
|---------------------------|
|                           |
|        VPLUS INFO         |
|                           |
|---------------------------|
|   TRANSACT OUTER BLOCK    |            66 words
|---------------------------|
|   TRANSACT PROCESSOR      |
|      CONTROL BLOCK        |           816 words
|---------------------------|
|                           |
|      DATA REGISTER        |DATA  REG.=  200 words
|---------------------------|
|                           |
|       TABLE REGISTER      |TABLE REG.= 5050 words __
|                           |                          \
|---------------------------|                           \
|       TABLE INDEX         |TABLE INDX=  485 words ____\___see Figure E-6
|---------------------------|                           /
|       TABLE LENGTH        |TABLE LEN.=  485 words ___/
|---------------------------|
|                           |
|       CODE REGISTER       |ROOT  SEG.= 3765 words
|                           |
|---------------------------|
|      ITEM REGISTER        |ITEM  REG.=   30 words
|---------------------------|
|      DATA INDEX           |DATA INDEX=   30 words
|---------------------------|
|      DATA LENGTH          |DATA  LEN.=   30 words
|___________________________|
|                           |
~                           ~
|                           |
~                           ~
|___________________________|

                                     __________
        Approx. total data stack =   10957 words
```

Figure E-5. Data Stack of Nonsegmented Program

```
                  |--------------------------|
                  |                          |
                  |          BASES           |      BASE=   1,   10 words
                  |                          |
                  |--------------------------|
                  |  VPLUS COMAREA; VPLUS,   |      FILE=  38,  544 words
                  |   KSAM, MPE, AND DATA    |
                  |     SET FILE INFO        |       SET=  12,  176 words
                  |--------------------------|
                  |       PROCEDURES         |      PROC=   0,    0 words
                  |--------------------------|
                  |        COMMANDS          |     $$CMD=  11,   65 words
                  |--------------------------|
                  |       SUBCOMMANDS        |      $CMD=   0,    0 words
                  |--------------------------|
                  |                          |
                  |          ITEMS           |      ITEM=  82, 1047 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |       TEXT STRINGS       |     STRNG= 195, 2192 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |     CONTROL STRINGS      |     CNTRL= 116,  916 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |       WORK SPACE         | WORK AREA=  30,  100 words
                  |                          |             ^      ^
                  |                          |             |      |
                  |--------------------------|             |      |
                                                           |      |
    TABLE INDEX and TABLE LENGTH entities__________________|      |
                                                                  |
                  TABLE REGISTER entities_________________________|
```

Figure E-6. Table Register Entities of Nonsegmented Program

Segmented Programs

Data stack requirements can be optimized by segmenting your Transact program. The root segment and a current segment are always represented on the data stack. The savings in data stack space is approximately equal to the size of the segments not loaded. Although some processor time is required to overlay segments onto the data stack as they are required, the efficiency gained by decreasing the size of the data stack can be significant. Keeping applications functionally divided into segments minimizes segment switching.

The compiler listing for a segmented version of the nonsegmented program discussed earlier is shown in Figure E-7. The program consists of four segments, each of which has compiler statistics in the following categories:

```
COMPILE TIME STATISTICS
RUN TIME STATISTICS
PARTIAL TABLE REG. SUMMARY
```

The FINAL TABLE REG. SUMMARY and the RUN TIME STACK SUMMARY reflect information for the largest segment, in this case segment 4. The following fields in the RUN TIME STACK SUMMARY are of special note:

SEG. TABLE= XFER TABLE=	Areas of the data stack used to keep track of where segments are located. The number of words required for SEG. TABLE= is version-dependent. XFER TABLE= contains 2 words for each label defined with a DEFINE(ENTRY) statement.
ROOT SEG.=	The number of words that the code register requires for the root segment. Keep this segment as small as possible, since it is always memory-resident.
SCODE REG.=	The number of words that the code register requires for the largest segment.

Since the largest segment influences the number of words allocated for the data stack, try to make your segments as uniform in size as possible.

Figures E-8 and E-9 illustrate how the compiler statistics map to the run-time data stack.

```
SEGMENT 0 STATISTICS:
     STACK=      11210
     TABLE=       3138

*******RUN TIME STATISTICS******
     PCODE=         46
     SCODE=         46

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=    0,    0
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   54,  675
     STRNG=   28,  154
     CNTRL=   67,  303
           ------------
         188, 1686

COMPILED SEGMENT 0

SEGMENT 1 STATISTICS:
     STACK=      12683
     TABLE=       4624

*******RUN TIME STATISTICS******
     PCODE=        632
     SCODE=        586

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=    6,   87
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   54,  675
     STRNG=   42,  477
     CNTRL=   67,  303
           ------------
         208, 2096

COMPILED SEGMENT 1
```

Figure E-7. Compiler Statistics for Segmented Program (1 of 3)

```
SEGMENT 2 STATISTICS:
     STACK=      15623
     TABLE=       6419

*******RUN TIME STATISTICS******
     PCODE=       1676
     SCODE=       1044

***PARTIAL TABLE REG. SUMMARY***
      BASE=     1,   10
      FILE=    38,  544
       SET=     7,  103
      PROC=     0,    0
     $$CMD=     0,    0
      $CMD=     0,    0
      ITEM=    61,  769
     STRNG=    76,  612
     CNTRL=    67,  303
           ------------
              250, 2341

COMPILED SEGMENT 2

SEGMENT 3 STATISTICS:
     STACK=      15644
     TABLE=       7392

*******RUN TIME STATISTICS******
     PCODE=       3123
     SCODE=       1447

***PARTIAL TABLE REG. SUMMARY***
      BASE=     1,   10
      FILE=    38,  544
       SET=    12,  176
      PROC=     0,    0
     $$CMD=     0,    0
      $CMD=     0,    0
      ITEM=    55,  685
     STRNG=    77,  869
     CNTRL=    68,  308
           ------------
              251, 2592

COMPILED SEGMENT 3
```

Figure E-7. Compiler Statistics for Segmented Program (2 of 3)

```
SEGMENT 4 STATISTICS:
     STACK=      15738
     TABLE=       7134

*******RUN TIME STATISTICS******
     PCODE=       3773
     SCODE=        650

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=   12,  176
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   74,  943
     STRNG=  110,  865
     CNTRL=  115,  911
          ------------
             361, 3514
COMPILED SEGMENT 4

****FINAL TABLE REG. SUMMARY****
WORK  AREA=   30,  100
          ------------
TABLE REG.=       3614
TABLE INDX=        391
TABLE LEN.=        391

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
SEG. TABLE=        128
TABLE REG.=       3614
TABLE INDX=        391
TABLE LEN.=        391
ROOT  SEG.=         46
XFER TABLE=          8
SCODE REG.=       1447
ITEM  REG.=         30
DATA INDEX=         30
DATA  LEN.=         30
          ------------
                  6315
CODE FILE STATUS: REPLACED
0 COMPILATION ERRORS
PROCESSOR TIME=00:01:41
```

Figure E-7. Compiler Statistics for Segmented Program (3 of 3)

```
|---------------------------|
|           PCBX            |
|                           |
|---------------------------|
|                           |
|        VPLUS INFO         |
|---------------------------|
|   TRANSACT OUTER BLOCK    |                  66 words
|---------------------------|
|    TRANSACT PROCESSOR     |
|       CONTROL BLOCK       |                 816 words
|---------------------------|
|                           |
|       DATA REGISTER       |DATA  REG.=     200 words
|---------------------------|
|  DISC ADDRESS SEG. TABLE  |SEG. TABLE=     128 words
|---------------------------|
|                           |
|      TABLE REGISTER       |TABLE REG.=    3614 words ___
|                           |                              \
|---------------------------|                               \
|        TABLE INDEX        |TABLE INDX=     391 words _____\__see Figure E-9
|---------------------------|                               /
|       TABLE LENGTH        |TABLE LEN.=     391 words ____/
|---------------------------|
|       CODE REGISTER       |
|      (Root Segment)       |ROOT  SEG.=      46 words
|---------------------------|
|      TRANSFER TABLE       |XFER TABLE=       8 words
|---------------------------|
|       CODE REGISTER       |
|  (SCODE - Overlay Area)   |SCODE REG.=    1447 words
|---------------------------|
|       ITEM REGISTER       |ITEM  REG.=      30 words
|---------------------------|
|       DATA INDEX          |DATA INDEX=      30 words
|---------------------------|
|       DATA LENGTH         |DATA  LEN.=      30 words
|---------------------------|
~                           ~
|                           |
~                           ~
|___________________________|
                                          __________
         Approx. total data stack  =      7197 words
```

Figure E-8. Data Stack of Segmented Program

```
                  |-------------------------|
                  |                         |
                  |          BASES          |        BASE=   1,    10 words
                  |                         |
                  |-------------------------|
                  |  VPLUS COMAREA; VPLUS,  |        FILE=  38,   544 words
                  |   KSAM, MPE, AND DATA   |
                  |     SET FILE INFO       |         SET=  12,   176 words
                  |-------------------------|
                  |       PROCEDURES        |        PROC=   0,     0 words
                  |-------------------------|
                  |        COMMANDS         |       $$CMD=  11,    65 words
                  |-------------------------|
                  |       SUBCOMMANDS       |        $CMD=   0,     0 words
                  |-------------------------|
                  |                         |
                  |          ITEMS          |        ITEM=  74,   943 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |      TEXT STRINGS       |       STRNG= 110,   865 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |     CONTROL STRINGS     |       CNTRL= 115,   911 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |       WORK SPACE        |  WORK AREA=  30,   100 words
                  |                         |
                  |                         |               ^      ^
                  |-------------------------|               |      |
                                                            |      |
    TABLE INDEX and TABLE LENGTH entities___________________|      |
                                                                   |
                  TABLE REGISTER entities__________________________|
```

Figure E-9. Table Register Entities of Segmented Program

Programs Using CALLs Without SWAP Option

Splitting Transact programs into subprograms also decreases stack requirements.

Figure E-10 illustrates the compiler statistics for the program used for the earlier examples, restructured into a main program and 4 subprograms. The main program statistics appear on the first page, and statistics for the subprograms appear on the subsequent four pages of the listing.

Figure E-11 illustrates the layout of the run-time data stack. Note that the top half of the stack, used by the main program, has the same components as the nonsegmented program data stack. The PROCESSOR PROC. VAR. area holds processor variables for calling subprograms; the size of this area is version-dependent. The next area is a second TRANSACT PROCESSOR CONTROL BLOCK. The remaining areas are used by entities of the CALLed subprograms.

Figure E-12 portrays the entities in the TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components for the main program.

```
                main program
COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

*****COMPILE TIME STATISTICS****
     STACK=      11208
     TABLE=        962

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=         54

***PARTIAL TABLE REG. SUMMARY***
      BASE=     1,   10
      FILE=     2,   83
       SET=     0,    0
      PROC=     0,    0
    $$CMD=     11,   65
     $CMD=      0,    0
     ITEM=      1,    9
    STRNG=     10,   67
    CNTRL=      2,    5
          ------------
               27,  239

****FINAL TABLE REG. SUMMARY****
WORK  AREA=     5,   50
          ------------
TABLE REG.=         289
TABLE INDX=          32
TABLE LEN.=          32

*****RUN TIME STACK SUMMARY*****
DATA  REG.=         200
TABLE REG.=         289
TABLE INDX=          32
TABLE LEN.=          32
ROOT  SEG.=          54
ITEM  REG.=          25
DATA INDEX=          25
DATA  LEN.=          25
          ------------
                    682

CODE FILE STATUS: REPLACED
```

Figure E-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (1 of 5)

```
                subprogram 1
COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

*****COMPILE TIME STATISTICS****
     STACK=       11208
     TABLE=        3262

*******RUN TIME STATISTICS******
     PCODE=           0
     SCODE=         590

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=    8,  163
       SET=    6,   87
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   28,  348
     STRNG=   33,  413
     CNTRL=   25,  159
          ------------
             112, 1245

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    5,   50
          ------------
TABLE REG.=       1295
TABLE INDX=        117
TABLE LEN.=        117

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        100
TABLE REG.=       1295
TABLE INDX=        117
TABLE LEN.=        117
ROOT  SEG.=        590
ITEM  REG.=         20
DATA INDEX=         20
DATA  LEN.=         20
          ------------
                  2279

CODE FILE STATUS: REPLACED
```

Figure E-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (2 of 5)

```
                    subprogram 2
        COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

        *****COMPILE TIME STATISTICS****
             STACK=      11208
             TABLE=       4125

        *******RUN TIME STATISTICS******
             PCODE=          0
             SCODE=       1101

        ***PARTIAL TABLE REG. SUMMARY***
              BASE=    1,   10
              FILE=   11,  190
               SET=    6,   87
              PROC=    0,    0
             $$CMD=   11,   65
              $CMD=    0,    0
              ITEM=   19,  240
             STRNG=   54,  468
             CNTRL=   13,   45
                  ------------
                     115, 1105

        ****FINAL TABLE REG. SUMMARY****
        WORK  AREA=    8,   60
                  ------------
        TABLE REG.=       1165
        TABLE INDX=        123
        TABLE LEN.=        123

        *****RUN TIME STACK SUMMARY*****
        DATA  REG.=        100
        TABLE REG.=       1165
        TABLE INDX=        123
        TABLE LEN.=        123
        ROOT  SEG.=       1101
        ITEM  REG.=         20
        DATA INDEX=         20
        DATA  LEN.=         20
                  ------------
                          2672

        CODE FILE STATUS: REPLACED
```

Figure E-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (3 of 5)

```
                *subprogram 3*
COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

*****COMPILE TIME STATISTICS****
     STACK=      13640
     TABLE=       5622

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=       1456

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   19,  310
       SET=   10,  146
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   20,  249
     STRNG=   66,  793
     CNTRL=   32,  120
          ------------
             159, 1693

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    8,   60
          ------------
TABLE REG.=       1753
TABLE INDX=        167
TABLE LEN.=        167

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        100
TABLE REG.=       1753
TABLE INDX=        167
TABLE LEN.=        167
ROOT  SEG.=       1456
ITEM  REG.=         20
DATA INDEX=         20
DATA  LEN.=         20
          ------------
                  3703

CODE FILE STATUS: REPLACED
```

Figure E-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (4 of 5)

```
                  subprogram 4
         COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

         *****COMPILE TIME STATISTICS****
              STACK=       13640
              TABLE=        5178

         *******RUN TIME STATISTICS******
              PCODE=           0
              SCODE=         652

         ***PARTIAL TABLE REG. SUMMARY***
               BASE=     1,   10
               FILE=     2,   82
                SET=    10,  144
               PROC=     0,    0
              $$CMD=    11,   65
               $CMD=     0,    0
               ITEM=    57,  735
              STRNG=   103,  796
              CNTRL=    68,  703
                     ------------
                       252, 2535

         ****FINAL TABLE REG. SUMMARY****
         WORK  AREA=    40,  200
                     ------------
         TABLE REG.=        2735
         TABLE INDX=         292
         TABLE LEN.=         292

         *****RUN TIME STACK SUMMARY*****
         DATA   REG.=        200
         TABLE REG.=        2735
         TABLE INDX=         292
         TABLE LEN.=         292
         ROOT   SEG.=        652
         ITEM   REG.=         25
         DATA INDEX=          25
         DATA   LEN.=         25
                     ------------
                            4246

         CODE FILE STATUS: REPLACED
```

Figure E-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (5 of 5)

```
|--------------------------|
|          PCBX            |
|--------------------------|
|       VPLUS INFO         |
|--------------------------|
|  TRANSACT OUTER BLOCK    |               66 words
|--------------------------|
| TRANSACT PROC. CNTL. BLK.|              816 words
|--------------------------|
|      DATA REGISTER       |DATA   REG.=  200 words
|--------------------------|
|     TABLE REGISTER       |TABLE REG.=   289 words __
|--------------------------|                          \
|       TABLE INDEX        |TABLE INDX=    32 words ___\___see Figure E-12
|--------------------------|                          /
|       TABLE LENGTH       |TABLE LEN.=    32 words __/
|--------------------------|
|      CODE REGISTER       |ROOT   SEG.=   54 words
|--------------------------|
|      ITEM REGISTER       |ITEM   REG.=   25 words
|--------------------------|
|       DATA INDEX         |DATA INDEX=    25 words
|--------------------------|
|       DATA LENGTH        |DATA   LEN.=   25 words
|--------------------------|
|  PROCESSOR PROC. VAR.    |              194 words
|--------------------------|
| TRANSACT PROC. CNTL. BLK.|              816 words
|--------------------------|
|      TABLE REGISTER      |TABLE REG.=  2735 words
|--------------------------|
|      TABLE INDEX         |TABLE INDX=   292 words
|--------------------------|
|      TABLE LENGTH        |TABLE LEN.=   292 words
|--------------------------|
|      CODE REGISTER       |ROOT   SEG.=  652 words
|--------------------------|
|      ITEM REGISTER       |ITEM   REG.=   25 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=    25 words
|--------------------------|
|      DATA LENGTH         |DATA   LEN.=   25 words
|--------------------------|
|                          |              __________
      Approx. Total Data Stack   =      6620 words
```

Figure E-11. Data Stack of Program Using CALLs Without the SWAP Option

```
                 |--------------------------|
                 |                          |
                 |          BASES           |       BASE=   1,    10 words
                 |                          |
                 |--------------------------|
                 |  VPLUS COMAREA; VPLUS,   |       FILE=   2     83 words
                 |   KSAM, MPE, AND DATA    |
                 |     SET FILE INFO        |        SET=   0,     0 words
                 |--------------------------|
                 |        PROCEDURES        |       PROC=   0,     0 words
                 |--------------------------|
                 |         COMMANDS         |     $$CMD=  11,    65 words
                 |--------------------------|
                 |        SUBCOMMANDS       |      $CMD=   0,     0 words
                 |--------------------------|
                 |                          |
                 |          ITEMS           |      ITEM=   1,     9 words
                 |                          |
                 |--------------------------|
                 |                          |
                 |       TEXT STRINGS       |     STRNG=  10,    67 words
                 |                          |
                 |--------------------------|
                 |                          |
                 |     CONTROL STRINGS      |     CNTRL=   2,     5 words
                 |                          |
                 |--------------------------|
                 |                          |
                 |        WORK SPACE        | WORK AREA=   5,    50 words
                 |                          |              ^      ^
                 |                          |              |      |
                 |--------------------------|              |      |
                                                             |      |
     TABLE INDEX and TABLE LENGTH entities___________________|      |
                                                                    |
                    TABLE REGISTER entities_________________________|
```

Figure E-12. Table Register Entities of Main Program Using CALLs Without the SWAP Option

Programs Using CALLs with SWAP Option

If your main program is large, the SWAP option can reduce the amount of data stack space required. This option causes some of the main program's stack entities to be written out to a temporary file when a subprogram is CALLed. The trade-off in this instance is the overhead required to create this file and restore its contents when control returns to the main program.

The compiler statistics provided for this program structure are the same as those provided when a program uses CALLs without the SWAP option. Refer back to Figure E-10 for compiler statistics produced when the earlier example was recoded to use the SWAP option with its CALLs.

When the main program is in control, the data stack looks like the top portion of the layout illustrated in Figure E-11. Components PCBX through PROCESSOR PROC. VAR. are present.

Figure E-13 illustrates how the data stack looks after subprogram 4 is CALLed:

- Only a subset of the main program's TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components are on the stack. The remainder of the entities have been placed in a temporary MPE file.
- The following components of the main program have also been placed in the temporary file: CODE REGISTER, ITEM INDEX, and DATA LENGTH.
- Two areas of the data stack are used for processor variables: PROCESSOR PROC. VAR. and SWAP PROC. VARIABLES. As in the case of CALLs without the SWAP option, the number of words in these areas is version dependent.

The entities in the main program's table register subsets are identified in Figure E-14. Note that the values for BASE=, FILE=, and SET= entities are represented in the compiler statistics for the main program in the PARTIAL TABLE REG. SUMMARY (refer to the first page of Figure E-10).

Figure E-15 illustrates the table components for the largest of the subprograms--subprogram 4.

```
|--------------------------|
|           PCBX           |
|--------------------------|
|        VPLUS INFO        |
|--------------------------|
|   TRANSACT OUTER BLOCK   |               66 words
|--------------------------|
|    TRANSACT PROCESSOR    |
|       CONTROL BLOCK      |              816 words
|--------------------------|
|      DATA REGISTER       |DATA   REG.=  200 words
|--------------------------|
| SUBSET OF TABLE REGISTER |               93 words __
|--------------------------|                          \
| SUBSET OF TABLE INDEX    |                3 words ___\___see Figure E-14
|--------------------------|                          /
| SUBSET OF TABLE LENGTH   |                3 words __/
|--------------------------|
|   PROCESSOR PROC. VAR.   |              194 words
|--------------------------|
|   SWAP PROC. VARIABLES   |               67 words
|--------------------------|
|    TRANSACT PROCESSOR    |              816 words
|       CONTROL BLOCK      |
|--------------------------|
|      TABLE REGISTER      |TABLE REG.= 2735 words __
|--------------------------|                          \
|      TABLE INDEX         |TABLE INDX=  292 words ___\___see Figure E-15
|--------------------------|                          /
|      TABLE LENGTH        |TABLE LEN.=  292 words __/
|--------------------------|
|      CODE REGISTER       |ROOT   SEG.= 652 words
|--------------------------|
|      ITEM REGISTER       |ITEM   REG.=  25 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=   25 words
|--------------------------|
|      DATA LENGTH         |DATA   LEN.=  25 words
|--------------------------|
|                          |           ___________
         Approx. Total Data Stack  =    6304 words
```

Figure E-13. Data Stack of Program Using CALLs
With the SWAP Option (CALLed Program is on the Stack)

```
                  |-------------------------|
                  |                         |
                  |          BASES          |BASE= 1, 10 words
                  |                         |
                  |-------------------------|
                  |  VPLUS COMAREA; VPLUS,  |FILE= 2, 83 words
                  |   KSAM, MPE, & DATA     |
                  |   SET FILE INFO         |SET = 0,  0 words
                  |-------------------------|
                                                 ^    ^
                                                 |    |
                                                 |    |
 TABLE INDEX and TABLE LENGTH entities___________|    |
                                                      |
                                                      |
               TABLE REGISTER entities________________|
```

Figure E-14. Table Register Subsets for Main Program After CALLing Subprogram

```
                |--------------------------|
                |                          |
                |          BASES           |      BASE=   1,    10 words
                |                          |
                |--------------------------|
                |  VPLUS COMAREA; VPLUS,   |      FILE=   2     82 words
                |   KSAM, MPE, AND DATA    |
                |     SET FILE INFO        |       SET=  10,   144 words
                |--------------------------|
                |       PROCEDURES         |      PROC=   0,     0 words
                |--------------------------|
                |        COMMANDS          |    $$CMD=   11,    65 words
                |--------------------------|
                |       SUBCOMMANDS        |     $CMD=    0,     0 words
                |--------------------------|
                |                          |
                |          ITEMS           |     ITEM=   57,   735 words
                |                          |
                |--------------------------|
                |                          |
                |       TEXT STRINGS       |    STRNG=  103,   796 words
                |                          |
                |--------------------------|
                |                          |
                |     CONTROL STRINGS      |    CNTRL=   68,   703 words
                |                          |
                |--------------------------|
                |                          |
                |        WORK SPACE        | WORK AREA=  40,   200 words
                |                          |             ^       ^
                |                          |             |       |
                |--------------------------|             |       |
                                                         |       |
  TABLE INDEX and TABLE LENGTH entities__________________|       |
                                                                 |
                  TABLE REGISTER entities________________________|
```

Figure E-15. Table Register Entities of Subprogram 4

Program Structure Comparison

The following chart summarizes the data stack requirements of the four program examples just examined. The values shown do not include the stack space required for the following components: PCBX, VPLUS, and subsystems such as SORT.

Application Structure	Approx. Data Stack
Nonsegmented Program	10957 words
Segmented Program	7197 words
Main Program CALLing Sub-Programs Without SWAP Option	6620 words
Main Program CALLing Sub-Programs With SWAP Option	6304 words

The main program in the final case is very small, so the savings in stack space is not as significant as it could be.

PROCESSING TIME OPTIMIZATION

The following are guidelines for improving the efficiency of your Transact code at run time:

- Adjust the WORK= option of the SYSTEM statement to minimize the work space recoveries during execution. The number of work space recoveries can be determined by running test mode 101 or 102. Adjusting work space size may increase data stack requirements.

- Use DEFINE(INTRINSIC) whenever possible when calling system intrinsics. This construct prevents the Transact processor from using LOADPROC to dynamically return the P-label of the intrinsic being called. The overhead of loading the Transact program is reduced by using DEFINE(INTRINSIC). See Appendix D for a list of allowable intrinsics.

- Avoid calling many separate user-defined procedures from a Transact application. One LOADPROC is executed per procedure, contributing to processing overhead. If possible, combine user-defined procedures into one procedure, and identify the procedure to be executed with a control or index parameter.

- Avoid using the UNLOAD option of the PROC verb with frequently called procedures, since both LOADPROC and UNLOADPROC are called each time a procedure is called.

- Use the NOLOAD option of the PROC verb with such infrequently called procedures as error routines.

- When the 255-entry/process limit of the Loader Segment Table (LST) is likely to be exceeded, UNLOAD can be used to release table entries as appropriate. A preferred approach is to combine user-defined procedures whenever possible.

- Avoid mixing character and block modes during a single application. This mixture requires considerable overhead in VPLUS whenever the switch from block mode to character mode occurs.

- Because only one VPLUS forms file can be opened at a time, processing overhead required to close and open forms files is minimized when only one forms file is used by any program or subprogram.

- Avoid switching between segments in order to minimize the input/output overhead incurred in loading segment information into the data stack. Segments should conform as much as possible to the functional characteristics of the application. Commonly used routines should be grouped in the root segment (segment 0), since this segment is always memory-resident. However, this segment should be as small as possible.

- Minimize the amount of calculations performed. If you need extensive numeric calculations, consider using subroutines in other languages and invoking them with the PROC statement.
- Use the MOVE verb whenever possible to transfer values between data items. The MOVE statement does no data type checks or conversions. The LET verb, however, performs time-consuming data type compatibility checks.
- Place frequently referenced items on the top of the list register, i.e., place them towards the end of the LIST verb statement. The list register is implemented as a linked list with list searches starting from the top of the list. Searches are minimized when the search item is at the top of the list.
- Avoid using fragmented lists when accessing data bases. Transact has to unscramble the list before data base input/output operations are performed.
- Minimize internal sorting of large files.

INDEX

READER COMMENT SHEET

HP 3000 Computer System

TRANSACT/3000

Reference Manual

32247-90001 Dec 1982

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate?	Yes ___	No ___
Are the concepts and wording easy to understand?	Yes ___	No ___
Is the format of this manual convenient in size, arrangement, and readability?	Yes ___	No ___

Comments:

FROM: Date ________

Name ____________________

Company ____________________

Address ____________________

____

____

FOLD FOL

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1070 CUPERTINO,CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
Information Networks Division
19420 Homestead Road
Cupertino, California 95014

FOLD FOLI

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

Product Line Sales/Support Key

Key	Product Line
A	Analytical
CM	Components
C	Computer Systems Sales only
CH	Computer Systems Hardware Sales and Services
CS	Computer Systems Software Sales and Services
E	Electronic Instruments & Measurement Systems
M	Medical Products
MP	Medical Products Primary SRO
MS	Medical Products Secondary SRO
P	Personal Computation Products
*	Sales only for specific product line
**	Support only for specific product line

IMPORTANT: These symbols designate general product line capability. They do not insure sales or support availability for all products within a line, at all locations. Contact your local sales office for information regarding locations where HP support is available for specific products.

HP distributors are printed in italics.

ANGOLA
Telectra
Empresa Técnica de Equipamentos
Eléctricos, S.A.R.L.
R. Barbosa Rodrigues, 41-I DT.
Caixa Postal 6487
LUANDA
Tel: 35515,35516
E,M,P

ARGENTINA
Hewlett-Packard Argentina S.A.
Avenida Santa Fe 2035
Martinez 1640 **BUENOS AIRES**
Tel: 798-5735, 792-1293
Telex: 17595 BIONAR
Cable: HEWPACKARG
A,E,CH,CS,P

Biotron S.A.C.I.M. e I.
Av Paseo Colon 221, Piso 9
*1399 **BUENOS AIRES,***
Tel: 30-4846, 30-1851
Telex: 17595 BIONAR
M

Fate S.A. I.C.I.Electronica
Venezuela 1326
*1095 **BUENOS AIRES***
Tel: 37-9020, 37-9026/9
Telex: 9234 FATEN AR
P

AUSTRALIA

Adelaide, South Australia Office
Hewlett-Packard Australia Ltd.
153 Greenhill Road
PARKSIDE, S.A. 5063
Tel: 272-5911
Telex: 82536
Cable: HEWPARD Adelaide
A*,CH,CM,,E,MS,P

Brisbane, Queensland Office
Hewlett-Packard Australia Ltd.
49 Park Road
MILTON, Queensland 4064
Tel: 229-1544
Telex: 42133
Cable: HEWPARD Brisbane
A,CH,CM,E,M,P
Effective November 1, 1982:
10 Payne Road
THE GAP, Queensland 4061
Tel: 30-4133
Telex: 42133

Canberra, Australia Capital Territory Office
Hewlett-Packard Australia Ltd.
121 Wollongong Street
FYSHWICK, A.C.T. 2609
Tel: 80 4244
Telex: 62650
Cable: HEWPARD Canberra
CH,CM,E,P

Melbourne, Victoria Office
Hewlett-Packard Australia Ltd.
31-41 Joseph Street
BLACKBURN, Victoria 3130
Tel: 877 7777
Telex: 31-024
Cable: HEWPARD Melbourne
A,CH,CM,CS,E,MS,P

Perth, Western Australia Office
Hewlett-Packard Australia Ltd.
261 Stirling Highway
CLAREMONT, W.A. 6010
Tel: 383-2188
Telex: 93859
Cable: HEWPARD Perth
A,CH,CM,,E,MS,P

Sydney, New South Wales Office
Hewlett-Packard Australia Ltd.
17-23 Talavera Road
P.O. Box 308
NORTH RYDE, N.S.W. 2113
Tel: 887-1611
Telex: 21561
Cable: HEWPARD Sydney
A,CH,CM,CS,E,MS,P

AUSTRIA
Hewlett-Packard Ges.m.b.h.
Grottenhofstrasse 94
Verkaufsburo Graz
A-8052 **GRAZ**
Tel: 291-5-66
Telex: 32375
CH,E*

Hewlett-Packard Ges.m.b.h.
Stanglhofweg 5
A-4020 **LINZ**
Tel: 0732 51585
CH

Hewlett-Packard Ges.m.b.h.
Lieblgasse 1
P.O. Box 72
A-1222 **VIENNA**
Tel: (0222) 23-65-11-0
Telex: 134425 HEPA A
A,CH,CM,CS,E,MS,P

BAHRAIN
Green Salon
P.O. Box 557
BAHRAIN
Tel: 255503-255950
Telex: 84419
P

Wael Pharmacy
P.O. Box 648
BAHRAIN
Tel: 256123
Telex: 8550 WAEL BN
M, E

BELGIUM
Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
B-1200 **BRUSSELS**
Tel: (02) 762-32-00
Telex: 23-494 paloben bru
A,CH,CM,CS,E,MP,P

BRAZIL
Hewlett-Packard do Brasil I.e.C. Ltda.
Alameda Rio Negro, 750
Alphaville 06400 **BARUERI SP**
Tel: (11) 421-1311
Telex: 01 133872 HPBR-BR
Cable: HEWPACK Sao Paulo
A,CH,CM,CS,E,M,P

Hewlett-Packard do Brasil I.e.C. Ltda.
Avenida Epitacio Pessoa, 4664
22471 **RIO DE JANEIRO**-RJ
Tel: (21) 286-0237
Telex: 021-21905 HPBR-BR
Cable: HEWPACK Rio de Janeiro
A,CH,CM,E,MS,P*

CANADA

Alberta
Hewlett-Packard (Canada) Ltd.
210, 7220 Fisher Street S.E.
CALGARY, Alberta T2H 2H8
Tel: (403) 253-2713
A,CH,CM,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
11620A-168th Street
EDMONTON, Alberta T5M 3T9
Tel: (403) 452-3670
A,CH,CM,CS,E,MS,P*

British Columbia
Hewlett-Packard (Canada) Ltd.
10691 Shellbridge Way
RICHMOND,
British Columbia V6X 2W7
Tel: (604) 270-2277
Telex: 610-922-5059
A,CH,CM,CS,E*,MS,P*

Manitoba
Hewlett-Packard (Canada) Ltd.
380-550 Century Street
WINNIPEG, Manitoba R3H 0Y1
Tel: (204) 786-6701
A,CH,CM,E,MS,P*

New Brunswick
Hewlett-Packard (Canada) Ltd.
37 Sheadiac Road
MONCTON, New Brunswick E2B 2VQ
Tel: (506) 855-2841
CH**

Nova Scotia
Hewlett-Packard (Canada) Ltd.
P.O. Box 931
900 Windmill Road
DARTMOUTH, Nova Scotia B2Y 3Z6
Tel: (902) 469-7820
CH,CM,CS,E*,MS,P*

Ontario
Hewlett-Packard (Canada) Ltd.
552 Newbold Street
LONDON, Ontario N6E 2S5
Tel: (519) 686-9181
A,CH,CM,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
MISSISSAUGA, Ontario L4V 1M8
Tel: (416) 678-9430
A,CH,CM,CS,E,MP,P

Hewlett-Packard (Canada) Ltd.
2670 Queensview Dr.
OTTAWA, Ontario K2B 8K1
Tel: (613) 820-6483
A,CH,CM,CS,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
220 Yorkland Blvd., Unit #11
WILLOWDALE, Ontario M2J 1R5
Tel: (416) 499-9333
CH

Quebec
Hewlett-Packard (Canada) Ltd.
17500 South Service Road
Trans-Canada Highway
KIRKLAND, Quebec H9J 2M5
Tel: (514) 697-4232
A,CH,CM,CS,E,MP,P*

Hewlett-Packard (Canada) Ltd.
Les Galeries du Vallon
2323 Du Versont Nord
STE. FOY, Quebec G1N 4C2
Tel: (418) 687-4570
CH

CHILE
Jorge Calcagni y Cia. Ltda.
Arturo Burhle 065
Casilla 16475
***SANTIAGO** 9*
Tel: 222-0222
Telex: Public Booth 440001
A,CM,E,M

Olympia (Chile) Ltda.
Av. Rodrigo de Araya 1045
Casilla 256-V
***SANTIAGO** 21*
Tel: 2-25-50-44
Telex: 340-892 OLYMP CK
Cable: Olympiachile Santiagochile
CH,CS,P

CHINA, People's Republic of
China Hewlett-Packard Rep. Office
P.O. Box 418
1A Lane 2, Luchang St.
Beiwei Rd., Xuanwu District
BEIJING
Tel: 33-1947, 33-7426
Telex: 22601 CTSHP CN
Cable: 1920
A,CH,CM,CS,E,P

COLOMBIA
Instrumentación
H. A. Langebaek & Kier S.A.
Carrera 7 No. 48-75
Apartado Aereo 6287
***BOGOTA** 1, D.E.*
Tel: 287-8877
Telex: 44400 INST CO
Cable: AARIS Bogota
A,CM,E,M,PS,P

COSTA RICA
Cientifica Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
SAN JOSÉ
Tel: 24-38-20, 24-08-19
Telex: 2367 GALGUR CR
CM,E,MS,P

CYPRUS
Telerexa Ltd.
P.O. Box 4809
14C Stassinos Avenue
NICOSIA
Tel: 62698
Telex: 2894 LEVIDO CY
E,M,P

DENMARK
Hewlett-Packard A/S
Datavej 52
DK-3460 **Birkerod**
Tel: (02) 81-66-40
Telex: 37409 hpas dk
A,CH,CM,CS,E,MS,P

Hewlett-Packard A/S
Navervej 1
DK-8600 **SILKEBORG**
Tel: (06) 82-71-66
Telex: 37409 hpas dk
CH,E

ECUADOR
CYEDE Cia. Ltda.
Avenida Eloy Alfaro 1749
Casilla 6423 CCI
QUITO
Tel: 450-975, 243-052
Telex: 2548 CYEDE ED
A,CM,E,P

Hospitalar S.A.
Robles 625
Casilla 3590
QUITO
Tel: 545-250, 545-122
Telex: 2485 HOSPTL ED
Cable: HOSPITALAR-Quito
M

EGYPT
International Engineering Associates
24 Hussein Hegazi Street
Kasr-el-Aini
CAIRO
Tel: 23829, 21641
Telex: IEA UN 93830
CH,CS,E,M

Informatic For Systems
22 Talaat Harb Street
CAIRO
Tel: 759006
Telex: 93938 FRANK UN
CH,CS,P

Egyptian International Office
for Foreign Trade
P.O.Box 2558
CAIRO
Tel: 650021
Telex: 93337 EGPOR
P

EL SALVADOR
IPESA de El Salvador S.A.
29 Avenida Norte 1216
SAN SALVADOR
Tel: 26-6858, 26-6868
Telex: Public Booth 20107
A,CH,CM,CS,E,P

FINLAND
Hewlett-Packard Oy
Revontulentie 7
SF-02100 **ESPOO** 10
Tel: (90) 455-0211
Telex: 121563 hewpa sf
A,CH,CM,CS,E,MS,P

Hewlett-Packard Oy
Aatoksenkatv 10-C

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

SF-40720-72 **JYVASKYLA**
Tel: (941) 216318
CH

Hewlett-Packard Oy
Kainvuntie 1-C
SF-90140-14 **OULU**
Tel: (981) 338785
CH

FRANCE

Hewlett-Packard France
Z.I. Mercure B
Rue Berthelot
F-13763 Les Milles Cedex
AIX-EN-PROVENCE
Tel: (42) 59-41-02
Telex: 410770F
A,CH,E,MS,P*

Hewlett-Packard France
Boite Postale No. 503
F-25026 **BESANCON**
28 Rue de la Republique
F-25000 **BESANCON**
Tel: (81) 83-16-22
CH,M

Hewlett-Packard France
Bureau de Vente de Lyon
Chemin des Mouilles
Boite Postale 162
F-69130 **ECULLY** Cédex
Tel: (7) 833-81-25
Telex: 310617F
A,CH,CS,E,MP

Hewlett-Packard France
Immeuble France Evry
Tour Lorraine
Boulevard de France
F-91035 **EVRY** Cédex
Tel: (6) 077-96-60
Telex: 692315F
E

Hewlett-Packard France
5th Avenue Raymond Chanas
F-38320 **EYBENS**
Tel: (76) 25-81-41
Telex: 980124 HP GRENOB EYBE
CH

Hewlett-Packard France
Centre d'Affaire Paris-Nord
Bâtiment Ampère 5 étage
Rue de la Commune de Paris
Boite Postale 300
F-93153 **LE BLANC MESNIL**
Tel: (01) 865-44-52
Telex: 211032F
CH,CS,E,MS

Hewlett-Packard France
Parc d'Activites Cadera
Quartier Jean Mermoz
Avenue du President JF Kennedy
F-33700 **MERIGNAC**
Tel: (56) 34-00-84
Telex: 550105F
CH,E,MS

Hewlett-Packard France
32 Rue Lothaire
F-57000 **METZ**
Tel: (8) 765-53-50
CH

Hewlett-Packard France
Immueble Les 3 B
Nouveau Chemin de la Garde
Z.A.C. de Bois Briand
F-44085 **NANTES** Cedex
Tel: (40) 50-32-22
CH**

Hewlett-Packard France
Zone Industrielle de Courtaboeuf
Avenue des Tropiques
F-91947 Les Ulis Cédex **ORSAY**
Tel: (6) 907-78-25
Telex: 600048F
A,CH,CM,CS,E,MP,P

Hewlett-Packard France
Paris Porte-Maillot
15, Avenue De L'Amiral Bruix
F-75782 **PARIS** 16
Tel: (1) 502-12-20
Telex: 613663F
CH,MS,P

Hewlett-Packard France
2 Allee de la Bourgonette
F-35100 **RENNES**
Tel: (99) 51-42-44
Telex: 740912F
CH,CM,E,MS,P*

Hewlett-Packard France
98 Avenue de Bretagne
F-76100 **ROUEN**
Tel: (35) 63-57-66 CH**,CS

Hewlett-Packard France
4 Rue Thomas Mann
Boite Postale 56
F-67200 **STRASBOURG**
Tel: (88) 28-56-46
Telex: 890141F
CH,E,MS,P*

Hewlett-Packard France
Pericentre de la Cépière
F-31081 **TOULOUSE** Cedex
Tel: (61) 40-11-12
Telex: 531639F
A,CH,CS,E,P*

Hewlett-Packard France
Immeuble Péricentre
F-59658 **VILLENEUVE D'ASCQ** Cedex
Tel: (20) 91-41-25
Telex: 160124F
CH,E,MS,P*

GERMAN FEDERAL REPUBLIC

Hewlett-Packard GmbH
Technisches Büro Berlin
Keithstrasse 2-4
D-1000 **BERLIN** 30
Tel: (030) 24-90-86
Telex: 018 3405 hpbln d
A,CH,E,M,P

Hewlett-Packard GmbH
Technisches Büro Böblingen
Herrenberger Strasse 110
D-7030 **BÖBLINGEN**
Tel: (07031) 667-1
Telex: bbn or
A,CH,CM,CS,E,MP,P

Hewlett-Packard GmbH
Technisches Büro Dusseldorf
Emanuel-Leutze-Strasse 1
D-4000 **DUSSELDORF**
Tel: (0211) 5971-1
Telex: 085/86 533 hpdd d
A,CH,CS,E,MS,P

Hewlett-Packard GmbH
Vertriebszentrale Frankfurt
Berner Strasse 117
Postfach 560 140
D-6000 **FRANKFURT** 56
Tel: (0611) 50-04-1
Telex: 04 13249 hpffm d
A,CH,CM,CS,E,MP,P

Hewlett-Packard GmbH
Technisches Büro Hamburg
Kapstadtring 5
D-2000 **HAMBURG** 60
Tel: (040) 63804-1
Telex: 021 63 032 hphh d
A,CH,CS,E,MS,P

Hewlett-Packard GmbH
Technisches Büro Hannover
Am Grossmarkt 6
D-3000 **HANNOVER** 91
Tel: (0511) 46-60-01
Telex: 092 3259
A,CH,CM,E,MS,P

Hewlett-Packard GmbH
Technisches Büro Mannheim
Rosslauer Weg 2-4
D-6800 **MANNHEIM**
Tel: (0621) 70050
Telex: 0462105
A,C,E

Hewlett-Packard GmbH
Technisches Büro Neu Ulm
Messerschmittstrasse 7
D-7910 **NEU ULM**
Tel: 0731-70241
Telex: 0712816 HP ULM-D
A,C,E*

Hewlett-Packard GmbH
Technisches Büro Nürnberg
Neumeyerstrasse 90
D-8500 **NÜRNBERG**
Tel: (0911) 52 20 83-87
Telex: 0623 860
CH,CM,E,MS,P

Hewlett-Packard GmbH
Technisches Büro München
Eschenstrasse 5
D-8028 **TAUFKIRCHEN**
Tel: (089) 6117-1
Telex: 0524985
A,CH,CM,E,MS,P

GREAT BRITAIN

Hewlett-Packard Ltd.
Trafalgar House
Navigation Road
ALTRINCHAM
Chesire WA14 1NU
Tel: (061) 928-6422
Telex: 668068
A,CH,CS,E,M

Hewlett-Packard Ltd.
Oakfield House, Oakfield Grove
Clifton
BRISTOL BS8 2BN, Avon
Tel: (027) 38606
Telex: 444302
CH,M,P

Hewlett-Packard Ltd.
(Pinewood)
Nine Mile Ride
EASTHAMPSTEAD
Wokingham
Berkshire, 3RG11 3LL
Tel: 3446 3100
Telex: 84-88-05
CH,CS,E

Hewlett-Packard Ltd.
Fourier House
257-263 High Street
LONDON COLNEY
Herts., AL2 1HA, St. Albans
Tel: (0727) 24400
Telex: 1-8952716
CH,CS,E

Hewlett-Packard Ltd
Tradax House, St. Mary's Walk
MAIDENHEAD
Berkshire, SL6 1ST
Tel: (0628) 39151
CH,CS,E,P

Hewlett-Packard Ltd.
Quadrangle
106-118 Station Road
REDHILL, Surrey
Tel: (0737) 68655
Telex: 947234 CH,CS,E

Hewlett-Packard Ltd.
Avon House
435 Stratford Road
SHIRLEY, Solihull
West Midlands B90 4BL
Tel: (021) 745 8800
Telex: 339105
CH

Hewlett-Packard Ltd.
West End House 41
High Street, West End
SOUTHAMPTON
Hampshire S03 3DQ
Tel: (703) 886767
Telex: 477138
CH

Hewlett-Packard Ltd.
King Street Lane
WINNERSH, Wokingham
Berkshire RG11 5AR
Tel: (0734) 784774
Telex: 847178
A,CH,E,M

GREECE

Kostas Karaynnis S.A.
8 Omirou Street
***ATHENS** 133*
Tel: 32 30 303, 32 37 371
Telex: 215962 RKAR GR
A,CH,CM,CS,E,M,P

PLAISIO S.A.
G. Gerardos
24 Stournara Street
ATHENS
Tel: 36-11-160
Telex: 221871
P

GUATEMALA

IPESA
Avenida Reforma 3-48, Zona 9
GUATEMALA CITY
Tel: 316627, 314786
Telex: 4192 TELTRO GU
A,CH,CM,CS,E,M,P

HONG KONG

Hewlett-Packard Hong Kong, Ltd.
G.P.O. Box 795
5th Floor, Sun Hung Kai Centre
30 Harbour Road
HONG KONG
Tel: 5-8323211
Telex: 66678 HEWPA HX
Cable: HEWPACK HONG KONG
E,CH,CS,P

CET Ltd.
1402 Tung Way Mansion
199-203 Hennessy Rd.
*Wanchia, **HONG KONG***
Tel: 5-729376
Telex: 85148 CET HX
CM

Schmidt & Co. (Hong Kong) Ltd.
Wing On Centre, 28th Floor
Connaught Road, C.
HONG KONG
Tel: 5-455644
Telex: 74766 SCHMX HX
A,M

ICELAND

Elding Trading Company Inc.
Hafnarnvoli-Tryggvagotu
P.O. Box 895
IS-REYKJAVIK
Tel: 1-58-20, 1-63-03
M

INDIA

Blue Star Ltd.
Sabri Complex II Floor
24 Residency Rd.
***BANGALORE** 560 025*
Tel: 55660
Telex: 0845-430
Cable: BLUESTAR
A,CH,CM,CS,E

Blue Star Ltd.
Band Box House
Prabhadevi
***BOMBAY** 400 025*
Tel: 422-3101
Telex: 011-3751
Cable: BLUESTAR
A,M

Blue Star Ltd.
Sahas
414/2 Vir Savarkar Marg
Prabhadevi
***BOMBAY** 400 025*
Tel: 422-6155
Telex: 011-4093
Cable: FROSTBLUE
A,CH,CM,CS,E,M

Blue Star Ltd.
Kalyan, 19 Vishwas Colony
*Alkapuri, **BORODA**, 390 005*
Tel: 65235
Cable: BLUE STAR
A

Blue Star Ltd.
7 Hare Street
***CALCUTTA** 700 001*
Tel: 12-01-31
Telex: 021-7655
Cable: BLUESTAR
A,M

Blue Star Ltd.
133 Kodambakkam High Road
***MADRAS** 600 034*
Tel: 82057
Telex: 041-379
Cable: BLUESTAR
A,M

Blue Star Ltd.
Bhandari House, 7th/8th Floors
91 Nehru Place
***NEW DELHI** 110 024*
Tel: 682547
Telex: 031-2463
Cable: BLUESTAR
A,CH,CM,CS,E,M

Blue Star Ltd.
15/16:C Wellesley Rd.
***PUNE** 411 011*
Tel: 22775
Cable: BLUE STAR
A

Blue Star Ltd.
2-2-47/1108 Bolarum Rd.
***SECUNDERABAD** 500 003*
Tel: 72057
Telex: 0155-459
Cable: BLUEFROST
A,E

Blue Star Ltd.
T.C. 7/603 Poornima
Maruthankuzhi
***TRIVANDRUM** 695 013*
Tel: 65799
Telex: 0884-259
Cable: BLUESTAR
E

INDONESIA

BERCA Indonesia P.T.
P.O.Box 496/JKT.
Jl. Abdul Muis 62
JAKARTA
Tel: 373009
Telex: 46748 BERSAL IA
Cable: BERSAL JAKARTA
P

BERCA Indonesia P.T.
Wisma Antara Bldg., 17th floor
JAKARTA
A,CS,E,M

BERCA Indonesia P.T.
P.O. Box 174/SBY.
Jl. Kutei No. 11
SURABAYA
Tel: 68172
Telex: 31146 BERSAL SB
Cable: BERSAL-SURABAYA
A,E,M,P*

IRAQ
Hewlett-Packard Trading S.A.
Service Operation
Al Mansoor City 9B/3/7
BAGHDAD
Tel: 551-49-73
Telex: 212-455 HEPAIRAQ IK
CH,CS

IRELAND
Hewlett-Packard Ireland Ltd.
82/83 Lower Leeson St.
DUBLIN 2
Tel: (1) 60 88 00
Telex: 30439
A,CH,CM,CS,E,M,P

Cardiac Services Ltd.
Kilmore Road
Artane
***DUBLIN** 5*
Tel: (01) 351820
Telex: 30439
M

ISRAEL
Eldan Electronic Instrument Ltd.
P.O. Box 1270
JERUSALEM 91000
16, Ohaliav St.
JERUSALEM 94467
Tel: 533 221, 553 242
Telex: 25231 AB/PAKRD IL
A

Electronics Engineering Division
Motorola Israel Ltd.
16 Kremenetski Street
P.O. Box 25016
TEL-AVIV 67899
Tel: 3-338973
Telex: 33569 Motil IL
Cable: BASTEL Tel-Aviv
CH,CM,CS,E,M,P

ITALY
Hewlett-Packard Italiana S.p.A.
Traversa 99C
Via Giulio Petroni, 19
I-70124 **BARI**
Tel: (080) 41-07-44
M

Hewlett-Packard Italiana S.p.A.
Via Martin Luther King, 38/111
I-40132 **BOLOGNA**
Tel: (051) 402394
Telex: 511630
CH,E,MS

Hewlett-Packard Italiana S.p.A.
Via Principe Nicola 43G/C
I-95126 **CATANIA**
Tel: (095) 37-10-87
Telex: 970291
C,P

Hewlett-Packard Italiana S.p.A.
Via G. Di Vittorio 9
I-20063 **CERNUSCO SUL NAVIGLIO**
Tel: (2) 903691
Telex: 334632
A,CH,CM,CS,E,MP,P

Hewlett-Packard Italiana S.p.A.
Via Nuova San Rocco a
Capodimonte, 62/A
I-80131 **NAPLES**
Tel: (081) 7413544
Telex: 710698
A,CH,E

Hewlett-Packard Italiana S.p.A.
Viale G. Modugno 33
I-16156 **GENOVA PEGLI**
Tel: (010) 68-37-07
Telex: 215238
E,C

Hewlett-Packard Italiana S.p.A.
Via Turazza 14
I-35100 **PADOVA**
Tel: (049) 664888
Telex: 430315
A,CH,E,MS

Hewlett-Packard Italiana S.p.A.
Viale C. Pavese 340
I-00144 **ROMA**
Tel: (06) 54831
Telex: 610514
A,CH,CM,CS,E,MS,P*

Hewlett-Packard Italiana S.p.A.
Corso Svizzera, 184
I-10149 **TORINO**
Tel: (011) 74 4044
Telex: 221079
CH,E

JAPAN
Yokogawa-Hewlett-Packard Ltd.
Inoue Building
1-21-8, Asahi-cho
ATSUGI, Kanagawa 243
Tel: (0462) 28-0451
CM,C*,E

Yokogawa-Hewlett-Packard Ltd.
Towa Building
2-2-3, Kaigandori, Chuo-ku
KOBE, 650, Hyogo
Tel: (078) 392-4791
C,E

Yokogawa-Hewlett-Packard Ltd.
Kumagaya Asahi Yasoji Bldg 4F
3-4 Chome Tsukuba
KUMAGAYA, Saitama 360
Tel: (0485) 24-6563
CH,CM,E

Yokogawa-Hewlett-Packard Ltd.
Asahi Shinbun Dai-ichi Seimei Bldg., 2F
4-7 Hanabata-cho
KUMAMOTO-SHI,860
Tel: (0963) 54-7311
CH,E

Yokogawa-Hewlett-Packard Ltd.
Shin Kyoto Center Bldg. 5F
614 Siokoji-cho
Nishiiruhigashi, Karasuma
Siokoji-dori, Shimogyo-ku
KYOTO 600
Tel: 075-343-0921
CH,E

Yokogawa-Hewlett-Packard Ltd.
Mito Mitsui Building
1-4-73, San-no-maru
MITO, Ibaragi 310
Tel: (0292) 25-7470
CH,CM,E

Yokogawa-Hewlett-Packard Ltd.
Sumitomo Seimei Nagoya Bldg.
2-14-19, Meieki-Minami,
Nakamura-ku
NAGOYA, 450 Aichi
Tel: (052) 571-5171
CH,CM,CS,E,MS

Yokogawa-Hewlett-Packard Ltd.
Chuo Bldg., 4th Floor
5-4-20 Nishinakajima,
Yodogawa-ku
OSAKA, 532
Tel: (06) 304-6021
Telex: YHPOSA 523-3624
A,CH,CM,CS,E,MP,P*

Yokogawa-Hewlett-Packard Ltd.
1-27-15, Yabe,
SAGAMIHARA Kanagawa, 229
Tel: 0427 59-1311

Yokogawa-Hewlett-Packard Ltd.
Shinjuku Dai-ichi Seimei 6F
2-7-1, Nishi Shinjuku
Shinjuku-ku, **TOKYO** 160
Tel: 03-348-4611-5
CH,E

Yokogawa-Hewlett-Packard Ltd.
3-29-21 Takaido-Higashi
Suginami-ku **TOKYO** 168
Tel: (03) 331-6111
Telex: 232-2024 YHPTOK
A,CH,CM,CS,E,MP,P*

Yokogawa-Hewlett-Packard Ltd.
Daiichi Asano Building 4F
5-2-8, Oodori,
UTSUNOMIYA, 320
Tochigi
Tel: (0286) 25-7155
CH, CS, E

Yokogawa-Hewlett-Packard Ltd.
Yasudaseimei Yokohama
Nishiguchi Bldg.
3-30-4 Tsuruya-cho
Kanagawa-ku
YOKOHAMA, Kanagawa, 221
Tel: (045) 312-1252
CH,CM,E

JORDAN
Mouasher Cousins Company
P.O. Box 1387
AMMAN
Tel: 24907, 39907
Telex: 21456 SABCO JO
CH,E,M,P

KENYA
ADCOM Ltd., Inc., Kenya
P.O.Box 30070
NAIROBI
Tel: 331955
Telex: 22639
E,M

KOREA
Samsung Electronics Computer Division
76-561 Yeoksam-Dong
Kangnam-Ku
C.P.O. Box 2775
SEOUL
Tel: 555-7555, 555-5447
Telex: K27364 SAMSAN
A,CH,CM,CS,E,M,P

KUWAIT
Al-Khaldiya Trading & Contracting
P.O. Box 830 Safat
KUWAIT
Tel: 42-4910, 41-1726
Telex: 22481 Areeg kt
CH,E,M

Photo & Cine Equipment
P.O. Box 270 Safat
KUWAIT
Tel: 42-2846, 42-3801
Telex: 22247 Matin-KT
P

LEBANON
G.M. Dolmadjian
Achrafieh
P.O. Box 165.167
BEIRUT
Tel: 290293
*MP***

LUXEMBOURG
Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
B-1200 **BRUSSELS**
Tel: (02) 762-32-00
Telex: 23-494 paloben bru
A,CH,CM,CS,E,MP,P

MALAYSIA
Hewlett-Packard Sales (Malaysia)
Sdn. Bhd.
1st Floor, Bangunan British
American
Jalan Semantan, Damansara Heights
KUALA LUMPUR 23-03
Tel: 943022
Telex: MA31011
A,CH,E,M,P*

Protel Engineering
Lot 319, Satok Road
P.O.Box 1917
*Kuching, **SARAWAK***
Tel: 53544
Telex: MA 70904 PROMAL
Cable: PROTELENG
A,E,M

MALTA
Philip Toledo Ltd.
Notabile Rd.
MRIEHEL
Tel: 447 47, 455 66
Telex: 649 Media MW
P

MEXICO
Hewlett-Packard Mexicana, S.A. de C.V.
Av. Periferico Sur No. 6501
Tepepan, Xochimilco
MEXICO D.F. 16020
Tel: 676-4600
Telex: 17-74-507 HEWPACK MEX
A,CH,CS,E,MS,P

Effective November 1, 1982:
Hewlett-Packard Mexicana, S.A. de C.V.
Ejercito Nacional #570
Colonia Granada
11560 **MEXICO**, D.F.
CH**

Hewlett-Packard Mexicana, S.A. de C.V.
Rio Volga 600
Pte. Colonia del Valle
MONTERREY, N.L.
Tel: 78-42-93, 78-42-40, 78-42-41
Telex: 038-2410 HPMTY ME
CH

Effective Nov. 1, 1982
Ave. Colonia del Valle #409
Col. del Valle
Municinio de garza garcia
MONTERREY, N.V.

ECISA
Taihe 229, Piso 10
*Polanco **MEXICO D.F.** 11570*
Tel: 250-5391
Telex: 17-72755 ECE ME
M

MOROCCO
Dolbeau
81 rue Karatchi
CASABLANCA
Tel: 3041-82, 3068-38
Telex: 23051, 22822
E

Gerep
2 rue d'Agadir
Boite Postale 156
CASABLANCA
Tel: 272093, 272095
Telex: 23 739
P

NETHERLANDS
Hewlett-Packard Nederland B.V.
Van Heuven Goedhartlaan 121
NL 1181KK **AMSTELVEEN**
P.O. Box 667
NL1180 AR **AMSTELVEEN**
Tel: (20) 47-20-21
Telex: 13 216
A,CH,CM,CS,E,MP,P

Hewlett-Packard Nederland B.V.
Bongerd 2
NL 2906VK **CAPPELLE**, A/D Ijessel
P.O. Box 41
NL2900 AA **CAPELLE**, Ijssel
Tel: (10) 51-64-44
Telex: 21261 HEPAC NL
A,CH,CS

NEW ZEALAND
Hewlett-Packard (N.Z.) Ltd.
169 Manukau Road
P.O. Box 26-189
Epsom, **AUCKLAND**
Tel: 687-159
Cable: HEWPACK Auckland
CH,CM,E,P*

Hewlett-Packard (N.Z.) Ltd.
4-12 Cruickshank Street
Kilbirnie, **WELLINGTON 3**
P.O. Box 9443
Courtenay Place, **WELLINGTON 3**
Tel: 877-199
Cable: HEWPACK Wellington
CH,CM,E,P

Northrop Instruments & Systems Ltd.
369 Khyber Pass Road
P.O. Box 8602
AUCKLAND
Tel: 794-091
Telex: 60605
A,M

Northrop Instruments & Systems Ltd.
110 Mandeville St.
P.O. Box 8388
CHRISTCHURCH
Tel: 486-928
Telex: 4203
A,M

Northrop Instruments & Systems Ltd.
Sturdee House
85-87 Ghuznee Street
P.O. Box 2406
WELLINGTON
Tel: 850-091
Telex: NZ 3380
A,M

NORTHERN IRELAND
Cardiac Services Company
95A Finaghy Road South
***BELFAST** BT 10 OBY*
Tel: (0232) 625-566
Telex: 747626
M

NORWAY
Hewlett-Packard Norge A/S
Folke Bernadottes vei 50
P.O. Box 3558
N-5033 **FYLLINGSDALEN** (Bergen)
Tel: (05) 16-55-40
Telex: 16621 hpnas n
CH,CS,E,MS

Hewlett-Packard Norge A/S
Österndalen 18
P.O. Box 34
N-1345 **ÖSTERÅS**
Tel: (02) 17-11-80
Telex: 16621 hpnas n
A,CH,CM,CS,E,M,P

OMAN
Khimjil Ramdas
P.O. Box 19
MUSCAT
Tel: 722225, 745601
Telex: 3289 BROKER MB MUSCAT
P

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

Suhail & Saud Bahwan
P.O. Box 169
MUSCAT
Tel: 734 201-3
Telex: 3274 BAHWAN MB

PAKISTAN
Mushko & Company Ltd.
1-B, Street 43
Sector F-8/1
ISLAMABAD
Tel: 26875
Cable: FEMUS Rawalpindi
A,E,M

Mushko & Company Ltd.
Oosman Chambers
Abdullah Haroon Road
KARACHI 0302
Tel: 511027, 512927
Telex: 2894 MUSKO PK
Cable: COOPERATOR Karachi
A,E,M,P*

PANAMA
Electrónico Balboa, S.A.
Calle Samuel Lewis, Ed. Alfa
Apartado 4929
PANAMA 5
Tel: 64-2700
Telex: 3483 ELECTRON PG
A,CM,E,M,P

Foto Internacional, S.A.
Colon Free Zone
Apartado 2068
COLON 3
Tel: 45-2333
Telex: 8626 IMPORT PG
P

PERU
Cía Electro Médica S.A.
Los Flamencos 145, San Isidro
Casilla 1030
LIMA 1
Tel: 41-4325, 41-3703
Telex: Pub. Booth 25306
A,CM,E,M,P

PHILIPPINES
The Online Advanced Systems Corporation
Rico House, Amorsolo Cor. Herrera Street
Legaspi Village, Makati
P.O. Box 1510
Metro MANILA
Tel: 85-35-81, 85-34-91, 85-32-21
Telex: 3274 ONLINE
A,CH,CS,E,M

Electronic Specialists and Proponents Inc.
690-B Epifanio de los Santos Avenue
Cubao, QUEZON CITY
P.O. Box 2649 Manila
Tel: 98-96-81, 98-96-82, 98-96-83
Telex: 40018, 42000 ITT GLOBE MACKAY BOOTH
P

PORTUGAL
Mundinter
Intercambio Mundial de Comércio S.a.r.l
P.O. Box 2761
Av. Antonio Augusto de Aguiar 138
P-LISBON
Tel: (19) 53-21-31, 53-21-37
Telex: 16691 munter p
M

Soquimica
Av. da Liberdade, 220-2
1298 LISBON Codex
Tel: 56 21 81/2/3
Telex: 13316 SABASA P

Telectra-Empresa Técnica de Equipmentos Eléctricos S.a.r.l.
Rua Rodrigo da Fonseca 103
P.O. Box 2531
P-LISBON 1
Tel: (19) 68-60-72
Telex: 12598
CH,CS,E,P

PUERTO RICO
Hewlett-Packard Puerto Rico
P.O. Box 4407
CAROLINA, Puerto Rico 00628
Calle 272 Edificio 203
Urb. Country Club
RIO PIEDRAS, Puerto Rico 00924
Tel: (809) 762-7255
A,CH,CS

QATAR
Nasser Trading & Contracting
P.O. Box 1563
DOHA
Tel: 22170, 23539
Telex: 4439 NASSER DH
M

Computearbia
P.O. Box 2750
DOHA
Tel: 883555
Telex: 4806 CHPARB
P

Eastern Technical Services
P.O. Box 4747
DOHA
Tel: 329 993
Telex: 4156 EASTEC DH

SAUDI ARABIA
Modern Electronic Establishment
Hewlett-Packard Division
P.O. Box 281
Thuobah
AL-KHOBAR
Tel: 864-46 78
Telex: 671 106 HPMEEK SJ
Cable: ELECTA AL-KHOBAR
CH,CS,E,M,P

Modern Electronic Establishment
Hewlett-Packard Division
P.O. Box 1228
Redec Plaza, 6th Floor
JEDDAH
Tel: 644 38 48
Telex: 402712 FARNAS SJ
Cable: ELECTA JEDDAH
CH,CS,E,M,P

Modern Electronic Establishment
Hewlett Packard Division
P.O. Box 2728
RIYADH
Tel: 491-97 15, 491-63 87
Telex: 202049 MEERYD SJ
CH,CS,E,M,P

SCOTLAND
Hewlett-Packard Ltd.
Royal Bank Buildings
Swan Street
BRECHIN, Angus, Scotland
Tel: (03562) 3101-2
CH

Hewlett-Packard Ltd.
SOUTH QUEENSFERRY
West Lothian, EH30 9GT
GB-Scotland
Tel: (031) 3311188
Telex: 72682
A,CH,CM,CS,E,M

SINGAPORE
Hewlett-Packard Singapore (Pty.) Ltd.
P.O. Box 58 Alexandra Post Office
SINGAPORE, 9115
6th Floor, Inchcape House
450-452 Alexandra Road
SINGAPORE 0511
Tel: 631788
Telex: HPSGSO RS 34209
Cable: HEWPACK, Singapore
A,CH,CS,E,MS,P

Dynamar International Ltd.
Unit 05-11 Block 6
Kolam Ayer Industrial Estate
SINGAPORE 1334
Tel: 747-6188
Telex: RS 26283
CM

SOUTH AFRICA
Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 120
Howard Place
Pine Park Center, Forest Drive,
Pinelands
CAPE PROVINCE 7405
Tel: 53-7954
Telex: 57-20006
A,CH,CM,E,MS,P

Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 37099
92 Overport Drive
DURBAN 4067
Tel: 28-4178, 28-4179, 28-4110
Telex: 6-22954
CH,CM

Hewlett-Packard So Africa (Pty.) Ltd.
6 Linton Arcade
511 Cape Road
Linton Grange
PORT ELIZABETH 6001
Tel: 041-302148
CH

Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 33345
Glenstantia 0010 TRANSVAAL
1st Floor East
Constantia Park Ridge Shopping Centre
Constantia Park
PRETORIA
Tel: 982043
Telex: 32163
CH,E

Hewlett-Packard So Africa (Pty.) Ltd.
Private Bag Wendywood
SANDTON 2144
Tel: 802-5111, 802-5125
Telex: 4-20877
Cable: HEWPACK Johannesburg
A,CH,CM,CS,E,MS,P

SPAIN
Hewlett-Packard Española S.A.
c/Entenza, 321
E-BARCELONA 29
Tel: (3) 322-24-51, 321-73-54
Telex: 52603 hpbee
A,CH,CS,E,MS,P

Hewlett-Packard Española S.A.
c/San Vicente S/N
Edificio Albia II,7 B
E-BILBAO 1
Tel: (4) 23-8306, (4) 23-8206
A,CH,E,MS

Hewlett-Packard Española S.A.
Calle Jerez 3
E-MADRID 16
Tel: (1) 458-2600
Telex: 23515 hpe
A,CM,E

Hewlett-Packard Española S.A.
c/o Costa Brava 13
Colonia Mirasierra
E-MADRID 34
Tel: (1) 734-8061, (1) 734-1162
CH,CS,M

Hewlett-Packard Española S.A.
Av Ramón y Cajal 1-9
Edificio Sevilla 1,
E-SEVILLA 5
Tel: 64-44-54, 64-44-58
Telex: 72933
A,CS,MS,P

Hewlett-Packard Española S.A.
C/Ramon Gordillo, 1 (Entlo.3)
E-VALENCIA 10
Tel: 361-1354, 361-1358
CH,P

SWEDEN
Hewlett-Packard Sverige AB
Sunnanvagen 14K
S-22226 LUND
Tel: (046) 13-69-79
Telex: (854) 17886 (via SPÅNGA office)
CH

Hewlett-Packard Sverige AB
Vastra Vintergatan 9
S-70344 OREBRO
Tel: (19) 10-48-80
Telex: (854) 17886 (via SPÅNGA office)
CH

Hewlett-Packard Sverige AB
Skalholtsgatan 9, Kista
Box 19
S-16393 SPÅNGA
Tel: (08) 750-2000
Telex: (854) 17886
A,CH,CM,CS,E,MS,P

Hewlett-Packard Sverige AB
Frötallisgatan 30
S-42132 VÄSTRA-FRÖLUNDA
Tel: (031) 49-09-50
Telex: (854) 17886 (via SPÅNGA office)
CH,E,P

SWITZERLAND
Hewlett-Packard (Schweiz) AG
Clarastrasse 12
CH-4058 BASLE
Tel: (61) 33-59-20
A

Hewlett-Packard (Schweiz) AG
Bahnhoheweg 44
CH-3018 BERN
Tel: (031) 56-24-22
CH

Hewlett-Packard (Schweiz) AG
47 Avenue Blanc
CH-1202 GENEVA
Tel: (022) 32-48-00
CH,CM,CS

Hewlett-Packard (Schweiz) AG
19 Chemin Château Bloc
CH-1219 LE LIGNON-Geneva
Tel: (022) 96-03-22
Telex: 27333 hpag ch
Cable: HEWPACKAG Geneva
A,E,MS,P

Hewlett-Packard (Schweiz) AG
Allmend 2
CH-8967 WIDEN
Tel: (57) 31 21 11
Telex: 53933 hpag ch
Cable: HPAG CH
A,CH,CM,CS,E,MS,P

SYRIA
General Electronic Inc.
Nuri Basha
P.O. Box 5781
DAMASCUS
Tel: 33-24-87
Telex: 11216 ITIKAL SY
Cable: ELECTROBOR DAMASCUS
E

Middle East Electronics
Place Azmé
Boite Postale 2308
DAMASCUS
Tel: 334592
Telex: 11304 SATACO SY
M,P

TAIWAN
Hewlett-Packard Far East Ltd.
Kaohsiung Office
2/F 68-2, Chung Cheng 3rd Road
KAOHSIUNG
Tel: 241-2318, 261-3253
CH,CS,E

Hewlett-Packard Far East Ltd.
Taiwan Branch
5th Floor
205 Tun Hwa North Road
TAIPEI
Tel:(02) 751-0404
Cable:HEWPACK Taipei
A,CH,CM,CS,E,M,P

Ing Lih Trading Co.
3rd Floor, 7 Jen-Ai Road, Sec. 2
TAIPEI 100
Tel: (02) 3948191
Cable: INGLIH TAIPEI
A

THAILAND
Unimesa
30 Patpong Ave., Suriwong
BANGKOK 5
Tel: 234 091, 234 092
Telex: 84439 Simonco TH
Cable: UNIMESA Bangkok
A,CH,CS,E,M

Bangkok Business Equipment Ltd.
5/5-6 Dejo Road
BANGKOK
Tel: 234-8670, 234-8671
Telex: 87669-BEQUIPT TH
Cable: BUSIQUIPT Bangkok
P

TRINIDAD & TOBAGO
Caribbean Telecoms Ltd.
50/A Jerningham Avenue
P.O. Box 732
PORT-OF-SPAIN
Tel: 62-44213, 62-44214
Telex: 235,272 HUGCO WG
A,CM,E,M,P

TUNISIA
Tunisie Electronique
31 Avenue de la Liberte
TUNIS
Tel: 280-144
E,P

Corema
1 ter. Av. de Carthage
TUNIS
Tel: 253-821
Telex: 12319 CABAM TN
M

TURKEY
Teknim Company Ltd.
Iran Caddesi No. 7
Kavaklidere, ANKARA
Tel: 275800
Telex: 42155 TKNM TR
E

E.M.A.
Medina Eldem Sokak No.41/6
Yuksel Caddesi
ANKARA
Tel: 175 622
M

UNITED ARAB EMIRATES
Emitac Ltd.
P.O. Box 1641
SHARJAH
Tel: 354121, 354123
Telex: 68136 Emitac Sh
CH,CS,E,M,P