HP 3000 Computer Systems

HP Transact Documentation Update Notice

HEWLETT PACKARD

HP Part No. 32247-90028
Printed in U.S.A.

Seventh Edition, Update #1
E1096

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	December 1981	32247A.00.00
Second Edition	December 1982	32247A.00.03
Update #1	June 1983	32247A.01.01
Update #2	February 1985	32247A.02.02
Fourth Edition	October 1987	32247A.03.07
Update #1	July 1988	32247A.06.00 & 30138A.00.00
Fifth Edition	February 1990	32247A.07.02 & 30138A.02.01
Sixth Edition	September 1992	32247A.09.00 & 30138A.04.00
Seventh Edition	April 1994	32247A.10.00 & 30138A.05.00
Update #1	October 1996	32247A.11.00 & 30138A.06.00

About This Manual

This manual is a reference for programming in the Transact programming language. It assumes that you have a working knowledge of computer programming and the HP 3000 computer system, including the subsystems TurboIMAGE and VPLUS. The manual contains the following chapters and appendixes:

- Chapter 1, "Introduction to Transact," describes the features and benefits of Transact.
- Chapter 2, "Program Structure," describes the program structure of Transact.
- Chapter 3, "Data Items," discusses data item definitions, names, types, sizes, as well as parent and child items, compound items, array subscripting, and defining and handling arrays.
- Chapter 4, "Transact Registers," describes registers, the areas of data storage in Transact, and how they work.
- Chapter 5, "User Interface," describes the three modes of user interface: command sequence, character mode, and block mode using VPLUS.
- Chapter 6, "Accessing Databases and Files," describes how to use databases, KSAM files, and MPE files with Transact.
- Chapter 7, "Error Handling," explains the error handling process and the effect of the STATUS option on various verbs.
- Chapter 8, "Verbs," provides detailed descriptions of the Transact verbs.
- Chapter 9, "Running Transact," tells how to compile and execute Transact programs and control execution at run time.
- Chapter 10, "Transact Test Facility," explains how to use the test facility, which is a major aid in program testing, integration, and optimization.
- Chapter 11, "TRANDEBUG," describes Transact/iX's symbolic debugging facility. It also provides a tutorial introduction to using the debugger and a dictionary of all TRANDEBUG commands.
- Appendix A, "Flowcharts of File and Database Operations," contains flowcharts showing the file and database procedures called when Transact verbs perform file and database operations.
- Appendix B, "Transact/iX Migration Guide," provides guidelines for migrating Transact/V programs to native mode Transact/iX programs on an MPE/iX system.
- Appendix C, "Optimizing Transact Applications," provides guidelines for optimizing the run-time performance and efficiency of Transact applications.
- Appendix D, "Architected Call Interface," explains how to call existing Transact/iX subprograms from COBOL or Pascal.
- Appendix E, "Native Language Support," describes how Transact provides access to MPE native language support at compile time and run time.

Introducing MPE/iX

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In Hewlett-Packard documentation and in talking with other HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a supersest of MPE XL. All programs written for MPE XL will run without change under MPE/iX, and you can continue to use MPE XL system documentation.

Finally, you may encounter references to MPE V, an HP 3000 operating system that is not based on the PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as compatibility mode (CM).

Transact Enhancements

This edition of the manual includes descriptions of the enhancements that have been made to Transact. Here is a list of these enhancements and where they are located in the manual.

Enhancement	Location
ALIGN Option for LIST	Chapter 8
ASCII Function for LET	Chapter 8
CALL, STATUS	Chapter 8
CHAR Function for MOVE	Chapter 8
COL Function for MOVE	Chapter 8
Expand Intrinsic Support of DEFINE(INTRINSIC)	Chapter 8
LENGTH Function for LET	Chapter 8
LOWER Function for MOVE	Chapter 8
POSITION Function for LET	Chapter 8
PROPER Function for MOVE	Chapter 8
PROPER Modifier for SET and RESET	Chapter 8
SPACE Function for MOVE	Chapter 8
STRING Function for MOVE	Chapter 8
UPPER Function for MOVE	Chapter 8
VALUE Function for LET	Chapter 8
WORKFILE Option for FIND	Chapter 8
CHCK Compiler Option	Chapter 9

This update to the manual includes two additional enhancements to Transact. These enhancements and their location in the manual are:

- Literal string parameters and numeric constant parameters can now be passed in the PROC verb. Also, for Transact/iX only, default values are passed for null parameters in option-extensible system routines. See Chapter 8.
- A method is provided for increased decimal precision in arithmetic expressions (IF, LET, REPEAT, and WHILE verbs) via the new !PRECISION compiler command. See Chapters 8 and 9.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Substantive changes are marked with a vertical bar in the margin. Changes due simply to changes in pagination or the correction of typographical errors may or may not be so dated and marked. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

Effective Pages	**Date**
ii-b to ii-c (this "List of Effective Pages")	Oct 1996
iii to iv	Oct 1996
3-9 to 3-10	Oct 1996
6-7 to 6-8	Oct 1996
7-1 to 7-2	Oct 1996
7-11 to 7-12	Oct 1996
8	Oct 1996
9	Oct 1996
11-41 to 11-42	Oct 1996
B-5 to B-6	Oct 1996
B-9 to B-10	Oct 1996
D-3 to D-4	Oct 1996
Index-1 to Index-14	Oct 1996

Data Types and Databases

There are several differences between the data types for databases and those for Transact. The main difference is that databases require all data items to be defined as whole words on word (16-bit) boundaries. To maintain consistency, you can define a data item in Transact with an odd number of bytes, but specify that the data item be stored in whole words. For example, you can define a data item in Transact as 9(5,0,6) to specify 5 digits, stored as 6 bytes.

This example illustrates the second difference between databases and Transact data types. Databases do not have a numeric ASCII string data type. This difference does not cause problems. Transact automatically converts any numeric ASCII (data type 9) data items to alphanumerics (data type X) before use. When data is transferred into a Transact type 9 data item, Transact checks to make sure the data is numeric.

Data Types and Data Dictionaries

You can create a data dictionary in which you define the data items, databases, forms files, MPE files, and KSAM files to be used in Transact programs. The use of a data dictionary as a central location for data definitions and attributes allows you to change existing definitions and attributes easily and dynamically. The data dictionary does not supply the data itself, which must come from MPE or KSAM files, databases, forms files, or the user.

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

Transact allows you to use either Dictionary/V or System Dictionary or both in one program. If you do not specify, Transact assumes Dictionary/V, by default. To use System Dictionary, you must include special compiler commands in your source file. These commands are described in Chapter 9.

When Transact takes data item definitions from System Dictionary, only attributes defined at the data item level can be accessed. Any attributes defined at the relationship levels are inaccessible, since Transact commands can include only the item name and provide no way for transmitting context information. Therefore, if an item is to have different attributes in different contexts, System Dictionary must contain a separate item name and definition for each different set of attributes. If the data in System Dictionary is structured so as to support Hewlett-Packard's information management software (such as BRW), it is recommended that dual dictionaries (or domains) be maintained—one to support Information Management applications and the other to support Transact applications. In the System Dictionary that supports Transact applications, data items can then be redefined as often as necessary.

When defining data items which are extracted from System Dictionary or Dictionary/V for use in a Transact application, you should note that Transact only supports data item names that are up to 16 characters long.

If a data dictionary is being used, the Transact compiler looks for any undefined data items in the appropriate data dictionary. If it cannot find the data items in the data dictionary, it issues a warning message.

When the Transact/V processor interprets the p-code, it, too, looks in the Dictionary/V data dictionary for undefined data items, including those which could not be resolved from a System Dictionary data dictionary. These data items can be those not satisfied during compilation or data items defined to be satisfied at run time by a DEFINE(ITEM) *item-name* * statement. If the processor cannot find the data items in the data dictionary, it issues an error message and terminates processing.

Transact/iX programs do not look in the Dictionary/V data dictionary at run time. Any items that are not resolved at compile time for Transact/iX will generate a run-time error when they are used.

At compile time, all data item attributes can be resolved from their data dictionary definitions. At run time, the Transact/V processor can only resolve such basic data item attributes as type, size, decimal length, and storage length. However, it does not get such secondary attributes as heading or entry text and edit masks.

Transact can resolve VPLUS forms file and form definitions and data set and file layout definitions only at compile time.

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 can be composed of three child items: month, day, and year, in any order you choose. A child item itself can be a parent item, and it can contain child items. In this case, it would be both a child item and a parent item.

You define the relationship of a child to its parent by including, in the child item's definition, the parent item's 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 second character position of the parent item. Refer to the DEFINE(ITEM) description in Chapter 8 for details about defining parent and child items.

Only the parent item name can be added to the list register; the child item names cannot. 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. Transact understands that these data item names are part of the parent item, and transfers the data accordingly. Transact makes the connection between parent and child items through the DEFINE(ITEM) or a data dictionary definition of their relation. This parent/child relationship can be resolved from a data dictionary only at compile time, not at run time. The child items can be the elements of an array, which is the parent item.

Using the LOCK Option with the Database Access Verbs

The LOCK option applies to all database access verbs, which include DELETE, FIND, GET, OUTPUT, PUT, REPLACE, and UPDATE. The LOCK option can be used to override the SET(OPTION) NOLOCK statement for any specific verb. Tables 6-2 and 6-3 show how locking is applied with the possible combinations of locking methods for database and MPE and KSAM files, respectively. See the description of the individual verbs in Chapter 8 for more information. There is also a LOCK option that applies to the LOGTRAN verb, which is discussed in the next subsection.

When using the PERFORM= option with an iterative database verb such as FIND, adding the LOCK option will help to ensure data consistency when there are concurrent processes modifying, adding, or deleting database records. For more information about database integrity and the PERFORM= option, see "Suppression of Optimization" under the FIND statement in Chapter 8.

Table 6-2 Understanding Database Locking

Automatic Locking Combined With:	Transact Verbs						
	FIND	OUTPUT	GET	PUT	DELETE	UPDATE	REPLACE
No options	A	A	A	B*	C*	B*	C*
LOCK option	B	B	B	B	B	B	B
LOCK option and SET(OPTION) NOLOCK	B	B	B	B	B	B	B
SET(OPTION) NOLOCK only	A	A	A	A	A	A	A

A = No locks
B = Lock for the entire verb
C = Lock and unlock for each record retrieved
* = Lock if database opened with mode 1; otherwise no locks

Table 6-3 Understanding KSAM and MPE File Locking

Automatic Locking Combined With:	Transact Verbs						
	FIND	OUTPUT	GET	PUT	DELETE*	UPDATE	REPLACE
No options	C	C	C	C	C	C	C
LOCK option	B	B	B	B	B	B	B
LOCK option and SET(OPTION) NOLOCK	B	B	B	B	B	B	B
SET(OPTION) NOLOCK only	A	A	A	A	A	A	A
SET(OPTION) NOLOCK and LOCK option on SYSTEM statement	A	A	A	A	A	A	A

A = No locks
B = Lock for the entire verb
C = If lock is specified in SYSTEM statement, lock and unlock for each record retrieved
* = Delete not allowed on an MPE file

Using the LOCK Option with the LOGTRAN Statement

Locking across a transaction can be handled by transaction-level locking executed when you specify the LOCK option on the LOGTRAN statement. Transaction locking can be used with or without database logging. The syntax is:

`LOGTRAN(BEGIN)` *base,log-message*`[,`*option-list*`];`

where *option-list* includes the LOCK option in the following format:

`LOCK(`*setname*`[(`*cond*`)][,`*setname*`[(`*cond*`)]]...)`

You specify *setname* as a list of data set names separated by commas or as a @ sign to specify that the entire database (such as the *base* specified in the SYSTEM statement) is locked. You can also specify a lock condition parameter, *cond*, which can be `COND` or `UNCOND`, representing conditional or unconditional locking, respectively. The default is conditional locking. The data sets specified are locked at the set level when Transact encounters the LOGTRAN(BEGIN) or LOGTRAN(XBEGIN) statements. The data sets are unlocked when Transact encounters a corresponding LOGTRAN(END), LOGTRAN(XEND), or LOGTRAN(XUNDO) statement with the same database name (same database access path).

When using the LOCK option on the LOGTRAN statement, you should also specify the SET(OPTION) NOLOCK statement to ensure that automatic locking is not activated for any database access verbs within your transaction. The SET(OPTION) NOLOCK statement does not affect transaction locking. To re-activate automatic locking, use the RESET(OPTION) LOCK statement. In the example shown here, transaction level locking is used to lock two data sets in two different databases. Transaction locking is also used in the second version of the subsequent example.

```
SYSTEM LOCKS,BASE=BASE1(";"),BASE2(";");

DEFINE(ITEM) X1 X(10):
             X2 X(10);

SET(OPTION) NOLOCK;

PROMPT X1:X2;

LOGTRAN(BEGIN) $HOME," Lock Base1 Set ",LOCK(Base1master);
LOGTRAN(BEGIN) BASE2," Lock Base2 Set ",LOCK(Base2master);

PUT Base1master,LIST=(X1);
PUT Base2master,LIST=(X2);

LOGTRAN(END) BASE2," Unlock Base2 Set ";
LOGTRAN(END) $HOME," Unlock Base1 Set ";

EXIT;
```

7

Error Handling

Transact has a significant amount of error processing built into the run-time environment. This chapter explains the error handling process and the effect of the STATUS option on various verbs, especially when errors are detected. The topics covered are:

- Automatic error handling
- Using the STATUS option
- Compiler error messages
- Processor error messages
- Using EXPLAIN
- Trap handling

Automatic Error Handling

Transact automatically traps various types of errors encountered during the execution of a program and takes certain predetermined actions. Transact traps errors during data entry, during database or file operations, and during arithmetic calculations in LET expressions.

Data Entry Errors

Transact validates a value entered as a response to a data entry prompt. This is done according to attributes defined for the data item in a data dictionary or the Transact program—that is, data type, field size, decimal field length, integer field length. If it detects an error during validation, it issues an appropriate error message on the terminal and reissues the data entry prompt.

Database or File Operation Errors

Transact assumes that a data set or file error was caused by an incorrect user input—for example, by the user specifying an incorrect value for a key item. (Other types of software error conditions should be eliminated before the program is put into production mode.) If Transact detects an error, 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 to allow the user to start over with a command. The return location can be to a data entry prompt too. For instance, if an error occurs on the second of two database or file operation verbs and there is a data entry prompt between the two, the return location is the prompt statement immediately following the first database 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 choose to use automatic error handling, do not include statements between the prompts and file or database access verbs which may alter the data used in the operation. This is important because automatic error handling re-executes all statements between where the error occurred (such as the file or database operation) and where the data was collected (such as the PROMPT verb). Ignoring this caution may give you unanticipated results. For example:

```
(1)  LET (COUNTER) = 0;

(2)  PROMPT DATA-ITEM; PROMPT(KEY) DATA-ITEM;

(3)  LET (COUNTER) = (COUNTER) + 1;

(4)  FIND (DATA-ITEM); FIND DATASET1;

(5)  PUT DATASET2, LIST=(DATA-ITEM,COUNTER);
```

Example

This example shows what is displayed when you enter `EXPLAIN TVB 1070` at the command line:

```
---------------------------------------------------------
DATABASE BUFFER NOT ON WORD BOUNDARY (1070)

     The data buffer for a database operation must start on a word
     boundary. If necessary, insert a one-character fill item before
     the first data item of the database list or use the ALIGN option
     of the LIST verb.

MSG GROUP:    Transact/V
MSG CATALOG:  RAPIDCAT.PUB.SYS
MSG KEY:      TVB 1070
---------------------------------------------------------
```

For more examples of using EXPLAIN, see the *MPE V Commands Reference* manual.

Trap Handling

Traps occur when Transact encounters specific arithmetic errors, Pascal library errors, or a control-y is issued. When an arithmetic trap is discovered in a Transact program, an error is issued and processing stops. Transact does not always process traps in the same way depending on whether they occur in the main program, a called subprogram, an Inform system, or an ACI subprogram. The different types of processing for Transact trap handling are described below.

Trap handling can also be modified by specifically calling the trap intrinsics via the PROC verb.

System Initialization

Arithmetic traps are armed and enabled during initialization of a Transact program. ARITRAP and XARITRAP are called to enable and arm the arithmetic traps. Control-y traps are also enabled during initialization by issuing the XCONTRAP intrinsic. For Transact/iX, the Pascal library traps are set by calling XLIBTRAP.

PROC

When calling a subprogram (other than a Transact subprogram) using the PROC call, you can select the TRAP or NOTRAP option. If no option is selected, the default TRAP option is used. When the TRAP option is used, any arithmetic or library traps that occur in the subprogram result in an error message. The subprogram terminates, and control is returned to the main program. If NOTRAP is specified in the PROC statement, all arithmetic traps are ignored in the subprogram, and processing continues as if no problems occurred. No status or information is passed back to the main program with either option. Online and batch modes are processed in the same way.

CALL

When calling a Transact subprogram, arithmetic traps and control-y are not disabled. Trap handling in the main program is the same throughout the Transact subprogram. In batch mode, the called subprogram issues an error message and terminates. Control is returned to the main program and processing continues.

When calling an Inform or Report program, control-y is first disabled then re-enabled after the Inform or Report program completes. Arithmetic traps are not enabled or disabled. The arithmetic trap that is set at the time of the call is used throughout the Inform or Report program.

ACI

When the Architected Call Interface (ACI) intrinsic TL_CALL_TRANSACT is used to call a Transact subprogram from another language, the main program's arithmetic trap handling is saved, then re-enabled before returning to the main program. The Transact subprogram enables arithmetic and library traps during initialization. HPENBLTRAP is called to enable arithmetic traps, then XARITRAP is called to arm them. XLIBTRAP is called to arm the library traps. For more information about ACI see Appendix D.

LET

The LET verb has two modes for handling errors. The default mode is automatic, and occurs when the error message is displayed and processing continues to the next statement. The other mode, error branching, occurs when an error branch is specified in the LET statement. When certain errors are encountered, control is transferred to the error branch, and no error message is displayed. When the default mode is used in batch mode, an error on the LET verb will cause the program to terminate. For more information see the LET verb in Chapter 8.

IF, REPEAT, WHILE

When an arithmetic error occurs in the IF, REPEAT, or WHILE statements, an error message is displayed, and processing continues to the next statement. In batch mode, the error causes the program to stop. For more information see the IF, REPEAT, or WHILE statements in Chapter 8.

8

Transact Verbs

This chapter 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 listed, 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. 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 are either included within the syntax descriptions, or they follow the entire verb description.

CALL

Transfers execution to another Transact program or to a Report/V or Inform/V program.

Syntax

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

CALL passes control to another Transact program or to a Report/V or Inform/V 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. The calling program then resumes execution of the statement following the CALL statement.

When a CALL from a main program is executed, any open files or data sets remain open across the call. However, when the called program is an Inform/V or Report/V program, the database passwords must be specified again. The passwords can be specified programmatically from the terminal or in the stream file.

When a CALL from a called system is executed, files opened by the system that made the call do *not* remain open for use by the system it calls.

While a called Transact program is executing, both the calling program and the called program are in the memory stack and share the data register. Called Inform/V or Report/V programs do not share the same memory stack or data register.

If a called Report/V or Inform/V program uses any database or data file named in the SYSTEM statement of the calling Transact program, that database or file must be opened in a non-exclusive mode. Furthermore, the open mode must be compatible with the open mode used by Report/V or Inform/V (default mode 5), or the open mode used by Report/V or Inform/V must be altered to be compatible with the mode used by Transact. Any database locks should be released before the CALL statement.

The Transact/iX compiler can generate code for two different types of calls, referred to as "static" and "dynamic" calls.

Static calls are direct procedure calls to the called program. Static calls must meet the following requirements:

- The name of the called program must be available at load time.
- Either the object code for the called program must be in an RL or in an RSOM file at link time or the executable code for the called program must be in an XL at load time.
- a literal program name must be used in the CALL statement.
- The DYNAMIC_CALLS option must be off.

Dynamic calls use the MPE/iX HPGETPROCPLABEL intrinsic to load the called program at run time. Dynamic calls must meet the following requirements:

- The object code for the called program must be in an XL. However, only those programs that are actually called at run time need to be present in the XL.
- Either a variable program name must be used in the CALL statement or the DYNAMIC_CALLS option must be on.

The name of the called program does not need to be available until run time.

There are advantages and disadvantages to both types of call. The primary advantage of static calls over dynamic calls is superior run-time performance. Dynamic calls must use HPGETPROCPLABEL whenever a CALL statement is executed, and this intrinsic must search the various libraries and load the requested program. With static calls, the called programs are loaded when the main program is loaded and the run-time overhead is negligible for most applications.

However, dynamic calls have the advantage that the name does not need to be known at compile time. Therefore, CALL statements that use a variable for the called program name are always compiled as dynamic calls.

A further advantage is that dynamic calls do not require the object code for the called programs to be available until the CALL statement is actually executed. Therefore, dynamic calls allow a main program to be executed even if some of the called programs it references have not yet been compiled (or even written), as long as the main program does not attempt to actually call any of the missing programs.

Statement Parts

file-name The name of one of the following:

- Another Transact program (as specified in a SYSTEM statement).
- A Report/V program (as specified in a REPORT statement).
- An Inform/V program (as specified in the report name of the catalog).

If *file-name* names an Inform/V file or Report/V file, the "Report" or "Inform" option must be specified in the *option-list*. *file-name* can also be specified as (*item-name*[(*subscript*)]), where *item-name* is the name of an item that contains the name of the program or report to be executed. A *subscript* is allowed if the referenced item is an array. (See "Array Subscripting" in Chapter 3.)

file-name can be fully qualified as *file-name.group.account*

If (*item-name*[(*subscript*)]) is specified, the call is generated dynamically at run time. If *file-name* is specified, then the call can be either static or dynamic, depending on the compile options specified. (See the discussion of compiler options in Chapter 9.)

password A password for access to the database used by the called program. This parameter is optional, required only if the called program does not specify a database password in its SYSTEM statement or if the database is not already opened by the called program. Transact prompts for a password at run time if it is not specified here. If the password is in both places, the password specified in the SYSTEM statement of the called program takes precedence.

password can be specified as:

"*text-string*"	The database password.
item-name [(*subscript*)]	The name of an item containing the database password. A *subscript* is allowed if the item being referenced is an array item.

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 database specified in the SYSTEM statement, the second password on the list is used to open the second database specified, and so on.

If only one password is passed, it opens the first database specified in the SYSTEM statement with that password as well as subsequent specified databases that have no password.

mode — The mode in which the database used by the called program is to be opened. This parameter is optional, and can be specified here if the SYSTEM statement in the called program does not specify it; if mode is specified both places, the mode specified in the called program takes precedence. *Mode* can be specified as:

- *digit* — Number 1 to 8. Default=1. A digit is only valid when calling another Transact program.
- *item-name* [(*subscript*)] — Name of item containing *mode* value. A *subscript* is allowed if the item being referenced is an array item.

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

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

- DATA=*item-name* [(*subscript*)] — The location in the data register of the calling program where a called Transact program can 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. A *subscript* is allowed if the item being referenced is an array item. (See "Array Subscripting" in Chapter 3.) Although the contents of the data register can be passed via a CALL statement, the list register contents are not. Therefore, the called program must set up its own list register before execution.

  If no DATA= is specified, the called system will start overlaying the calling program's data register with its own list/data registers. The item must start on a 16-bit boundary.

- SIZE=*number* — The number of 16-bit words of data register space that a called Transact 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 16-bit words in the data register and must start on a 16-bit word boundary.

Note

When Transact CALLs a Transact 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 maximum 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/V report specified by *file-name*. None of the Inform/V menus are displayed. If needed, a database password is prompted for. After the Inform/V report is complete, control returns to the statement following the call.

REPORT

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

STATUS

When the STATUS option is used, the success of a CALLed Transact program is described by the value in the 32-bit status register. After a called program completes, Transact sets the calling program's 32-bit status register to one of the values in the table below.

Status Register Value	Meaning
0	No errors were detected by Transact within the called program.
-2	An error was detected by Transact within the called program.

A 0 will be returned in the status register in cases where the error is handled by the programmer or end user. A 0 will be returned in the following cases:

- Data errors or command errors for interactive programs occur.
- Error messages are suppressed by the subprogram using the STATUS option or the NOMSG option.

- Error messages are suppressed by the ERROR= option on the LET verb.

When the STATUS option is not used on the CALL verb, Transact does not alter the calling program's status register.

The STATUS option can be used only with called Transact programs. The Transact compiler returns an `INVALID OPTION` error message when used with called Report/V and Inform/V programs.

Limitations on the CALL Statement

The following limitations apply to the CALL statement when you use the Transact/iX compiler:

- Calls from a Transact/iX program can only be made to Transact programs that have been compiled with the Transact/iX compiler. The called program must be linked to the calling program in one of the ways described above.
- The SWAP option is not supported by Transact/iX and is ignored if it appears on a CALL statement. Since MPE/iX systems have far more data space than MPE V systems, this option is not needed.

The Transact/iX compiler issues an informational message if the SWAP option is encountered:

```
*INFO: THE 'SWAP' OPTION FOR THE CALL VERB IS NOT NECESSARY ON AN MPE/iX SYSTEM
```

Floating Point Format

When passing parameters or data that access real numbers, the called program must be compiled with the same real-number format as the main program.

Examples

The first example calls the INVMGT program, provides a password for opening any databases used by INVMGT, and allows the database to be opened in mode 7 for exclusive read access. INVMGT can use data register space beginning at the item named ORDER, and it can use 1000 16-bit words of space.

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

In the next example, the user is prompted for the name of the application to run. Then the password needed to access the database is retrieved from the PASSWORD-DSET detail set.

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

```
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 the programs MAIN and CALC. MAIN uses a CALL verb with and without the STATUS option. The status register is initialized to the value "111" and tested after each CALL verb for the expected status register value. Without a STATUS option on the CALL verb, MAIN's status register will not be changed. When a STATUS option is used, the status register will be set to -2 because of the arithmetic error in the caller program CALC.

In the called program CALC, the arithmetic operation fails. Although the LET verb results in an error, the status register for CALC is unchanged.

```
SYSTEM MAIN;
DEFINE(ITEM)  ZEROS        I(5,,2):
              PSTATUS      I(5,,4);
LIST ZEROS,INIT:PSTATUS,INIT;
LET STATUS = 111;

<<Example of CALL verb without the STATUS option.>>
CALL CALC, DATA=ZEROS;
IF STATUS = 111 then display "MAIN's STATUS IS STILL 111.";

<<Example of CALL verb with the STATUS option.>>
CALL CALC, DATA=ZEROS,STATUS;
IF STATUS = -2 THEN DISPLAY "MAIN's STATUS IS NOW -2.";

EXIT;
END;
```

```
 SYSTEM CALC;
 DEFINE(ITEM)  ZEROS          I(5,,2):
               PSTATUS        I(5,,4);
 LIST ZEROS:PSTATUS;
 LET STATUS = 222;
 LET (ZEROS) = (ZEROS) / (ZEROS);
 <<Causes an arithmetic error.>>
 LET (PSTATUS) = STATUS;
 DISPLAY "CALC's STATUS REGISTER AFTER ERROR>>",LINE=2:PSTATUS,NOHEAD;
 EXIT;
 END;
************************* RESULTS ***************************
ERROR: INTEGER DIVIDE BY ZERO  (PROG 54,6) [CALC]

CALC's STATUS REGISTER AFTER ERROR>>  222
MAIN's STATUS IS STILL 111.

*ERROR: INTEGER DIVIDED BY ZERO  (PROG 54,6) [CALC]

CALC's STATUS REGISTER AFTER ERROR>>  222
MAIN's STATUS IS NOW -2.
```

This example shows modifications to the programs MAIN and CALC. The program MAIN uses the CALL verb with and without the STATUS option. The status register is still set to the value "111" and tested after each CALL verb for the expected status. Without a STATUS option on the CALL verb, MAIN's status register will not be changed. When a STATUS option is used, MAIN's status register will be set to 0 because the error in the called program is handled by the called program.

In the called program CALC, we still perform an arithmetic operation which causes an error. The ERROR= option which has been added causes the status register to be set to a value of 3 when the arithmetic operation fails. The status register is not shared between programs, but the program MAIN displays or checks the value stored in shared item PSTATUS.

```
SYSTEM MAIN;
DEFINE(ITEM)  ZEROS         I(5,,2):
              PSTATUS       I(5,,4);
LIST ZEROS,INIT:PSTATUS,INIT;
LET STATUS = 111;

<<Example of CALL verb without the STATUS option.>>
CALL CALC, DATA=ZEROS;
IF STATUS = 111 then display "MAIN's STATUS IS STILL 111.";

<<Example of CALL verb with the STATUS option.>>
CALL CALC, DATA=ZEROS,STATUS;
IF STATUS = 0 THEN DISPLAY "MAIN's STATUS IS NOW 0.";
DISPLAY "PSTATUS has a value of: ":PSTATUS,NOHEAD;

EXIT;
END;

SYSTEM CALC;
```

```
 DEFINE(ITEM)  ZEROS         I(5,,2):
               PSTATUS       I(5,,4);
 LIST ZEROS:PSTATUS;
 LET (ZEROS) = (ZEROS) / (ZEROS), ERROR= NEXT-LINE(*);
 <<Causes an error>>
NEXT-LINE;
 LET (PSTATUS) = STATUS;
 <<See LET verb for table of STATUS values. >>
 DISPLAY "CALC's STATUS REGISTER AFTER ERROR>> ",LINE=2:PSTATUS,NOHEAD;
 EXIT;
 END;
************************* RESULTS *****************************
CALC's STATUS REGISTER AFTER ERROR>>  3
MAIN's STATUS IS STILL 111.

CALC's STATUS REGISTER AFTER ERROR>>  3
MAIN's STATUS IS NOW 0.
PSTATUS has a value of: 3

END OF PROGRAM
```

CLOSE

Closes an MPE or KSAM file, a data set or database, or a VPLUS forms file.

Syntax

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

CLOSE closes and rewinds an MPE or KSAM file or a data set, or closes the entire database. 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 database access verb that performs serial or reverse serial access. These verbs are FIND, GET, DELETE, and OUTPUT which have (SERIAL) and (RSERIAL) modifiers. You would also use CLOSE before a FILE(SORT) statement.

The CLOSE statement has the following special forms:

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.
CLOSE $VPLS	Closes the terminal block mode and the active VPLUS forms file, releasing the memory space used by VPLUS in the DB-DL stack area (Transact/V only). This relieves the contention for DB-DL stack memory between VPLUS and other subsystems such as DSG/3000. Do not use between a SET(FORM) verb and another forms verb.

For a discussion about using CLOSE during dynamic transactions, see the "Database Close" section in Chapter 6.

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 can close an entire database by specifying *file-name* as a database 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*]) Suppresses 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. The *item-name* must be a parent.

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 Chapter 7.

NOMSG Suppresses the standard error message produced as a result of a file or database error.

STATUS Suppresses the action defined in Chapter 7 under "Automatic Error Handling." You may have to add status checking to your code if you use this option.

When STATUS is specified, the effect of a CLOSE statement is described by the 32-bit integer 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 the database or MPE/KSAM file system error documentation that corresponds to the value.

See "Using the STATUS Option" in Chapter 7.

Examples

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;
```

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

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

DATA

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

Syntax

```
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	Places value in data register. (See Syntax Option 1.)
ITEM	Prompts for item name and if found, places value in data register. (See Syntax Option 2.)
KEY	Places value in argument register. (See Syntax Option 3.)
MATCH	Places value in data register. Sets up match criteria in match register. (See Syntax Option 4.)
PATH	Places value in data register and in argument register. (See Syntax Option 5.)
SET	Places value in data register unless user presses carriage return. (See Syntax Option 6.)
UPDATE	Places value in data register. Places 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, Transact 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 Transact detects an error, then it displays an appropriate error message and reissues the prompt.

With native language support, Transact validates numeric data using the thousands and decimal indicators of the language in effect. (See Appendix E, "Native Language Support" for more information.)

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. The Syntax Options subsection below describes the impact of each modifier in detail.
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 more recent SET or RESET command.
prmpt-string	The string that prompts the 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 data dictionary, if one exists.
optn-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	Does not suppress leading blanks supplied in the input value; leading and trailing blanks are normally stripped.
CHECK=*set-name*	Checks input value against the master set *set-name* to ensure that the value already exists. If the condition is not met at execution time, Transact displays an appropriate error message and reissues the prompt. You cannot use this option with a KSAM or MPE file, in a DATA(MATCH) statement, nor with child items.
CHECKNOT=*set-name*	Checks input value against the master set *set-name* to ensure that the value does not already exist. If the option condition is not met at execution time, then Transact issues an appropriate error message and reissues the prompt. You cannot use this option with a KSAM or MPE file, in a DATA(MATCH) statement, nor with child items.
NOECHO	Does not echo the input value to the terminal.
NULL	Fills item with ASCII null characters (binary zeros) instead of blanks.
RIGHT	Right-justifies the input value within the register field.

STATUS — Suppresses normal processing of "]" and "]]", which cause an escape to a higher processing or command level.

Status Register Value	Meaning
−1	User entered a "]".
−2	User entered a "]]".
−3	User entered one or more blanks and no non-blank characters.
−4	If timeout is enabled with a FILE(CONTROL) statement, a timeout has occurred.
> 0	Number of characters (includes leading blanks if BLANKS option is specified); no trailing blanks are counted.

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[(subscript)]}[("prmpt-string")"][,optn-list][:item-name ... ] ... ;
         {          *           }
```

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. *item-name* can be modified with *subscript* if the referenced item is an array item. (See "Array Subscripting" in Chapter 3.)

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

The ITEM modifier is typically used to update or correct one or more values in the data register. DATA(ITEM) issues a prompt *prompt-string* to request an item name. When the user enters an item name in response to this prompt, Transact looks for this item in the list register. If the item name cannot be found, it displays an error message and reissues 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 "]", Transact reissues the *prompt-string* prompt. If the user responds with "]]", Transact returns to command mode.

If you use the REPEAT option, then the operation is repeated until a termination character (] or]]) 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 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 in the key register.

(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. You cannot specify either CHECK= or CHECKNOT= with DATA(MATCH). It places the value in the data register in an area associated with the current data item if the "*" is used or in an area associated with a named data item if an item name is specified. The item name and value are also placed in the match register as a selection criterion for subsequent database or file operations.

If the item name is an unsubscripted array, only the value of the first element of the array will be set in the data register. This value from the data register will be set up as match criterion in the match register.

You should note that when a single key value is entered for the match, Transact performs a chained read on the data set if the item is a search or key item. However, if a range of values or non-key value is specified, a serial read is performed.

User responses to the DATA(MATCH) prompt are further explained in the discussion of "Match Register" in Chapter 4. (See also "MATCH Specification Characters" and "Responding to a MATCH Prompt" in Chapter 5.) The MATCH modifier 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 options, which specify that a match selection is to be performed on a basis other than equality.

If you specify one of the options listed below, the entire user input is treated as a single value. The match specification characters described in Chapter 5 are not allowed as user input with the options listed below.

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

For example, if the program contains the data statement

```
DATA(MATCH) CUSTNO,GE;
```

and if the user responds to the prompt by entering 079333, then only customer numbers greater than or equal to 079333 will be selected.

(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. The value 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 data sets. 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.

(6) DATA(SET) {*item-name*[(*subscript*)]}[("*prompt-string*")][,*option-list*]
{ * }

[:*item-name* ...] ... ;

The primary use of the SET modifier is to update values in the data register for existing items in the list register. 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. *item-name* may be modified with (*subscripts*) if the referenced item is an array item. (See "Array Subscripting" in Chapter 3.)

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 user respond with a carriage return to the reissued prompt. Both CHECK= and CHECKNOT= look for the item in the master set even if the user enters a carriage return.

A special option, SHOW, is available only with the (SET) modifier. SHOW causes the old value to appear in the prompt for a new value. This allows the user to see what the item will contain if a carriage return is entered. The values are displayed left justified, with trailing blanks suppressed. One blank is displayed when an alphanumeric item is all blank. The SHOW option can only be used in the DATA(SET) statement. The following example uses the SHOW option:

```
DEFINE(ITEM) PRODUCT X(40):
             QUANTITY I(3);
LIST PRODUCT,INIT:
     QUANTITY,INIT;
DATA(SET) PRODUCT,SHOW:
          QUANTITY,SHOW;
```

This example causes the following prompts to be displayed the first time data is entered:

```
PRODUCT(= )>
QUANTITY(=0)>
```

If the values "grapefruit" and "10" are entered, the prompts appear like this when displayed again:

```
PRODUCT(=grapefruit)>
QUANTITY(=10)>
```

If an alphanumeric string is longer than 30 characters, the first 30 characters are displayed:

```
PRODUCT(=mason valley delightful grapef...)>
```

The trailing periods (...) indicate that the value is too long.

(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

This example asks the user for an account number, which is placed in the argument register for subsequent access to the ACCOUNT-MASTER set. 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 reissued.

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

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

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.

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

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 set to blanks. If you want Return to leave the existing data, you must use a DATA(SET) statement.

DEFINE

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

Syntax

`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 are:

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

The *definition-list* depends on the modifier, or 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 V system intrinsics that are called by the PROC verb. Declaring the intrinsic in this manner causes Transact to load the intrinsic at system startup.

If you include an intrinsic name that is not recognized by the compiler, a compile time error message will be issued. If this occurs, remove the unrecognized intrinsic from the DEFINE(INTRINSIC) statement. If the DEFINE(INTRINSIC) statement is removed, Transact tries to load the intrinsic when the intrinsic is called with a PROC statement. Intrinsics specified with the DEFINE(INTRINSIC) statement are resolved at system startup from SL.PUB.SYS.

(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*`"]`
`[,INIT=[`*value*`|(BINARY(`*value*`))|(HEX(`*value*`))|(OCTAL(`*value*`))]]`
`[,OPT]`
`[:`*item-name* `... ] ... ;`

This option defines an *item-name* not defined in the data dictionary. It can also be used to redefine items already defined in the data dictionary. Any number of item-names, separated by colons (:) can be specified in a single DEFINE(ITEM) statement. (See Chapter 3, "Data Items," for detailed descriptions of data types.)

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 database 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 data 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. The *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 zeros 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. (See Chapter 3, "Data Items," for detailed descriptions of data types.)

Note

Transact's "E" item type is different from the TurboIMAGE "E" item type that is defined as IEEE real.

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, K, 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. (See Table 3-3 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. For Z and P types only, 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

The name of the 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 equivalence in SPL or 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);
```

When working with KSAM or MPE files, it is useful to define the record as a parent item and the fields as child items. (See the example in the description of the SYSTEM verb.)

position

The byte position in the parent item that is the starting position of the child item. Begin counting at position 1. The default is 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:
             MONTH X(2)=DATE(3):
             DAY X(2)=DATE(5);
```

ALIAS=(*alias-reference*)

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 storage length as the data item value referenced in the data set or file.

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

COMPUTE= *arithmetic-expression*
: An arithmetic expression that specifies the computation to be performed before the item is used in a DISPLAY, OUTPUT, or LET statement. It may contain two or more variables separated by one or more arithmetic operators. Use the form shown for 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 a description of the edit mask feature.) When a numeric value to be printed is too large for the edit mask, a series of pound signs (#) are printed in place of the value, to indicate an overflow.

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.

INIT=[*value*]
: Initial value moved into the item each time it is added to the list register. The INIT parameter on the LIST verb overrides this parameter. If this parameter appears without a value, the item is initialized to zero for numeric or blank for ASCII, eliminating the need to use the INIT parameter with the LIST verb. For example:

```
DEFINE(ITEM) CODE I(3), INIT=999;
DEFINE(ITEM) QUANTITY I(3), INIT=;
```

The INIT= option works similarly to the LET verb. If an array is being initialized, each element in the array is initialized to *value*.

This option also allows initialization of I, J, and K types in terms of a binary, octal, or hexadecimal base. The number specified is treated as a signed, 32-bit number. Enough storage must be allocated to hold the specified number.

The following examples illustrate the use of this option.

The first example defines an initial value of -1. Two bytes of storage are sufficient.

```
DEFINE(ITEM) OCT1 I(5,,2), INIT=(OCTAL(37777777777));
```

The second example defines an initial value of -2. Two bytes of storage are sufficient.

```
DEFINE(ITEM) OCT2 I(5,,2), INIT=(OCTAL(37777777776));
```

The third example defines an initial value of 65535. Two bytes of storage are not sufficient so four bytes must be allocated.

```
DEFINE(ITEM) HEX1 I(5,,4), INIT=(HEX(ffff));
```

The fourth example defines an initial value of -32768. Two bytes of storage are not sufficient so four bytes must be allocated.

```
DEFINE(ITEM) HEX2 I(5,,4), INIT=(HEX(ffff8000));
```

The last example defines an initial value of 2147483647, the maximum possible using a binary, octal, or hexadecimal base. Eight bytes of storage are required.

```
DEFINE(ITEM) HEX3 I(10,,8), INIT=(HEX(7fffffff));
```

The INIT= option cannot be used for child items.

Note Initializing a positive type with a negative value results in a run-time error.

OPT

OPT is used in combination with the compiler control option, OPT@, OPTE, OPTH, OPTI, and OPTP. When OPT is specified for an item, the compiler does not store the item's textual name in the p-code file if the OPTI control option has been specified. OPT, used in conjunction with the above compiler control options, saves data segment stack space at execution time. (See Chapter 9 for a discussion of the OPT@, OPTE, OPTH, OPTI, and OPTP compiler options.)

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 a database.

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

This option defines a synonym for an item defined elsewhere in the program or in the dictionary. Other item attributes may not be defined using this syntax option.

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 Transact 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 MARKER1 and MARKER2. The second statement assigns space in the list register to MARKER1. 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 MARKER1 and MARKER2. (This example assumes that the current entry has been set up appropriately by a previous get of the EMPLOYEES data set.)

You might 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 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.

Note This format is not valid in Transact/iX.

Examples

The following example shows how to define a key item (called KEY-NUM) 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):
                KEY-NUM    X(10)= RECORD(3);

MOVE (KEY-NUM) = "A123456789";      <<Assign value to key           >>
SET(KEY) LIST(KEY-NUM);             <<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-NUM    X(10) = RECORD(3):
                GEN-KEY    X(2)  = RECORD(3);
```

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

DELETE

Deletes KSAM files or data set entries. DELETE cannot be used with MPE files.

Syntax

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

DELETE specifies the deletion of one or more KSAM file entries or data set 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.

If you are performing dynamic transactions (Transact/iX only), be aware that transactions have a length limit. For a discussion about how DELETE is affected by this limitation, see "Limitations" under "Dynamic Roll-back" in Chapter 6.

Note After the first retrieval, Transact uses an asterisk (*) for the call list to optimize subsequent retrievals of that data set.

Statement Parts

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

none — Deletes an entry from a master set based on the key value in the argument register; this option does not use the match register.

CHAIN — Deletes entries from a 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. If match criteria is used, all items specified in the match register must be included in a LIST= option.

CURRENT — Deletes the last entry that was accessed from the KSAM file or data set.

DIRECT — Deletes the entry stored at the specified record number in a KSAM file, a detail set, or a master set. Before using this modifier, you must store the record number as a 32-bit integer in the item specified by the RECNO= option.

PRIMARY — Deletes the 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.

Note DELETE(PRIMARY) deletes only one entry at the primary location, and the secondary entry, if any, automatically migrates to the primary location after the delete.

RCHAIN — Deletes entries from a 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 — Deletes entries from a data set in the same manner as the SERIAL option, except in reverse order. For a KSAM file, this operation is identical to SERIAL.

SERIAL — Deletes entries in serial mode from a KSAM file or from a 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 KSAM 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*])] — Suppresses 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. The *item-name* must be a parent.

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.

LIST=(*range-list*) — The list of items from the list register to be used for the DELETE operation. For data sets, no child items can be specified in the range list.

If the LIST= option is omitted with any modifier, all the items named in the list register are used.

When the LIST= option is used, only the items specified in a LIST= option have their match conditions applied when the items are included in the match register. When the LIST= option is omitted, items which appear in the list register and the match register have their match conditions applied. Otherwise, the match conditions for an item are ignored.

The match register can 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.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the most recent entry added to the list register. (See Chapter 4 for more information on registers.)

The LIST= option has a limit of 64 individually listed item names and a limit of 255 items specified by a range for a TurboIMAGE data set.

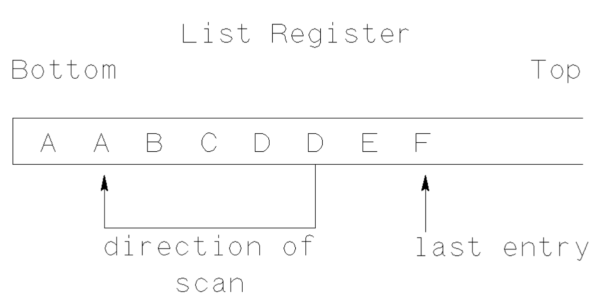
All item names specified must be parent items.

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

(*item-name*) A single data item.

(*item-nameX*: *item-nameY*) All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error

	is returned if duplicate entries are selected.
	If *item-nameX* and *item-nameY* are marker items (see the DEFINE(ITEM) verb) and if there are no data items between the two on the list register, no database access is performed.
(*item-nameX:*)	All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.
(:*item-nameY*)	All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.
(*item-nameX, item-nameY, ... item-nameZ*)	The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM files, data items must be specified in the order of their occurrence in the physical record. This order need not match the order of the data items on the list register. This option is less efficient to use than the options listed above.
(@)	Specifies a range of all data items of *file-name* as defined in the data dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.
(#)	Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the dictionary. This order need not match the order of the data items in the list register.
( )	A null data item list. That is, delete the entry or entries, but do not retrieve any data.

LOCK

Locks the specified file or database. If a data set is being accessed, the lock is set the whole time that DELETE executes. If LOCK is not specified but the database is opened in mode 1, which requires a lock, the lock specified by the type of automatic locking in effect is active while the entry is processed by any

	PERFORM= statements, but is unlocked briefly before the next entry is retrieved.
	For a KSAM file, if LOCK is not specified on DELETE but is specified for the file in the SYSTEM statement, then the file 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. (DELETE is not allowed on MPE files.)
	Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the DELETE verb.
	A database opened in mode 1 must be locked while DELETE executes. For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on DELETE if SET(OPTION) NOLOCK is specified. If a lock is not specified (for a database opened in mode 1) an error is returned.
	See "Database and File Locking" in Chapter 6 for more information.
NOCOUNT	Suppresses the message normally generated to indicate the number of deleted entries.
NOMATCH	Ignores any match criteria set up in the match register.
NOMSG	Suppresses the standard error message produced as a result of a file or database error.
PERFORM=*label*	Executes the code following the specified label for every entry retrieved by the DELETE verb before the DELETE operation. The entries can be optionally selected by match criteria.
	This option allows operations to be performed on retrieved entries without having to code loop-control logic. You can nest up to a maximum of ten PERFORM options.
RECNO=*item-name* [(*subscript*)]	With the DIRECT modifier, you must define *item-name* to contain the 32-bit integer number (I(9,,4)) of the record to be deleted.
	With other modifiers, Transact returns the record number of the deleted record in the 32-bit integer *item-name*.
	The *item-name* can be modified with *subscript* if the referenced item is an array item. (See "Array Subscripting" in Chapter 3.)
SINGLE	Deletes only the first selected entry.

SOPT	Suppresses the optimization of database calls. This option is primarily intended to support a database operation in a performed routine that is called recursively. The option allows a different path for the same detail set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of a call list of "*" after the first call is made. Use SOPT if you are calling TurboIMAGE through the PROC or CALL verbs. For an example of how SOPT is used, see "Examples" at the end of the FIND verb description. For more detailed information about SOPT, see "Suppression of Optimization versus WORKFILE" under the FIND verb in this chapter.
STATUS	Suppresses the actions defined in Chapter 7 under "Automatic Error Handling." You may want to add status checking to your code if you use this option. When STATUS is specified, the effect of a DELETE statement is described by the 32-bit integer 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 database 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) or DELETE(RSERIAL).
- The normal find of the chain head by the DELETE is suppressed, so PATH should be used before DELETE(CHAIN) or DELETE(RCHAIN).

See "Using the STATUS Option" in Chapter 7.

Examples

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 (database error 17—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;
```

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= option because DELETE reads each record in the chain into the data register area associated with DEBT-LEVEL in order to check the match criteria before deleting the entry.

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

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

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

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

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

DISPLAY

Produces a display of values from the data register.

Syntax

DISPLAY[([[TABLE],[FILE=*mpe-file*]])][*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 can be formatted by any active FORMAT verb.

Statement Parts

none or TABLE without *display-list*
: Transcact 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 is displayed before each line:
- 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 with *display-list*
: Headings are displayed only at the top of each new page in the information display.

mpe-file
: The name of an MPE file that will receive the output from the DISPLAY statement.

display-list
: The display list contains one or more display fields and their formatting parameters, as shown in the following format:

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

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

display-field The following options can be used for display fields:

- A reference to a data item name in the list register (the data item name can be subscripted if the item referenced is an array item).
- A child item name whose parent item is in the list register.
- A character string delimited by quotation marks.
- If no display field is specified, Transact defaults to a NULL (" ") character string.

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

Five system variables can also be used as display fields. As noted, some are affected by native language support. (See Appendix E, "Native Language Support," for more information.)

$CPU	Displays the cumulative amount of CPU time used by the Transact program, in milliseconds.
$DATELINE	Displays the current date and time in the form Tue, Apr 14, 1992, 3:07 P.M. The format is affected by native language support.
$PAGE	Displays the current page number.
$TIME	Displays the current time; the default format is HH:MM AA (for example, 03:07 PM). The format is affected by native language support.
$TODAY	Displays the current date; the default format is MM/DD/YY (for example, 04/14/92). The format is affected by native language support.

Note Text can be displayed only in columns 1 through 79. Column 80 is reserved for the carriage control character.

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

CCTL=*number* Issues 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*. Note that the use of

	CCTL=*number* and LINE, NOCRLF, or ROW, may affect output due to conflicting values. Valid range is 30–1025.
CENTER	Centers a display field on a line. The entire field, including leading or trailing blanks, is centered.
COL=*number*	Starts the display field in the absolute column position specified by *number*. The first column position is 1. Maximum is 299. If the display is already at a column position equal to or 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 meanings when used in the *edit-string* for all *display-field*s except system variables $TIME and $TODAY: ^ — Inserts the character from the source data field into this position in the display field. Z — Suppresses leading zeros. Note that you must use an uppercase Z. $ — Adds business (single character) currency symbol. If the language-defined currency symbol precedes, then the symbol is floated. If the symbol succeeds, then it follows the last character of the number and the edit mask is shifted left one character to leave room. If the symbol imbeds, it replaces the radix (decimal point or equivalent). If no business currency symbol is defined for the current language, then "$" edit characters are treated the same as "other" edit characters, explained below.

Note In Transact/iX native language mode, the pound sterling currency sign (£) does not float the way the dollar sign ($) does in a displayed field with the edit mask. To get the pound sign to float, change your terminal configuration to KEYBOARD=UK. When you specify the edit mask, use the dollar sign in place of the pound sign. The pound sign will then be displayed.

Note

The number of digits available for the source number depends on the type of currency symbol. Thus, the same value might cause a field overflow in some languages and not in others.

*	Fills field with asterisks.
.	Aligns the implied decimal point as specified in the dictionary or in a DEFINE(ITEM) definition statement with this edit character in the edit mask and outputs the language-defined radix character.
!	Ignores the implied decimal place and replaces this character with a language defined radix character.
,	Outputs the language-defined thousands separator character (numeric only).
(	Surrounds negative values with parentheses (must be last character in edit mask).

All "other" characters, which mean any character not defined above in the list of special characters, are treated as insert characters. For example:

```
EDIT="@@@@@@.@@"
```

displays entered data as:

```
@@@@@@.@@
```

To denote numeric data type 9, Z, P, I, J, K, R, or E negative values with a trailing "-", "CR", or "DR", add a trailing "-", "CR", or "DR" to the edit string. Some edit-string examples follow:

Number	Edit String	Result
1234	$$,$$$!^^	$12.34
123456	$$,$$$!^^	$1,234.56
123456	***,**$!^^	*$1,234.56
000009	ZZZZ!^^	.09
475.49	XXX,XXX.XX	XXX,XXX.XX
-123456	$$,$$$!^^CR	$1,234.56CR
-123456	Z,ZZZ!^^-	$1,234.56-
230485	^^/^^/^^	23/04/85

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 the hour with no leading blank or zero if hour $<$ 10.
ZH	Displays the hour with leading blank if hours $<$ 10.
HH	Displays the hour with leading zero if hours $<$ 10.
24	Displays the hour as expressed on a 24-hours clock; used as a prefix to H.
M	Displays the minute with no leading blank or zero if minute $<$ 10.
ZM	Displays the minute with leading blank if minute $<$ 10.
MM	Displays the minute with leading zero if minute $<$ 10.
S	Displays the second with no leading blank or zero if second $<$ 10.
ZS	Displays the second with leading blank if second $<$ 10.
SS	Displays the second with leading zero if second $<$ 10.
T	Displays the 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.
AA	Displays both letters in the AM or PM sequence in uppercase.
aa	Displays both letters 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 the day of the month with no leading blank or zero if day < 10.
ZD	Displays the day of the month with leading blank if day < 10.
DD	Displays the day of the month with leading zero if day of the month < 10.
DDD	Displays the Julian day of year.
M	Displays the month with no leading blank or zero if month < 10.
ZM	Displays the month with leading blank if month < 10.
MM	Displays the month with leading zero if month < 10.
*n*M	Displays the first *n* letters of month name in uppercase; if *n* > number of letters in month name, trailing blanks are not inserted.
*n*m	Displays the first *n* letters of month name in lowercase except for the first letter, which appears in uppercase.
YY	Displays the last two digits in current year.
YYYY	Displays the current year.
*n*W	Displays the first *n* letters of day of week in uppercase; if *n* > length of the week name, no trailing blanks are inserted.
*n*w	Displays the 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 April 14, 1992, make it appear as follows:

Edit Mask	Displayed Date
3w. 3m DD, YYYY	Tue. Apr 14, 1992
DD 3M, YY	14 APR, 92
M-DD-YY	4-14-92
MM/DD/YY	04/14/92
DDD, YYYY	105, 1992

Note

When a numeric value to be printed is too large for the edit mask, a series of pound signs (#) are printed in place of the value, to indicate an overflow.

HEAD="*character-string*"
: Uses the *character-string* 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*]
: Places 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. The default is 1; valid range is 0–299.

LEFT
: Left-justifies the data item value in the display field. This is the default specification.

LINE[=*number*]
: Starts the next 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 overprint and you want it to do so, you should specify LINE=0. [LINE=0 and LINE= cause a carriage return but no line feed.] The default is 1; maximum is 99.

LNG=*number*
: Truncates 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. Range is 1–299.

NEED=*number*
: Prints 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. If you are grouping a set of items together on a single line, the

	NEED= must appear with the first item. Range is 1–99.
NOCRLF	Does not issue a carriage return and line feed for the display line containing the display field. This parameter allows you to print output from the next DISPLAY statement on the same line where the previous display left off. NOCRLF is processed when a listing goes to the terminal or printer. If the listing is sent to a disk file, the option is ignored.
NOHEAD	Suppresses 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*]	Starts the display field on a new page or on a page after a page skip count specified by *number*. The default is 1; maximum is 99.
RIGHT	Right-justifies the data item value in the display field.
ROW=*number*	Places 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. Maximum is 99.
SPACE[=*number*]	Places 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 is 1; maximum is 299.
TITLE	Displays the associated display field and any preceding display fields only at the start of each new page for which this statement applies.
TRUNCATE	Truncates this display field if it overflows the end of the display line; if field is a numeric type, displays pound signs and does not truncate.
ZERO[E]S	Right-justifies a numeric data value in the display field and inserts leading zeros.

Redirecting Output To A File

The formatted output generated by DISPLAY can be redirected to a specified file by using the FILE= option. This feature allows you to generate multiple reports and to save each in a different file. The only requirement is that the specified file must first be identified by a corresponding SYSTEM statement using the FILE= option. If the file is not defined in the SYSTEM statement, an INVALID FILE NAME error will occur during compilation. The default output width for DISPLAY is 79 characters.

When using this option, the DISPLAY verb sets the status register to indicate the number of characters written to the specified file or −1 to indicate an end-of-file. The status register is not altered unless the FILE= option is used.

When using SET(OPTION) PRINT, the output file must be built with records = 133 characters.

Examples

Assuming the items NAME, ADDRESS, CITY, DISCOUNT, and CUR-BAL have been defined and also specified in a LIST statement, 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. For example,

```
                          CUSTOMER LIST
CUST-NO:   FIRST-NAME:   LAST-NAME:   STREET-ADDR:     CITY:          ZIP:

 22431       John    Jones                5 Main Avenue    Centerville    12345

 34567       Mary    Smith                123 4th St.      Roseville      95747
```

The following example shows the use of the FILE= option to redirect formatted output. It routes EMPLOYEE-NAME, EMPLOYEE-ADDRESS, and SALARY to the MPE file "REPORT."

```
DISPLAY(FILE=REPORT) EMPLOYEE-NAME: EMPLOYEE-ADDRESS: SALARY;
```

END

Returns control to next higher level or structure.

Syntax

`END[`*modifier*`];`

The function of the END verb depends on the modifiers used.

Statement Parts

To specify the impact of the END verb, use one of the following modifiers:

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 with an E to exit from the program or an R to restart the program. Necessary only if program branches can cause more than one program end.

RESTART causes the following things to happen:

- List, key, update, match, and argument registers are reset (the data register is not reset).
- The work space is reset.
- Stack markers Z and DL are reset.
- MPE, KSAM, and form files are closed.

(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 whatever their conditions were immediately before the level sequence began.

If you use END without (LEVEL) to terminate a level, 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 description of the LEVEL verb in this chapter.

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

In this example, END terminates the command sequence and clears the program registers.

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

The following example terminates the program PROG1.

```
SYSTEM PROG1;
 .
 <<process program code>>
 .
END PROG1;
```

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

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

The following example 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 next statement is the first statement following the label, NEXT.

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

NEXT:
```

EXIT

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

Syntax

```
EXIT;
```

EXIT causes control to return to the operating system from a main program. If Transact was processing a called program, control returns to 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.

Syntax

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

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.)

Several of the above FILE operations can be performed by other Transact verbs.

```
For:   FILE(CLOSE)               Use:  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 data sets and KSAM files 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.

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*`[(`*subscript*`)]];`

FILE(CONTROL) specifies that the FCONTROL operation designated by CODE=*number* is to be performed. The value of *number* must be an unsigned integer (See 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*. The *item-name* may be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.) FILE(CONTROL) is the only statement that performs the FCONTROL functions on an MPE file.

To set a time-out interval for a DATA, INPUT, or PROMPT verb, use CODE=4 and let *item-name* equal the number of seconds of the time-out interval. In this case, *file-name* is the name of a dummy file defined in the SYSTEM STATEMENT. At run time, you should set up a file equation, FILE *file-name* = $STDLIST, using the dummy file specified in your program.

The FILE(CONTROL) statement only applies to the next access to the terminal, so it should appear immediately before the data entry statement to which it applies. (See the example at the end of this subsection.)

(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. The temporary sort file is purged upon exit of the Transact program.

MPE files can also be sorted with the FIND statement, but FILE(SORT) is more efficient.

(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 0 if the operation was successful or an undefined value if the operation was not successful.

Examples

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

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

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

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

This example is a complete program that can be used to familiarize yourself with setting a time-out interval before a data entry statement. Note that there are two loops, one nested in the other, with time-out applied only to the second PROMPT statement. The following file equate must be set at run time for the following program:

```
:FILE TERM=$STDLIST

SYSTEM TIMEO, FILE=TERM;
DEFINE(ITEM) TIME-OUT I(4):
             NUMBER   I(4);

LEVEL;
PROMPT TIME-OUT;

LEVEL;
FILE(CONTROL) TERM,CODE=4,PARM=TIME-OUT;
PROMPT NUMBER;
IF STATUS = -4 THEN DISPLAY "TIME OUT!";
```

FIND

Performs multiple retrievals from a file or data set.

Syntax

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

FIND executes multiple retrievals from a file or data set and places retrieved data in the data register one entry at a time. 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 by the match criteria.

Note After the first retrieval, Transact uses an asterisk (*) for the call list to optimize subsequent retrievals of that data set.

Statement Parts

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

none	Retrieves an entry from a master set based on the key value in the argument register. This option does not use the match register.
CHAIN	Retrieves entries from a KSAM file key or a detail 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 or KSAM key in which the retrieval is to occur. If no match criteria are specified, all entries on the chain are selected. Items used in the match criteria must be included in the LIST= option.
CURRENT	Retrieves the last entry that was accessed from the file or data set.
DIRECT	Retrieves the entry stored at a specified record number from an MPE or KSAM file or a data set. Before using this modifier, you must store the record number as a 32-bit integer in the item referenced by the RECNO= option.
PRIMARY	Retrieves the 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 — Retrieves entries from a detail set in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.

RSERIAL — Retrieves entries from a data set in the same manner as the SERIAL option, except in reverse order. If an equal match without match characters exists, Transact will convert an RSERIAL option to an RCHAIN option to improve the application's efficiency. For a KSAM or MPE file, this operation is identical to SERIAL.

SERIAL — Retrieves entries in serial mode from an MPE or KSAM file or a data set that meet any match criteria set up in the match register. If an equal match without match characters exists, Transact will convert an SERIAL option to an CHAIN option to improve the application's efficiency. 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.

Note

FIND(SERIAL) or FIND(RSERIAL) with the PERFORM= option on a master set will skip entries if a delete is done within the perform, and a secondary entry migrates to the position of the deleted entry. (Transact/V Only.)

---

file-name — The file or data set to be accessed by 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*])] — Suppresses the default error return 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. The *item-name* must be a parent.

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 touched. For more information, see "Automatic Error Handling," in Chapter 7.

LIST=(*range-list*) — The list of items from the list register to be used for the FIND operation. For data sets, no child items can be specified in the range list.

If the LIST= option is omitted with any modifier, all the items named in the list register are used.

When the LIST= option is used, only the items specified in a LIST= option have their match conditions applied when the items are included in the match register. When the LIST= option is omitted, items which appear in the list register and the match register have their match conditions applied. Otherwise, the match conditions for an item are ignored. The match register can 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 retrieval is performed.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

The LIST= option has a limit of 64 individually listed item names and a limit of 255 items specified by a range for a TurboIMAGE data set.

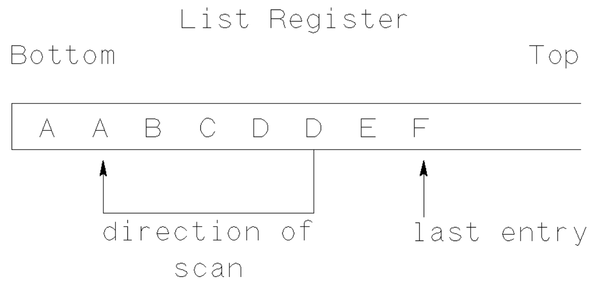
All item names specified must be parent items.

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

(*item-name*)
: A single data item.

(*item-nameX: item-nameY*)
: All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

  Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items (see the DEFINE(ITEM) verb), and if there are no data items between the two on the list register, no database access is performed.

(*item-nameX:*) All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.

(:*item-nameY*) All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.

(*item-nameX, item-nameY, ... item-nameZ*) The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files, data items must be specified in the order of their occurrence in the physical record. This order need not match the order of the data items on the list register. This option incurs some system overhead.

(@) Specifies a range of all data items of *file-name* as defined in the data dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.

(#) Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the data dictionary. This order need not match the order of the data items in the list register.

() — A null data item list. That is, the entry or entries are read, but do not retrieve any data.

LOCK — Locks the specified file or database. The lock is active the whole time that the FIND executes. If LOCK is not specified and a TurboIMAGE data set is being accessed, no locking is done.

For a KSAM or MPE file, if LOCK is not specified on FIND but is specified for the file in the SYSTEM statement, then the file 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. Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the FIND verb.

For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on FIND if SET(OPTION) NOLOCK is specified.

See "Database and File Locking" in Chapter 6 for more information on locking.

NOMATCH — Ignores any match criteria set up in the match register.

NOMSG — Suppresses the standard error message produced as a result of a file or database error.

PERFORM=*label* — Executes the code following the specified label for every entry retrieved by FIND. The entries can 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 the need to code loop-control logic.

You can nest up to 10 PERFORM= options.

RECNO=*item-name* [(*subscript*)] — The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 2.) With the DIRECT modifier, you must define *item-name* to contain the 32-bit integer number (I(9,,4)) of the record to be retrieved.

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

SINGLE — Retrieves only the first selected entry.

SOPT

Suppresses the optimization of database calls. SOPT forces Transact to re-establish its path, list, and record pointers before each record is used. Use SOPT if you are calling TurboIMAGE through the PROC or CALL verbs within a PERFORM option, or if you use the same FIND verb recursively for TurboIMAGE access. For an example of how SOPT is used, see "Examples" at the end of the FIND verb description.

SORT=(*item-name1*[(ASC)] [,*item-name2*[(ASC)]
[(DES)] [(DES)]]])

FIND creates a work file of the records selected when the SORT option is specified. FIND sorts each data entry or record by *item-name1* and, optionally, *item-name2*, and so forth. The key items in the SORT= option must also be included in the LIST= option (they can be child items); the items in the LIST= option are the record definition for the sort file. You can specify ascending (ASC) or descending (DES) sort order for each item. The default is ascending order.

The FIND statement only creates and 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, passes each record of data to the data register,
- retrieves each selected record,
- then writes each selected record to the sort file,
- sorts the sort file by any specified items, and
- passes each record one by one to the PERFORM= statements.

The sort file size is determined by the SYSTEM statement.

STATUS

Suppresses the actions defined in Chapter 7 under "Automatic Error Handling." You may want to add status checking to your code if you use this option.

When STATUS is specified, the effect of a FIND statement is described by the 32-bit integer 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 database or MPE/KSAM file system error documentation that corresponds 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).

See "Using the STATUS Option" in Chapter 7 for a discussion of how to use STATUS data.

WORKFILE

FIND creates a work file of the records selected when the WORKFILE option is specified. The FIND statement only creates the work file if a PERFORM option is also included. The processing sequence for a work file is:

- first, passes each record of data to the data register,
- evaluates each record selecting those that meet the MATCH criteria,
- then writes each selected record to the work file,
- passes each record one by one to the PERFORM statements.

If the SORT and WORKFILE options are both used in a single verb, the work file is sorted according to the SORT option.

Suppression of Optimization versus WORKFILE

Transact's features resolve issues associated with retaining the correct location in a file or data set on multiple retrieval verbs (OUTPUT, FIND, DELETE, REPLACE) when the PERFORM procedure also operates on the same file or data set. These multiple retrieval verbs are optimized to avoid repositioning them before each record or entry is read.

Automatic Optimization

Transact tries to optimize the TurboIMAGE/KSAM interface for the set of multiple retrieval verbs. If Transact determines that the current multiple retrieval verb is the only verb accessing the file or data set within a program, optimization can occur.

Automatic Suppression of Optimization

Transact automatically suppresses the optimization of TurboIMAGE and KSAM calls when more than one verb accesses the same file or data set within a program. On multiple retrieval verbs, automatic suppression allows a different path for each access of the file or data set. This feature is always active.

Automatic suppression of optimization occurs when:

- both a FIND verb and its PERFORM option access data set X.
- both a FIND verb and its PERFORM option access KSAM file Y.

Suppression of Optimization Limitations

There are situations where the automatic suppression of optimization is limited. It is either not invoked, or optimization is not sufficient to prevent multiple retrieval verbs from losing their location in the file or data set. These situations are described below.

The SOPT Option

Transact cannot detect the need for suppression of optimization in three specific situations. The SOPT option on multiple retrieval verbs is intended to handle these situations where suppression is needed but is not activated automatically. This can occur under the following situations:

- The PERFORM option is a recursive call.
- A PROC verb is used within the PERFORM option to call a procedure that accesses the same file or data set as the multiple retrieval verb.
- A CALL verb is used within the PERFORM option to call another Transact system that accesses the same file or data set as the multiple retrieval verb.

Corrupted Location in the File/Data Set

Adding, deleting, updating, or replacing more than one record from within the PERFORM option procedure of a multiple retrieval verb can cause the multiple retrieval verb to lose its location if the current and next/previous logical records in the chain are deleted.

Revisiting a Record

When records are added, updated, or replaced from within the PERFORM procedure, these new or changed records can be retrieved a second time by the multiple retrieval verb. The specific conditions where an updated or added record can be retrieved a second time depend on the access mode of the multiple retrieval verb. For a multiple retrieval verb using the CHAIN or RCHAIN modifier where the key item is a sorted key, revisiting can occur:

- When a PUT verb adds a record between the current record and the last record in the chain.
- When a REPLACE verb updates any item, and the TurboIMAGE critical item update is OFF.
- When an UPDATE verb updates a sort item or extended sort item, and the TurboIMAGE critical item update is ON.

- When a REPLACE verb is used to replace a key value other than the key in the current path.
- When multiple PUTs, UPDATEs, REPLACEs, or DELETEs are done within the PERFORM option procedure, and the last operation is not a delete of the current record for the multiple retrieval verb.

For a multiple retrieval verb using the SERIAL or RSERIAL modifier, revisiting may occur:

- When a PUT verb adds a record to the data set.
- When a REPLACE verb update is used to replace a record in the data set.

Note With the CHAIN and RCHAIN access method, SORTED keys can cause revisiting of an entry. Transact multiple retrieval verbs retain the original end of chain location and stop processing after this record is read. Therefore, any records added to the chain after the original end of the chain record will not be processed.

Using the WORKFILE Option to Remedy Optimization Limitations

The WORKFILE option can be used to remedy these optimization limitations, but other options can yield better performance.

In situations where it is undesirable to have new or modified records reread by the multiple retrieval verb, you can use two tactics:

- If the access is CHAIN or RCHAIN and the key item is a sorted key, the access direction can be changed to place added/updated records on a part of the chain you have already processed. If the access is SERIAL or RSERIAL, there is no way to control access to eliminate new or updated records.
- For either SERIAL or CHAIN access, the MATCH register can be used to filter out records that have already been processed. If the current record is to be deleted by the PERFORM option, do this as the last operation against the data set.

Using the WORKFILE Option

If none of the above techniques allow the multiple retrieval verb to process the file or data set as desired, you can use the WORKFILE option. In terms of performance, this option is the least desirable of any of the methods mentioned above. This option should be used under the following specific circumstances:

- When multiple PUTs, DELETEs, UPDATEs, or REPLACEs done within the PERFORM procedure of a multiple retrieval verb cause the multiple retrieval verb to lose its location.
- When no other method for eliminating reprocessing records added to the file or data set via a REPLACE, UPDATE, or PUT can be found and reprocessing would damage the record.
- When the PERFORM procedure alters the MATCH register in such a way that the MATCH conditions are no longer valid for the calling multiple retrieval verb.

Examples

In the following example of FIND, use of the STATUS option suppresses automatic error handling. The STATUS option enables you to perform a routine to control operations when an end of chain or broken chain occurs.

```
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;
```

Instead of using the STATUS option, (such as using automatic error handling), you could 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 set, the status register contains a −1 when the argument value is not in the master set.

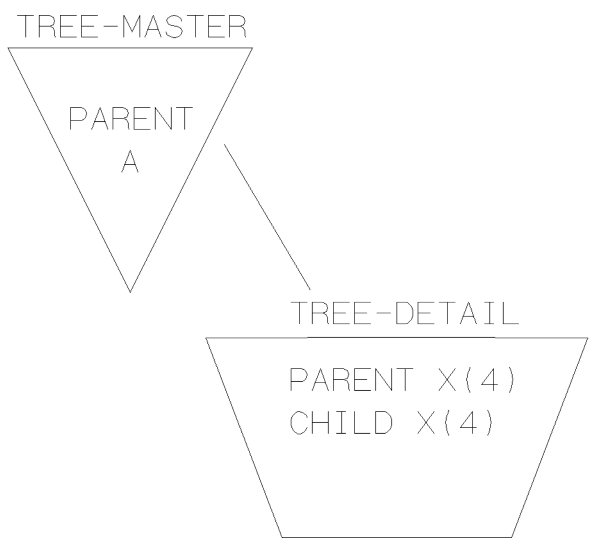
The following example uses a PERFORM= option to test data values in each retrieved entry. The routine TEST1 is performed on every record retrieved by FIND(CHAIN).

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

```
        .
        .
        DISPLAY ...;
        RETURN;
```

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

Assume that the database TREE has the following structure:




```
    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;
```

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-IT 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-IT;
            .
            .
      DISPLAY-IT:
         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;
```

This example shows the use of the WORKFILE option. All qualified records have their record number written to a work file. This file will be used to retrieve records instead of the TurboIMAGE chain. This example assumes that SHIP-DATE is a sort or search item within the TurboIMAGE data set ORDER-DET.

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

   FIND(CHAIN) ORDER-DET,
        LIST=(ORD-NO:SHIP-DATE),
        WORKFILE,
        PERFORM=INCDATE;
        .
        .
INCDATE:
     LET (SHIP-DATE)=(SHIP-DATE)+3;
     UPDATE ORDER-DET,
         LIST=(ORD-NO:SHIP-DATE);
```

FORMAT

Specifies the format of information displayed by the OUTPUT verb or by an unformatted DISPLAY verb.

Syntax

`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 re-used 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.

When native language support is used, the decimal and thousands indicators are language sensitive. As indicated below, some of the EDIT= mask characters are also language sensitive. (See Appendix E, "Native Language Support," for more information.)

The default format is:

- Displays the values in the order in which they appear in the data register.
- Accompanies 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 a data 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 from the next. 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 separated by a colon. The fields are separated from their formatting parameters by commas as shown below:

display-field[,*format-parameter*] . . .
[:*display-field*[,*format-parameter*] . . .] . . .

If you omit *display-list*, the display is formatted according to the default format described earlier in this verb description.

display-field The following options can be used for display fields:

- A reference to a data item name in the list register (the item name may be subscripted if an array item is being referenced).
- 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 Transact generates an error at execution time.

Five system variables can also be used as display fields. As noted, some are affected by native language support. (See Appendix E, "Native Language Support," for more information.)

$CPU	displays the cumulative amount of CPU time used by the Transact program, in milliseconds.
$DATELINE	displays the current date and time in the form Tue, Apr 14, 1992, 3:07 P.M. The format is affected by native language support.
$PAGE	displays the current page number.
$TIME	displays the current time; the default format is HH:MM AA (for example, 03:07 PM). The format is affected by native language support.
$TODAY	displays the current date; the default format is MM/DD/YY (for example, 04/14/92). The format is affected by native language support.

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

CCTL=*number* Issues 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*. Note that the use of CCTL=*number* and

	LINE, NOCLRF, or ROW, may affect output due to conflicting values. Valid range is 30–1025.
CENTER	Centers a display field on a line. The entire field, including leading or trailing blanks, is centered.
COL=*number*	Starts the display field in the absolute column position specified by *number*. The first column position is 1. Maximum is 299. If the display is already at a column position equal to or 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 meanings when used in the *edit-string* for all *display-fields* except the system variables $TIME and $TODAY:

^	Inserts the character from the source data field into this position in the display field.
Z	Suppresses leading zeros. Note that you must use an uppercase Z.
$	Adds business (single character) currency symbol. If the language-defined currency symbol precedes, then the symbol is floated. If the symbol succeeds, then it follows the last character of the number and the edit mask is shifted left one character to leave room. If the symbol imbeds, it replaces the radix (decimal point or equivalent). If no business currency symbol is defined for the current language, then "$" edit characters are treated the same as "other" edit characters, explained below.

Note

The number of digits available for the source number depends on the type of currency symbol. Thus, the same value might cause a field overflow in some languages and not in others.

*	Fills field with leading asterisks.
.	Aligns the implied decimal point as specified in a data dictionary or in a DEFINE(ITEM) definition statement with this edit character in the edit mask and output the language defined radix character.

! Ignores the implied decimal place and replaces this character with a language defined decimal character.

' Outputs the language defined thousands separator character (numeric only).

(Surrounds negative values with parentheses (must be last character in the edit mask).

All "other" characters, which means any character not defined above in the list of special characters, are treated as insert characters. For example:

```
EDIT="@@@@@@.@@"
```

displays entered data as:

```
@@@@@@.@@
```

To denote numeric data type 9, Z, P, I, J, K, R, or E negative values with a trailing "-", "CR", or "DR", add a trailing "-", "CR", or "DR" to the edit string. Some edit-string examples follow:

Number	Edit String	Result
1234	$$,$$$!^^	$12.34
123456	$$,$$$!^^	$1,234.56
123456	***,**$!^^	*$1,234.56
000009	ZZZZ!^^	.09
475.49	XXX,XXX.XX	XXX,XXX.XX
-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 the hour with no leading blank or zero if hour < 10.

ZH Displays the hour with leading blank if hour < 10.

HH Displays the hour with leading zero if hour < 10.

24 Displays the hour as expressed on a 24-hour clock; used as a prefix to H.

M Displays the minute with no leading blank or zero if minute < 10.

ZM	Displays the minute with leading blank if minute < 10.
MM	Displays the minute with leading zero if minute < 10.
S	Displays the second with no leading blank or zero if second < 10.
ZS	Displays the second with leading blank if second < 10.
SS	Displays the second with leading zero if second < 10.
T	Displays the 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.
AA	Displays both letters in the AM or PM sequence in uppercase.
aa	Displays both letters 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 the day of the month with no leading blank or zero if day < 10.
ZD	Displays the day of the month with leading blank if day < 10.
DD	Displays the day of the month with leading zero if day of the month < 10.
DDD	Displays the Julian day of year.
M	Displays the month with no leading blank or zero if month < 10.

ZM	Displays the month with leading blank if month < 10.
MM	Displays the month with leading zero if month < 10.
*n*M	Displays the first *n* letters of month name in uppercase; if $n >$ number of letters in month name, trailing blanks are not inserted.
*n*m	Displays the first *n* letters of month name in lowercase except for the first letter, which appears in uppercase.
YY	Displays the last two digits in current year.
YYYY	Displays the current year.
*n*W	Displays the first *n* letters of day of week in uppercase; if $n >$ length of the week name, no trailing blanks are inserted.
*n*w	Displays the 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 April 14, 1992, make it appear as follows:

```
      Edit Mask          Displayed Date

   3w. 3m DD, YYYY      Tue. Apr 14, 1992
   DD 3M, YY            14 APR 92
   M-DD-YY              4-14-92
   MM/DD/YY             04/14/92
   DDD, YYYY            105, 1992
```

Note

When a numeric value to be printed is too large for the edit mask, a series of pound signs (#) are printed in place of the value, to indicate an overflow.

HEAD="*character-string*"	Uses the *character-string* as the heading rather than the default, which is the heading from a data dictionary, the heading from DEFINE(ITEM), or the item or system variable name.
JOIN[=*number*]	Places 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. The default is 1; valid range is 0–299.
LEFT	Left-justifies the data item value in the display field. This is the default specification.
LINE[=*number*]	Starts 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 overprint and you want it to do so, you should specify LINE=0. [LINE=0 and LINE= cause a carriage return but no line feed.] The default is 1; maximum is 99.
LNG=*number*	Truncates 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. Range is 1–299.
NEED=*number*	Prints 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. If you are grouping a set of items together on a single line, the NEED= must appear with the first item on the page. Range is 1–99.
NOCRLF	Does not issue a carriage return and line feed for the display line containing the display field. NOCRLF is processed when a listing goes to the terminal or printer. If the listing is sent to a disk file, the option is ignored.
NOHEAD	Suppresses the default heading for this item reference.
NOSIGN	Allows no sign position in the display field. If a negative value occurs, the display field contains a string of minus signs (-).
PAGE[=*number*]	Starts the display field on a new page or on a page after a page skip count specified by *number*. The default is 1; maximum is 99.
RIGHT	Right-justifies the data item value in the display field.
ROW=*number*	Places 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. Maximum is 99.
SPACE[=*number*]	Places 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 is 1; maximum is 299.
TITLE	Displays the associated display field and any preceding display fields only at the start of each new page for which this statement applies.

TRUNCATE	Truncates 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-justifies a numeric data value in the display field and inserts 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.

Syntax

`GET` [(*modifier*)] *source* [, *option-list*] ;

GET retrieves a single entry from a data set or KSAM or MPE file after rewinding the file or data set. 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
: For master sets, retrieves a master set entry based on the value in the argument register. For MPE files, the next entry is serially read. For KSAM files, this option does not use the match register.

CHAIN
: Retrieves an entry from a 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
: Retrieves the last entry that was accessed from the data set or the MPE or KSAM file.

DIRECT
: Retrieves the entry stored at a specified record number in an MPE or KSAM file, or a detail or master set. Before using this modifier, you must store the record number as a 32-bit integer in the item specified in the RECNO= option.

FORM
: GET(FORM) displays a VPLUS form on any VPLUS compatible terminal and then waits for the user to press ENTER 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
: Executes a calculated access on a master set using the key and argument register contents, but transfers no data. The LIST= option cannot be specified with this modifier. (Use GET with no modifier for a calculated retrieval from a master 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 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	Retrieves the 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	Retrieves an entry from a 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	Retrieves an entry from a data set in the same manner as the SERIAL option, except in reverse order. If an equal match without match characters exists, Transact will convert an RSERIAL option to an RCHAIN option to improve the application's efficiency. For a KSAM or MPE file, this operation is identical to SERIAL.
	SERIAL	Retrieves an entry in serial mode from an MPE or KSAM file or a detail or master set. It retrieves the first entry that matches any match criteria set up in the match register. If an equal match without match characters exists, Transact will convert an SERIAL option to an CHAIN option to improve the application's efficiency. 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.

source The file, data set, or form to be accessed by 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* can be specified as any of the following:

form-name	Name of the form to be displayed by GET(FORM).
(*item-name* [(*subscript*)])	Name of an item that contains the name of the form to be displayed by GET(FORM). *subscript* can be included if the referenced item is an array item. (See "Array Subscripting" in Chapter 3.)
*	Displays 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.
&	Displays 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. For GET(FORM) ONLY, items in the range list can be child items.

If the LIST= option is omitted for GET(FORM), the list of items named in the list register, and either in the SYSTEM statement or the data dictionary for the form are used.

The LIST= option should not be used when specifying an asterisk (*) as the source.

When the LIST= option is used, only the items specified in a LIST= option have their match conditions applied when the items are included in the match register. When the LIST= option is omitted, items which appear in the list register and the match register have their match conditions applied. Otherwise, the match conditions for an item are ignored.

The match register can be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

The LIST= option has a limit of 64 individually listed item names. A range limitation of 255 items for TurboIMAGE data sets and 128 items for VPLUS forms also exists.

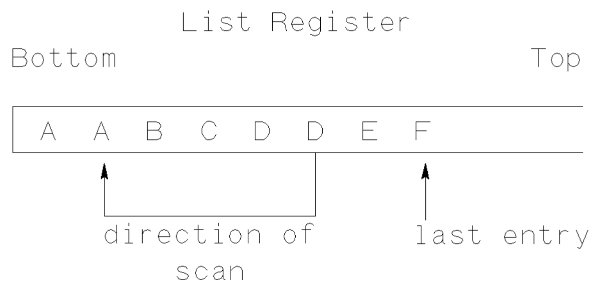
All item names specified must be parent items.

The options for *range-list* and the data items they cause GET to retrieve include the following:

(*item-name*) A single data item.

(*item-nameX:* *item-nameY*) All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items (see the DEFINE(ITEM) verb), and if there are no data items between the two on the list register, no database access is performed.

(*item-nameX:*) All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.

(:*item-nameY*) All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.

(*item-nameX*, *item-nameY*, ... *item-nameZ*) The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files or for VPLUS forms, data items must be specified in the order of their occurrence in the physical record or form. This order need not match the order of the data items on the list register. Does not include child items in the list unless they are defined in the VPLUS form. This option is less efficient to use than the options listed above.

(@) Specifies a range of all data items of *file-name* as defined in a data dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.

(#) Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the data dictionary. This order need not match the order of the data items in the list register.

()	A null data item list. That is, accesses the file or data set, but does not retrieve any data.

Options Available Without the Form Modifier

ERROR=*label* ([*item-name*])	Suppresses the default error return that Transact normally takes. Instead, branches to the statement identified by *label*, and sets the stack pointer for the list register to the data item *item-name*. Transact generates an error at execution time if the item cannot be found in the list register. The *item-name* must be a parent. 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 Chapter 7.
LOCK	Locks the specified file or database. The lock is active the whole time that the GET executes. If LOCK is not specified and a TurboIMAGE data set is being accessed, no locking is done. For a KSAM or MPE file, if LOCK is not specified on GET but is specified for the file in the SYSTEM statement. The file is then locked before each entry is retrieved, remains locked while the entry is processed by any PERFORM= statements, and is unlocked briefly before the next entry is retrieved. Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the GET verb. For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on GET if SET(OPTION) NOLOCK is specified. For more information on locking, see "Database and File Locking" in Chapter 6.
NOFIND	Ensures that a matching entry is not present in the referenced master 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	Ignores any match criteria set up in the match register.
NOMSG	Suppresses the standard error message produced as a result of a file or database error. All other error actions occur.
RECNO=*item-name* [(*subscript*)]	With the DIRECT modifier, you must define *item-name* to contain the 32-bit 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 32-bit integer *item-name*. The *item-name* can be modified with *subscript* if the referenced item is an array item. (See "Array Subscripting" in Chapter 3.)

STATUS

Suppresses the action defined in the Chapter 7 under "Automatic Error Handling." You may want to add status checking to your code if you use this option.

When STATUS is specified, the effect of a GET statement is described by the 32-bit integer value in the status register:

Status Register Value	Meaning
0	The GET operation was successful.
−1	A KSAM or MPE end-of-file condition for serial read or end-of-chain for chain read has occurred.
>0	For a description of the condition that occurred, refer to database or MPE/KSAM file system error documentation that corresponds to the value.
1	If NOFIND is used and the record is found.

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).

See "Using the STATUS Option" in Chapter 7.

Options Available Only With the Form Modifier

APPEND

Appends 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

Accepts any function key not specified in an F*n=label* option to transfer data from the form to the data register. If a key has been specified in an F*n=label* option, GET does not execute AUTOREAD for that key.

CLEAR

Clears the previously displayed form when the requested 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

Uses the form currently displayed on the terminal screen; that is, it performs 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.

CURSOR=field-name|*item-name*[(*subscript*)]

Positions the cursor within the specified field. Field-name identifies the field and the item-name names the item identifying the field. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

Note To ensure that the cursor will be positioned on the correct field, you must have a one to one correspondence between the fields defined in VPLUS. Transact determines where to position the cursor by counting the fields.

FEDIT
: Performs the field edits defined in the FORMSPEC definition immediately before displaying the form.

FKEY=*item-name* [(*subscript*)]
: Moves the number of the function key the operator presses in this retrieval operation to the single word integer (I(4)) *item-name*. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.) The function key is determined by the contents of the field LAST-KEY in the VPLUS comarea. It can 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. Note that pressing [f8] returns an 8 in the item field and does not cause an automatic exit.

F*n*[(AUTOREAD)]= *label*
: Control passes to the labelled statement if the operator presses function key *n*. This option can be repeated for each desired function key as many times as necessary in a single GET(FORM) statement. If (AUTOREAD) is included, transfers the data from the form to the data register before going to the specified label. F0, or ENTER, always transfers data. This option is cancelled by the STATUS option.

FREEZE
: Freezes the specified form and appends 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
: Initializes the fields in a VPLUS form to any initial values specified for the form by FORMSPEC, or performs any Init Phase processing specified for the form by FORMSPEC. The INIT processing is performed before the form is displayed on the screen.

STATUS
: Suppresses 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. The STATUS option suppresses any branch specified by Fn= *label*. If field edit errors exist, Transact sets the value of the 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*)
: Places a message in the window area of the screen and, optionally, enhances a field in the form. The fields *field* and *message* can be specified as follows:

  field
  : Either the name of the data item for the field to be enhanced, or an *item-name*[(*subscript*)] within parentheses which will contain the data item 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*[(*subscript*)] within parentheses containing the message string to be displayed in the window.

Examples

The following example shows the use of the WINDOW 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));
```

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-NO;
LIST CUST-NAME:
     CUST-PHONE;
GET(CHAIN) 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 DATA(MATCH) statement.

```
PROMPT(PATH) CUST-ID;
LIST CUST-NAME:
     CUST-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);
```

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.

```
GET(FORM) CUSTFORM, INIT, LIST=(CUST-NAME, CUST-ADDR, CUST-PHONE);
```

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;
```

The following example shows a method for "structured programming" with VPLUS forms. Each RETURN statement passes control back to the PERFORM statement.

```
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.

GO TO

Transfers control to a labeled Transact statement.

Syntax

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

The following statement transfers control to the statement labeled "NEW-TOTAL".

```
GO TO NEW-TOTAL;
```

IF

Performs a specified action based on a conditional test.

Syntax

`IF` *condition-clause* `THEN` *statement* [`ELSE` *statement*] ;

IF specifies tests to be performed on *test-variables* . IF introduces a *condition-clause*, which contains one or more conditions, each made up of a *test-variable*, a *relational-operator*, and one or more *values*. Multiple conditions are joined by AND or OR. If the condition clause is true, then the specified statement is performed. You can 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

One or more conditions, connected by AND or OR, where

AND — A logical conjunction. The condition clause is true if all of the conditions are true; it is false if one of the conditions is false.

OR — A logical inclusive OR. The condition clause is true if any of the conditions is true; it is false if all of the conditions are false.

Each condition contains a *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value [,*value*] ...

test-variable — Can be one or more of the following:

(*item-name* [(*subscript*)]) — The value in the data register that corresponds to *item-name*. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

[*arithmetic expression*] — An arithmetic expression containing item names and/or constants. The expression is evaluated before the comparison is made.

Note

An *arithmetic-expression* must be enclosed in square brackets ([ ]).

EXCLA-MATION	Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. (See "Data Entry Control Characters" in Chapter 5.) If the null response is set, the EXCLAMATION test variable is a positive integer. If it is not set, it is zero. The default is 0.
FIELD	Current status of FIELD command identifier. If a user qualifies a command with FIELD, the FIELD test variable is a positive integer. Otherwise, it is a negative integer. The default is <0.
INPUT	The last value input in response to the INPUT prompt.
PRINT	Current status of PRINT or TPRINT command qualifier. 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. The default is <0.
REPEAT	Current status of REPEAT command qualifier. If a user qualifies a command with REPEAT, the REPEAT test variable is a positive integer. Otherwise, REPEAT is a negative integer. The default is <0.
SORT	Current status of SORT command qualifier. If a user qualifies a command with SORT, the value of the SORT test variable is a positive integer. Otherwise SORT is a negative integer. The default is <0.
STATUS	The 32-bit integer 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 can 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 value can be an arithmetic expression that will be evaluated before the comparison is made.

The allowed value depends on the test variable, as shown in the comparison below. Alphanumeric strings must be enclosed in quotation marks.

If the *test-variable* is:	The value must be:
item name	Then *value* must be an alphanumeric string, a numeric value, an arithmetic expression, a reference to a variable as in (*item-name*), or a class condition as described below.
[*arithmetic expression*]	A numeric value, an arithmetic expression, or an expression, or a reference to a variable as in (*item-name*).
INPUT	An alphanumeric string.
EXCLA-MATION FIELD PRINT REPEAT SORT	A positive or negative integer or expression.
STATUS	A 32-bit integer number or expression.

If more than one value is given, then:

- The *relational-operator* can be only "=" or "<>".
- When the relational operator is "=", 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.
- When the relational operator is "<>", the action is taken if the *test-variable* is not equal to *value1* AND *value2* AND ... *valuen*. In other words, a comma in a series of values is interpreted as an AND when the operator is "<>".

When the test variable is an *item-name*, the *value* can be one of the following class conditionals, which are used to determine whether a string is all numeric or alphabetic. The operator can only be "=" or "<>".

NUMERIC This class condition includes the ASCII characters 0 through 9 and a single operational leading sign. Leading and trailing blanks around both the number and sign are ignored. Decimal points are not allowed in NUMERIC data. This class test is only valid when the item has the type X, U, 9, or Z, or when the item is in the input register.

ALPHABETIC	This class condition includes all ASCII native language alphabetic characters (upper and lowercase) and space. This class test is only valid for items of type X or U or when the item is in the input register.
ALPHABETIC-LOWER	This class condition includes all ASCII lowercase native language alphabetic characters and space. This class test is only valid for items of type X or U or when the item is in the input register.
ALPHABETIC-UPPER	This class condition includes all ASCII uppercase native language alphabetic characters and space. This class test is only valid for items of type X or U or when the item is in the input register.

statement Any simple or compound Transact statement; a compound statement is one or more statements bracketed by a DO/DOEND pair.

Order of Evaluation

When complex conditions are included, the operator precedence is:

- Arithmetic expressions are evaluated.
- Truth values are established for simple relational conditions.
- Truth values are established for simple class conditions.
- Multiple value conditions are evaluated.
- Truth values are established for complex AND conditions.
- Truth values are established for complex OR conditions.

Parentheses can be used to control the order of precedence when conditional clauses are being evaluated. In multiple value conditions, evaluation terminates as soon as a truth value is determined.

Examples

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 INPUT = "YES", "Y" THEN
   GO TO PROCEED;
```

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 (COUNT) > 3 THEN
   GO TO TOO-HIGH;
```

This statement causes an exit from the current command sequence if the status register value does not equal 0.

```
IF STATUS <> 0 THEN END;
```

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.

```
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 next example gives the user a choice between two activities. The second ELSE construct checks to see that the user did one of the two specified activities. If he did not, a message is displayed, and control returns to the label OPTION at the third line, so that the user is prompted again.

```
SYSTEM IFS;
DEFINE(ITEM) FIELD I(2);
OPTION:
PROMPT FIELD;
  IF (FIELD) = 1 THEN
    DO
      DISPLAY "FIELD = 1";
    DOEND
  ELSE
    DO
    IF (FIELD) = 2 THEN
      DO
        DISPLAY "FIELD = 2";
      DOEND
    ELSE
      DO
        DISPLAY "YOU MUST ENTER 1 OR 2";
        GO TO OPTION;
      DOEND;
    DOEND;
  END;
```

The next examples demonstrate how to use complex conditionals.

```
IF (LAST-NAME) = "SMITH" AND (FIRST-NAME) = "JACK" THEN ...

IF (ACCT-BALANCE) < 0 OR (LOAN-AMOUNT) >= (LOAN-MAX) THEN ...

IF (RENTAL-OFFICE) = "098","978","656" AND
   (CUST-NO-PREFIX) = (PREFERRED-PREFIX) OR
   (CUST-NAME) = "ABCINC" THEN ...

WHILE (BALANCE) < 0 AND STATUS = 0
   DO
     GET(CHAIN) CUST-DETAIL,STATUS;
     LET (BALANCE) = (BALANCE) + (AMOUNT);
   DOEND;

REPEAT
   DO
     LET (TOTAL-OVERDUE) = (TOTAL-OVERDUE) + (AMT-OVERDUE);
     FIND(SERIAL) CUST-INVOICE,STATUS;
   DOEND
UNTIL (TOTAL-OVERDUE) > 999999.99 OR
      (TOTAL-OVERDUE) > (MIN-OVERDUE) AND
      (CUST-CODE) = "NEW";
```

The next examples demonstrate the use of the relational operator "<>" with multiple values.

```
IF (STATE) <> "OR","CA","CO","VA" THEN ...

WHILE (PART-NO-PREFIX) <> (PROTOTYPE),(DEVELOPMENT)
     GET(CHAIN) PART-DETAIL,STATUS;
```

The next examples demonstrate the use of class conditionals.

```
IF INPUT = ALPHABETIC THEN ... ELSE ...;

DATA (PART-NUMBER);
IF (PART-NUMBER) <> NUMERIC THEN ...;
```

The next example demonstrates the use of multiple expressions in *test-variables* or in *values*.

```
IF (AREA) = [(LENGTH)*(WIDTH)],[(BASE)*(HEIGHT)*.5],
              [(3.1416)*(RADIUS)**2] THEN ...;

REPEAT

    FIND(SERIAL) STK-ON-HAND,STATUS

UNTIL ((WEIGHT) > [(KILO-PER-METER) * (METERS)] AND
       (METERS) > (MIN-LENGTH) OR
       (PRICE) > [(UNIT-PRICE) * (KILO-PER-METER) * (METERS)]);

IF [(DELAY) * (DFACTOR)] = [(COUNT) * 3] THEN ...;
```

INPUT

Prompts for a value and places it in the input register.

Syntax

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. The response can 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 cOne or more of the following options separated by commas:

BLANKS Does not suppress leading blanks supplied in the input value.

NOECHO Does not echo the input value to the terminal.

STATUS Suppresses normal processing of "]" and "]]", which cause an escape to a higher processing or command level.

Status Register Value	Meaning
−1	User entered a "]".
−2	User entered a "]]".
−3	User entered one or more blanks and no non-blank characters.
−4	If timeout is enabled with a FILE(CONTROL) statement, a timeout has occurred.
> 0	Number of characters (includes leading blanks if BLANKS option is specified); no trailing blanks are counted.

The STATUS option allows you to control subsequent processing by testing the contents of the register with an IF statement.

Examples

This example shows 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.

```
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;
```

ITEM

Defines variables for use in the program that have not been defined in a data dictionary. The DEFINE(ITEM) verb is preferred. See DEFINE(ITEM) in this chapter for syntax option descriptions and additional information.

LET

Specifies arithmetic operations.

Syntax

LET *destination-variable* = *arithmetic-expression* [,ERROR=*label*[([*item-name*])]] ;

The LET verb is primarily used to perform arithmetic operations.

Note

At one time the LET verb was also used to manipulate arrays through an optional syntax variation that used the LET OFFSET option. However, the current version of Transact supports subscripting of arrays so that use of the LET OFFSET is no longer necessary. Although it is now recommended that you use subscripts to manipulate arrays, the LET OFFSET option is still available and is described later to aid in maintaining older Transact programs.

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 Transact-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 names listed below. All of the names except *item-name* are stored in a special area outside the list and data registers. They are, therefore, not affected by SET(STACK) or RESET(STACK).

(*item-name* [(*subscript*)])	The computed or assigned value of *item-name*. The *item-name* identifies a location in the data register. The *item-name* can be subscripted if an array item is referenced. (See "Array Subscripting" in Chapter 3.)
LINE	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the computed or assigned value of the line counter for the current line of terminal display or line printer output.
OFFSET (*item-name*)	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the offset of an item starting at position zero.
PAGE	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the computed or assigned value of the page counter.
PLINE	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the computed or assigned value of the line counter for the current line of line printer output.

STATUS	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the computed or assigned value of the status register.
TLINE	An integer, defined as I(5,,2) in Transact/V or defined as I(10,,4) in Transact/iX, that contains the computed or assigned value of the line counter for the current line of terminal display output.

arithmetic-expression A single source, or multiple sources connected by arithmetic operators in the format:

[-]*source1* [*operator source2*] ... [*operator sourcen*]

[-] If the expression is preceded by a minus sign, its negative is assigned.

Note If a positive-only integer is set to a negative number, an error occurs. The value of the specified item will be undefined. Since the outcome is undefined, you should not rely on this procedure to zero out values. Instead, use the ERROR= option to branch to a label and negate the desired fields.

---

source An *item-name*[(*subscript*)] within parentheses, a numeric constant, one of the Transact-defined names listed above under the description of *destination-variable*, or one of the *function*s listed below and described later in this verb.

The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

function

```
ASCII
LENGTH
LN
LOG
POSITION
SQRT
VALUE
```

operator

+	addition
−	subtraction
*	multiplication
/	division giving the quotient
//	division giving the remainder
**	exponentiation

ERROR=*label* [([*item-name*])] An option to cause branching on arithmetic errors. In addition to branching and resetting the list register, this option causes the status register to be set to a value that identifies the type of error. (See "Error Branching" later in the description of this verb.)

label The label to which the program is to branch when an arithmetic error is encountered.

item-name The point to which the list register is to be reset before branching to the error label. If you do not specify an *item-name* (for example, ERROR=*label*() or ERROR=*label*), the list register is reset to empty. If you specify an asterisk (for example, ERROR=*label*(*)), the list register is not changed.

The order of precedence for arithmetic operators is:

**	exponentiation
//	division giving remainder
/	division giving quotient
*	multiplication
−	subtraction
+	addition

You can change the order of evaluation by using square brackets. For example, the following two statements may yield different results:

```
LET (A)=(B) + (C)/(D);
LET (A)=[(B) + (C)]/(D);
```

Invalid Values Cause Differing Results

If a problem occurs within a LET statement, the results obtained with Transact/V and Transact/iX may differ. Transact/V will try to substitute a zero for the invalid value and continue with the assignment. Transact/iX will leave the existing value intact and not re-assign the target. This occurs under the following situations:

- Unsubscripted source array is shorter than unsubscripted target array.
- Item is invalid.
- Subscript for a source array is invalid.
- Argument for a function is invalid.

To avoid the arithmetic differences, use the same number of elements for unsubscripted arrays, and do not use X or U data types in an arithmetic expression.

Functions

The following sections describe the functions available within the LET verb, including parameters and examples. These functions can be used whenever an expression can be used. An additional set of parentheses around item parameters is optional. For example, SQRT(Z) and SQRT((Z)) are both acceptable.

A function cannot be embedded or nested in another function. In the following example, the compiler will treat LOG as an array item and generate a warning if LOG is not defined.

```
LET (A) = SQRT(LOG(100.0));
```

The ERROR= option causes the branch to a label to be taken when specific errors occur while processing a function just as specific errors in a LET statement without a function cause such a branch to be taken.

ASCII

The ASCII function converts the first character of a string to the number for its ASCII code. The result will be a number between 0 and 255 inclusive. This function is only valid for string constants and data items of type X or U.

Syntax

$$\texttt{ASCII(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
LET (CODE) = ASCII("A");

                        Before       After
     CODE         I(5)  0            65

LET (CODE) = ASCII((ARRAY(2)));

                        Before       After
     ARRAY(2)     X(4)  BCDE         BCDE
     CODE         I(5)  123          66
```

LENGTH

The LENGTH function returns the length in characters of a string by returning the integer index of the position of the last non-blank character in the string. Embedded blanks are included in this count, but trailing blanks are not included. Nulls are considered valid characters and are counted.

When calculating the length of an X or U item, the maximum length will be the display length. This function is only valid for string constants and data items of type X or U.

Syntax

$$\texttt{LENGTH(}\left\{\begin{array}{l}\texttt{(}\mathit{item\text{-}name}\left[\texttt{(}\mathit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\mathit{character\text{-}string}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
LET (COUNT) = LENGTH("␣␣APPLE␣");

                         Before       After
     COUNT        I(5)   0            7

LET (COUNT) = LENGTH((ARRAY(2)));

                         Before       After
     ARRAY(2)     X(7)   ABC␣DE␣      ABC␣DE␣
     COUNT        I(5)   0            6

LET (COUNT) = LENGTH("␣␣␣␣");

                         Before       After
     COUNT        I(5)   0            0
```

LN

The LN function computes the natural logarithm of a number.

Note

Previously, an additional set of parentheses was not allowed around an item parameter in this function. This has been changed so that additional parentheses around an item are optional. For example, LN(A) and LN((A)) are both acceptable.

Syntax

$$\texttt{LN(}\left\{\begin{array}{l}\texttt{(}\mathit{item\text{-}name}\left[\texttt{(}\mathit{subscript}\texttt{)}\right]\texttt{)}\\ \mathit{numeric\text{-}constant}\end{array}\right\}\texttt{)}$$

Examples

```
LET (RESULT) = LN(100.0);

                        Before       After
     RESULT    R(6,2,4)  0.00         4.61

LET (RESULT) = LN((ARRAY(2)));

                        Before       After
     ARRAY(2)  R(6,2,4)  10.00        10.00
     RESULT    R(6,2,4)   0.00         2.30
```

Errors

If the value of the parameter is zero or less, an error message is issued to indicate a computational error has occurred.

LOG

The LOG function computes the common logarithm to the base 10 of a number.

Note Previously, an additional set of parentheses was not allowed around an item parameter in this function. This has been changed so that additional parentheses around an item are optional. For example, LOG(A) and LOG((A)) are both acceptable.

Syntax

$$\texttt{LOG(}\left\{\begin{array}{l}\texttt{(}\mathit{item\text{-}name}\left[\texttt{(}\mathit{subscript}\texttt{)}\right]\texttt{)}\\ \mathit{numeric\text{-}constant}\end{array}\right\}\texttt{)}$$

Examples

```
LET (RESULT) = LOG(100.0);

                    Before      After
     RESULT    R(6,2,4)  0.00        2.0

LET (RESULT) = LOG((ARRAY(2)));

                    Before      After
     ARRAY(2)  R(6,2,4)  10.00       10.00
     RESULT    R(6,2,4)   0.00        1.00
```

Errors

If the value of the parameter is zero or less, an error message is issued to indicate a computational error has occurred.

POSITION

The POSITION function returns the integer index of the position of the first occurrence of the second string in the first string. Trailing blanks in both strings are ignored. Hence, a string only consisting of blanks cannot be found.

If no match is found, then 0 is returned. This function is case sensitive (for example, "a" does not match "A").

This function is only valid for string constants and data items of type X or U. The display length will be used when calculating the length of a data it of type X or U.

Syntax

$$\texttt{POSITION(}\left\{\begin{array}{l}\texttt{(}\textit{item-name1}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string1}\texttt{"}\end{array}\right\}\texttt{,}\left\{\begin{array}{l}\texttt{(}\textit{item-name2}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string2}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
LET (INDEX) = POSITION("GOOD␣DOG","Z");

                        Before       After
     INDEX        I(5)  99           0

LET (INDEX) = POSITION((STRING1),"D");

                        Before       After
     STRING1      X(8)  BAD␣DOG␣     BAD␣DOG␣
     INDEX        I(5)  99           3
```

Note In the following example note that the trailing blanks in both arguments are ignored.

```
LET (INDEX) = POSITION((STRING1),(STRING2(4)));

                        Before       After
     STRING1      X(8)  BANANA␣␣     BANANA␣␣
     STRING2(4)   X(4)  NA␣␣         NA␣␣
     INDEX        I(5)  99           3
```

SQRT

The SQRT function computes the square root of a number.

Note

Previously, an additional set of parentheses was not allowed around an item parameter in this function. This has been changed so that additional parentheses around an item are optional. For example, SQRT(A) and SQRT((A)) are both acceptable.

Syntax

$$\texttt{SQRT(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \textit{numeric-constant}\end{array}\right\}\texttt{)}$$

Examples

```
LET (RESULT) = SQRT(100.0);

                         Before       After
     RESULT    R(6,2,4)  0.00         10.00

LET (RESULT) = SQRT((ARRAY(2)));

                         Before       After
     ARRAY(2)      I(5)  64           64
     RESULT        I(5)   0            8
```

Errors

If the value of the parameter is less than zero, an error message is issued to indicate a computational error has occurred.

VALUE

The VALUE function returns the numerical value of a string containing the character representation of an integer or a floating point number. Leading blanks are ignored. An initial plus or minus sign is allowed. The number is then terminated by one of the following: (1) the first character that would not be valid in the number; (2) the end of the defined length of the item; or (3) a delimiter defined via the SET(DELIMITER) verb.

With Native Language Support, Transact validates numeric data using the thousands and decimal indicators of the language in effect. (See Appendix E, "Native Language Support," for more information.) If a number is not represented in the string, then 0 is returned. Scientific notation (type E) is not parsed in the string.

When searching through an item, the last character searched depends upon the data type. For an X or U item, the display length is used to get the last character. For an item defined as I, J, Z, P, K, R, or 9, the value function operates in the same way as a LET assignment.

Syntax

$$\texttt{VALUE(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
LET (NUM) = VALUE("-3A");

                        Before       After
     NUM          I(5)  0            -3

LET (NUM) = VALUE("␣␣+43.21ABC");

                        Before       After
     NUM      R(6,2,4)  0.0          43.21

LET (NUM) = VALUE((ARRAY(2)));

                        Before       After
     ARRAY(2)     X(4)  42␣3          42␣3
     NUM          I(5)  0            42

LET (NUM) = VALUE("␣␣A3A");

                        Before       After
     NUM          I(5)  0            0

LET (NUM) = VALUE(".52Time");

                        Before       After
     NUM      R(6,2,4)  0.0          0.52
```

```
LET (NUM) = VALUE(I);
                        Before       After
     NUM          I(5)  0            12345
     I            I(5)  12345        12345

LET (NUM) = VALUE("123-456");

                        Before       After
     NUM          I(5)  0            123
```

Syntax Options

(1) LET (*variable*)=[-]*arithmetic-expression*;

Choose this option to place a single value or the result of an arithmetic operation into a location in the data register *variable* or into one of the Transact-defined names allowed for the destination variable. The following are examples of this syntax option:

```
LET (SUBTOTAL)=(SUBTOTAL) + (AMOUNT);  <<Add values of AMOUNT and SUBTOTAL>>
                                       <<and place result in SUBTOTAL    >>

LET (PERCENT)=9.8;                     <<Set value of PERCENT to 9.8     >>

LET (INVERSE)=1/(DIVISOR);             <<Calculate inverse value         >>


LET (CNT)=-(CNT);                      <<Negate value of CNT             >>

LET (DEDUCTION)=-[(SUBTOTAL)-(BENEFIT)]; <<The result of subtracting     >>
                                         <<BENEFIT from SUBTOTAL is      >>
                                         <<negated and placed in DEDUCTION>>

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>>

  << Set UNIT-PRICE, but if an arithmetic error occurs, branch  >>
  << to CALC-ERROR label and reset list register at UNIT-PRICE. >>

LET (UNIT-PRICE) = (SUBTOTAL-PRICE)/(QUANTITY),ERROR=CALC-ERROR(UNIT-PRICE);
```

Note The LET verb is primarily used to perform arithmetic operations on numeric items. No error is generated if a character (X or U type) item is used and processing continues for that character type, but the results may not be as expected. (Use MOVE to handle character items.)

When LET is used with character items, be aware that the display length is used to determine the size of the item. If the destination item is defined with a display length equal to or larger than the source, the entire source is placed in the destination. If the destination is defined with a display length smaller than the source, the source value is truncated on the right when placed in the destination. The following example demonstrates how different display lengths affect the result.

```
SYSTEM T6100;

DEFINE(ITEM) SMALL   X(5,,6):
             LARGE   X(6,,6);

LIST SMALL:
     LARGE;

  <<LET uses the display length as the size of the item>>

MOVE (SMALL) = "12345";          <<Small has "12345 " in storage      >>
DISPLAY SMALL;                   <<Small displays "12345"             >>

LET (SMALL) = -(SMALL);          <<Small has "-12345" in storage      >>
DISPLAY SMALL;                   <<Small displays "-1234"             >>

LET (LARGE) = (SMALL);           <<Large has "-1234 " in storage      >>
DISPLAY SMALL: LARGE;            <<Both display "-1234"               >>

EXIT;
```

(2) LET OFFSET(*item-name*)=[-]*arithmetic-expression*

(*item-name*) Identifies an ordinary data item or a child item.

[-]*arithmetic - expression* Is as defined earlier for the LET verb, except that in this context the variables may not be subscripted.

This option of the LET verb sets the value of OFFSET for a particular item. It allows you to refer to a child item within a parent item by telling Transact the byte location at which the item begins.

Note It is strongly recommended that you address array items by using subscripts. This discussion is included for those dealing with older versions of Transact programs written before subscripting of arrays was implemented. In any case, the LET OFFSET and subscripting should not be used together. Doing so may cause the program to update the data registers in areas outside the limits of the item referenced and could lead to unpredictable results. Since this was previously the only way to manipulate arrays, no error will be generated. (See "Array Subscripts" in Chapter 3.)

By changing the value of OFFSET, you can 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 can do it as follows (where *element-size* is expressed in bytes):

LET OFFSET(YEAR)=(*element-number* - 1) * *element-size*

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 complex arrays. Consider the complex array of sales figures shown in Figure 8-1. Its compound items are district, year, and month. Each cell, which is a child item, contains a sales figure in integer format. Note that each value in each cell requires four bytes of storage.

This SALES matrix requires the following DEFINE(ITEM) statement:

```
DEFINE(ITEM) SALES-ARRAY X(144):
             DIST 2 X(72)  = SALES-ARRAY:
             YEAR 3 X(24)  = DIST:
             MONTH 12 X(2) = YEAR:
             SALES I(4,,2) = MONTH;
```

The fifth line of the DEFINE statement above redefines MONTH as SALES to further identify the data being stored.

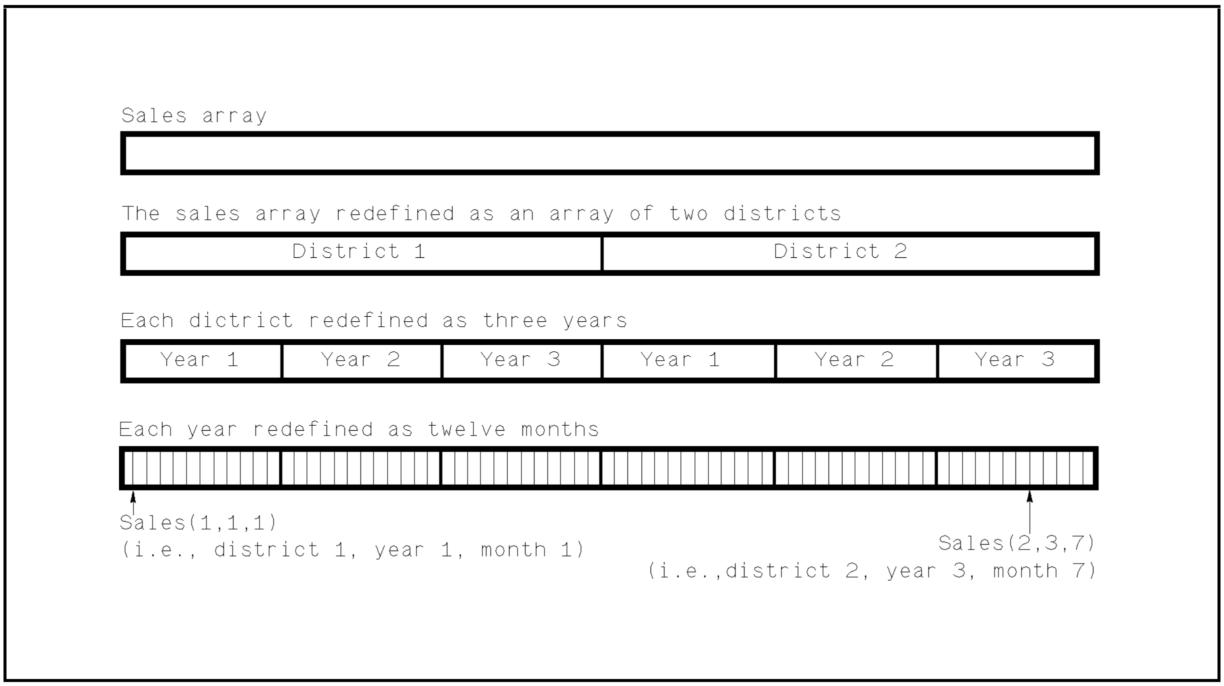


Figure 8-1. Complex Array of Sales Figures.

To locate the position of one SALES element within the array, 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)  = OFFSET(DIST)  + 1 * 72;
LET OFFSET(YEAR)  = OFFSET(YEAR)  + 2 * 24;
LET OFFSET(MONTH) = OFFSET(MONTH) + 6 * 2;
```

Since OFFSET leaves the pointer at the last position referenced, it is necessary to either reset the pointer before further manipulation or plan the next OFFSET in terms of the current position. The following statements reset all offsets to zero, representing the position SALES(1,1,1).

```
LET OFFSET(DIST)  = 0;
LET OFFSET(YEAR)  = 0;
LET OFFSET(MONTH) = 0;
```

When assigning a value to an array, LET assigns each element in the array to that value. If a subscript is specified, then only that element is assigned the value. All other elements remain unchanged.

For example, ARRAY-A is defined as 4X(2), and ARRAY-B is defined as 4I(5,,2).

```
MOVE (TEMP-X) = "ND";
LET (ARRAY-A) = (TEMP-X);        <<Sets all elements in ARRAY-A        >>

DISPLAY ARRAY-A;
MOVE (TEMP-X) = "YR";
LET (ARRAY-A(2)) = (TEMP-X);     <<Sets second elements only in ARRAY-A>>
DISPLAY ARRAY-A;

LET (ARRAY-B) = 67;              <<Sets all elements in ARRAY-B        >>
DISPLAY ARRAY-B;
LET (ARRAY-B(3)) = 78;           <<Sets third element in ARRAY-B       >>
DISPLAY ARRAY-B;
```

Rounding

To determine how rounding is done in Transact, it is necessary to understand how Transact performs arithmetic operations. In general, if you want arithmetic results to be rounded instead of truncated to a desired precision, you should ensure that the operands, including constants, have at least one more digit of precision than the desired result.

Transact uses one of three different methods to process arithmetic expressions. The three methods are:

- double integer arithmetic
- long real arithmetic
- packed decimal arithmetic

The first two methods, double integer and long real arithmetic, are used only if the values meet particular criteria. When these criteria are not met, the third method, packed decimal, is used by default. Since packed decimal arithmetic is slower than the other two methods, it is advisable to use variables that meet the criteria for one of the other two methods whenever possible.

The factors that determine the method to be used are:

- Whether the expression consists of a single operation or multiple operations.
- Data types of the destination variable and the operands.
- The number of decimal places defined for the destination variable and the operands.
- Storage length of the destination variable and the operands.

32-Bit Integer Arithmetic

To qualify for 32-bit integer arithmetic, an expression must meet all of the following conditions:

- The expression must consist of only one operation and that operation can only be +, -, =, or unary minus.
- The data types of the destination variable and the operands must be either I or J. Numeric constants cannot be used.
- The number of decimal places must be identical in target item and both operands.
- Storage length of destination variable and all operands must be 16-bits.

When 32-bit integer arithmetic is used, the target item and the operands are converted to 32-bit integers before the operation is performed. The final result is rounded to the precision of the destination variable and then converted back to a 16-bit integer. Although the operands are converted to 32-bit integers before computation, the final result for 16-bit integers should lie between -32768 and 32767.

The following is an example of 32-bit integer arithmetic:

```
SYSTEM ARIT02;
   DEFINE(ITEM) I1   I(4,1):
                I2   I(4,1):
                I3   I(4,1);
   LIST I1:
        I2:
        I3;

   LET (I1) = 45.99;            << Packed decimal arithmetic         >>
   LET (I2) = 35.99;            << Packed decimal arithmetic         >>
   LET (I3) = (I1) + (I2);      << Double integer arithmetic         >>
   DISPLAY;
   EXIT;
```

When the program is run, the values displayed are:

```
I1 = 46.0
I2 = 36.0
I3 = 82.0
```

64-Bit Real Arithmetic

To qualify for the 64-bit (long) real method of operation, the operands must meet all of the following conditions:

- The expression must consist of a single operation which must be +, -, *, /, //, =, unary minus, LN, LOG, SQRT, or **.
- The destination variable and the operands must all be of data type R or E. Numeric constants can be used; they are converted to the type of the destination variable.
- The storage length of all three variables should be 32-bit or 64-bit.

For 64-bit real arithmetic, if the destination variable and the operands are not already 64-bit real, they are converted to 64-bit real before the operation is performed. The final result is converted back to the size of the destination variable. The internal value of the final item may carry more precision than its defined decimal count. Hence, for subsequent 64-bit real arithmetic, the internal value carrying more precision will be used. On the other hand, for subsequent packed decimal arithmetic (see the following discussion), the internal value will be rounded according to the defined precision and then will be used. The internal value will be rounded up for DISPLAY statements.

For example,

```
SYSTEM LONGRL;

   DEFINE(ITEM) REAL1     R(8):   << No decimal place.                  >>
                REAL2     R(8,2):
                REAL3     R(8,2);
   LIST     REAL1:
            REAL2:
            REAL3;

   LET  (REAL1) = 1440 / 900;     << 64-Bit Real                        >>
```

The display value of REAL1 is 2 (rounded). Internally the value is 1.5555553436279 L+00.. In subsequent 64-bit real arithmetic, the internal value of REAL1 1.555555..... will be used.

```
   LET  (REAL2) = (REAL1) + (REAL1);
```

In subsequent packed decimal arithmetic, REAL1 and REAL2 will be rounded before computation.

```
   LET  (REAL3) = (REAL1) * (REAL2) / 3.11;

   DISPLAY REAL1: REAL2: REAL3;
```

The values displayed are as follows:

```
   REAL1 = 2
   REAL2 = 3.11
   REAL3 = 2.00
```

Packed Decimal Arithmetic

If the values in an expression do not meet the criteria for processing by either the 32-bit integer or 64-bit real methods, then the packed decimal method is used.

In this approach, an arithmetic expression is processed according to the rules of precedence described earlier.

Before computation, the data types of the destination variable and operands are converted to P - - packed P(27,0,14) - - if the operation is `+, -, *, /, //, =,` or unary minus. For the remaining functions, such as SQRT, LOG, LN, exponentiation, and so on, the operands and destination variable are converted to 64-bit real. If this function is an intermediate operation, the result is converted to data type P and stacked for continuing with the rest of the expression. Any operands of type R or E that carry greater precision due to previous long real arithmetic are rounded according to the precision defined for packed decimal arithmetic.

While an expression is being evaluated according to the rules of precedence, each intermediate computational result is computed to the highest precision of the two operands and the destination item. If the precision of the expression is greater than the precision of the destination item, the result is rounded to the precision of the destination item. For example, 3.0/2.0 would produce 1.5 as an intermediate result, which would round to 2 if stored in a receiving item with no decimal places. Unlike COBOL, Transact does not maintain extra precision just for rounding. Thus some division operations may result in a loss of precision. For example, 3/2 produces 1 instead of 1.5 for an intermediate result if the destination variable has no decimal precision.

To ensure that precision is not lost, either the receiving item must have the desired precision (at least one decimal place greater than in the arithmetic expression) or all operands in the entire expression must have the desired precision. For applications that require the destination variable to have fewer or zero decimal places, a two-step arithmetic sequence is recommended. The destination variable of the first LET should have an adequate number of decimal places for processing the whole expression and then the second LET statement should contain a simple assignment (=) to an integer item having fewer or zero decimal places. The following example shows this technique.

```
SYSTEM PAKDEC;
   DEFINE(ITEM) R1 R(6):         << No decimal place                >>
                R2 R(11,5):      << More decimal places             >>
                I3 I(9,2);       << Fewer decimal places            >>
```

Example 1

The LET statement below uses the destination variable R1, which has no decimal places. Compare the final results with the next LET statement.

```
LET (R1) = 11590.0000 * [[6353.6100 / 6354] * [1440/900]];

                             |_________________|
                                   .9999
                                                    |________|
                                                         1
                                      |_________________|
                                             .9999
                 |___________________________|
                         11588.8410
       |_______________________|
           11589 (ROUNDED)
```

Example 2

The LET statement below uses the destination variable R2, which has five decimal places.

```
LET (R2) = 11590.0000 * [[6353.6100 / 6354] * [1440/900]];
                             |_________________|
                                   .99993
                                                    |________|
                                                     1.60000
                                    |____________________|
                                           1.59989
                 |___________________________|
                         18542.72510
      |______________________|
           18542.72510
```

The LET statement below does a simple assignment to the item I3, which has two decimal places. The result is rounded. This is a packed decimal operation, since data types are different. The internal value of I3 does not carry extra precision.

```
LET (I3) = (R2);

      |________|
       18542.73 (ROUNDED)
```

The next two examples show the effect of using either 2 or 5 digits of decimal precision on the first example above.

Example 3

Suppose the following was included in the source file:

```
!PRECISION(2)
```

The original LET statement assigning a value to item R1 (Example 1) is now evaluated as shown below. The major difference from the original evaluation is that 1440/900 is treated as 1.60 rather than 1 which affects all subsequent intermediate results.

```
LET (R1) = 11590.0000 * [[6353.6100 / 6354] * [1440/900]];
                         |_______________|
                               .9999
                                            |________|
                                               1.60
                                 |________________|
                                      1.5998
                 |_________________________|
                            18541.6820
      |______________________|
                18542 (ROUNDED)
```

Example 4

Now replace the !PRECISION statement above with the following statement:

```
!PRECISION(5)
```

The original LET statement assigning a value to item R1 (Example 1) is now evaluated as shown below. This LET statement is evaluated the same as in Example 2, except that here it is rounded for the final assignment.

```
LET (R1) = 11590.0000 * [[6353.6100 / 6354] * [1440/900]];
                         |_______________|
                               .99993
                                            |________|
                                              1.60000
                                 |__________________|
                                      1.59989
                 |_________________________|
                           18542.72510
      |______________________|
                18543 (ROUNDED)
```

Note

The result of the LET statement assigning a value to item R2 in Example 2 is not altered by changing the minimum decimal precision to either 2 or 5. Since the destination variable is defined with 5 decimal digits, all intermediate results automatically maintain at least 5 digits of decimal precision.

For more information about !PRECISION, see "Compiler Commands" in Chapter 9.

LEVEL

Defines processing levels within a program.

Syntax

`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. The data register, key register, match register, list register, update register, SET(DELIMITER) and SET(OPTION) are unique to that level. When an end of level occurs, these registers and settings are reset to the condition they were in on entering the level.

Statement Parts

label The statement to which the program should branch at the end of the level sequence if the user enters "]" in reply to a program prompt.

item-name The location in the list register where the pointer is to be set.

If you do not specify *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*(*));, the list register is reset to the condition it was in on entering the level.

Exits From LEVEL Sequences

If no label is specified, four types of exits from LEVEL sequences are possible;. two of which the user controls and two of which the programmer controls. They are described below.

] When the user enters "]" in response to any prompt in a level sequence, control passes to the previous processing level, which may be the command level or to the label specified in LEVEL(*label*). Any changes made to the match, list, or update registers within the level are reset to their original state.

]] When the user enters "]]" in response to any prompt in a level sequence, control passes to the Transact command level, or if not in a command sequence, Transact issues the EXIT or RESTART(E/R)> prompt.

END(LEVEL) The end of the current level. This causes control to fall through to the statement following the END(LEVEL) statement and resets the match, list, or update registers to whatever their conditions were immediately before the last level sequence began.

END If you use END without (LEVEL) to terminate a level, Transact generates a loop after the first execution of the level. The loop begins at the top of the level. The match, list, or update registers are reset to whatever their values were at the beginning of the level.

Examples

Nested level sequences are possible, as this example shows. The following statements minimize the data entry required for a sequence of entries for a time card. It requires values for year and month, then multiple entries for employee. Each level change provides the opportunity for the user to enter data.

```
PROMPT YEAR:
       MONTH;
LEVEL;
  PROMPT EMPLOYEE;
  LEVEL;
    PROMPT DAY;
    LEVEL;
      PROMPT ACTIVITY:            <<A loop through this level resets    >>
             HOURS;               <<list REGISTER and data register for >>
      PUT TIME-RECORD;            <<these data items (activity, hours). >>
    END;
  END;
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 user is prompted with "DAY". When all days have been entered for one employee, the user should enter "]" in response to "DAY". Control passes to the next higher level and the user is then prompted for the next employee.

LIST

Adds item names to list, key, match, and/or update registers.

Syntax

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 register affected depends on the verb modifier. You can choose from the following:

none	Adds specified item name to list register, reserves space, and, optionally, places value in data register. (See Syntax Option 1.)
AUTO	Adds the names of all items in a dictionary associated with the specified file to the list register or adds all items defined in the program plus all items resolved from the dictionary to the list register. (See Syntax Option 2.)
KEY	Places specified item name in key register. (See Syntax Option 3.)
MATCH	Adds specified item name to list register and copies existing value for that item from the data register to the match register. (See Syntax Option 4.)
PATH	Adds specified item name to list register and places it in key register. (See Syntax Option 5.)
UPDATE	Adds specified item name to list register and copies value for that item from the data register to the update register. (See Syntax Option 6.)

Consider the following when setting up your list register:

- For use with database access, list items may be in any position in the register. However, consecutive order allows simpler range lists in the data management statements.
- For use with KSAM or MPE files or VPLUS forms, list items can be in any position in the register. However, with the LIST= option, the items *must* be specified in the same order as the items occur in the physical file or form.
- Child item names cannot 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.
- See Chapter 4, "Transact Registers," for a discussion of adding items to the LIST register multiple times.

Statement Parts

modifier A 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. The formats of these options are not affected by the choice of language in the SET(LANGUAGE) statement.

ACCOUNT An X(8) item that contains the account name from the system log on.

ALIGN Forces the item to be aligned on a 16-bit boundary on Transact/V and on a 32-bit boundary on Transact/iX.

Note Only compile time alignment is supported.

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/C An X(8) item that contains the current system date in YYYYMMDD format.

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 system log on.

HOME-GROUP 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 system log on.
PROCTIME	An I(9) item that contains the 32-bit integer of process 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 32-bit 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 the system 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(AUTO) {*file-name*[,*option-list*];
{ @[,*option-list*];

LIST(AUTO) *file-name* adds the names of all the items in the specified file to the list register. *file-name* can refer to a form, a file, or a data set, but not a database or forms file. Transact uses the dictionary to acquire the item names, and a compiler error results if *file-name* is not defined in a dictionary or if it has no item names associated with it. Alias definitions are not retrieved from the dictionary.

The option INIT sets blanks to the data item if its type is an alphanumeric string or sets binary zero to the data item for all other data types. When the DEFN option is used during program compilation, all item names in the specified file will be included in the compile listing and it will give the name and relative list register position of each item.

The option ALIGN forces the item to be aligned on a 16-bit boundary on Transact/V and on a 32-bit boundary on Transact/iX. The first item with LIST(AUTO) *filename*,ALIGN will be aligned.

LIST(AUTO) @ causes Transact to place all the user-defined data items in the program into the list register in the order in which they are encountered during compilation. This includes items resolved from the dictionary. The option INIT sets blanks to the data item if its type is an alphanumeric string or sets binary zero to the data item for all other data types. All items with LIST(AUTO)@,ALIGN will be aligned.

When multiple LIST(AUTO) statements are issued for different files that have some items in common, you must ensure that the resultant structure of the list register will support the statements that follow.

(3) LIST(KEY) *item-name*;

LIST(KEY) places *item-name* in the key register only.

(4) 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. Matching with alphanumeric data is affected by the native language set by a SET(LANGUAGE) statement. For more information, see Appendix E, "Native Language Support."

The following values for *option-list* specify a match selection to be performed on a basis other than equality.

option-list: Any of the following options can be selected:

ALIGN	Forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX
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) LIST(PATH) *item-name*[,*option-list*];

LIST(PATH) adds *item-name* to the list register and places it in the key register.

The ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

(6) LIST(UPDATE) *item-name*[,*option-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.

The ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

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; it is only mapped and remapped through the list register. To illustrate this point, consider the following example that references two databases. 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 database         >>

END TEST1;
```

Note that the list register was reset programmatically with the RESET(STACK) statement.

The next example shows the use of LIST(AUTO) to include all defined items in the list register and initialize them.

```
LIST(AUTO) @,INIT;
```

The next example is used to put dictionary items for a file in the list register.

```
LIST(AUTO) PASSENGER-DTL;
```

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;
```

The next example shows how the DATE/C option is used.

```
SYSTEM DATES;

  DEFINE(ITEM) TODAYS-DATE  X(8):
               TODAYS-YEAR  X(4)   = TODAYS-DATE(1):
               TODAYS-MONTH X(2)   = TODAYS-DATE(5):
               TODAYS-DAY   X(2)   = TODAYS-DATE(7);

  LIST TODAYS-DATE, DATE/C;

  DISPLAY "TODAY'S DATE:
":TODAYS-DATE,NOHEAD,EDIT="^^^^/^^/^^";

  DISPLAY "FORMATTED DATE: ", LINE=2:
          TODAYS-MONTH, NOHEAD, SPACE=0:
          "/", SPACE=0:
          TODAYS-DAY,   NOHEAD, SPACE=0:
          "/", SPACE=0:
          TODAYS-YEAR,. NOHEAD, SPACE=0;
  EXIT;
```

The output from this example is:

```
TODAY'S DATE: 1992/08/18

FORMATTED DATE: 08/18/1992
```

The last example shows how the ALIGN option causes *item1* and *item3* to be word-aligned in the list register. *Item2* will follow *item1* and may or may not be aligned, depending on length of *item1*.

```
LIST item1,ALIGN;
     item2;
     item3,INIT,ALIGN;
```

LOGTRAN

Makes the database calls needed to maintain the database log files and optionally performs database transaction locking.

Syntax

LOGTRAN(*modifier*) *base*, *log-message*[,*option-list*];

LOGTRAN is used to define a static or dynamic logical transaction for database transaction logging or locking purposes. If this verb is to be used for database logging and recovery, several steps must first be completed before the statement can be used. If this verb is to be used for transaction locking, no preliminary steps need to be taken. See the discussions of transaction logging in the TurboIMAGE reference manuals for more information regarding static and dynamic transactions.

Transact file access verbs lock at the start of execution for a statement and unlock before the next statement. Therefore, other processes can modify the data during a logical transaction covered by LOGTRAN if the transaction comprises more than one statement. It is therefore always advisable to lock the transaction being logged.

If LOGTRAN is used for locking, it should be used consistently throughout all programs, and databases and data sets should be locked and unlocked in the same order by all programs. LOGTRAN locking should not be mixed with Transact's automatic locking. Automatic locks should be disabled by SET(OPTION) NOLOCK, and automatic error handling should be disabled by specifying the STATUS option. Multiple LOGTRAN locks can only be issued on different data sets in a database with an intervening LOGTRAN(END) or LOGTRAN(XEND) verb on that database. See "Database and File Locking" in Chapter 6 for more information.

Statement Parts

modifier Specifies the type of operation.

BEGIN Starts a static transaction and writes a record to the log file if user logging is enabled.

Optionally, BEGIN locks the data sets specified in SET(LIST). The LOCK option should be specified unless the PROC statement is used for locking.

The LOGTRAN(BEGIN) statement must always be paired with a LOGTRAN(END) statement to mark the beginning and end of a static transaction for a given database. No other LOGTRAN(BEGIN) or LOGTRAN(END) statement referencing the same database access path can appear between a pair of LOGTRAN(BEGIN) and LOGTRAN(END) statements.

MEMO Writes a log record in the log file to provide more information about the logical transaction if user logging is enabled.

END Ends a static transaction and writes a record to the log file if user logging is enabled.

Unlocks the database locked by its corresponding LOGTRAN(BEGIN) statement.

The LOGTRAN(END) statement must always be preceded by a LOGTRAN(BEGIN) statement. No other LOGTRAN(BEGIN) or LOGTRAN(END) statement referencing the same database access path can appear between a pair of LOGTRAN(BEGIN) and LOGTRAN(END) statements.

Note

The following modifiers, XBEGIN, XEND, and XUNDO, apply to Transact/iX only. They support the TurboIMAGE/XL dynamic roll-back feature that provides MPE/iX transaction management logging.

XBEGIN
: Starts a dynamic transaction and writes a record to the log file if user logging is enabled.

Optionally, XBEGIN locks the data sets specified in SETLIST. The lock option should be specified unless the PROC statement is used for locking.

Nesting of dynamic or static transactions within a dynamic transaction is not allowed when using the same database access path. The LOGTRAN(XBEGIN) statement must always be paired with a LOGTRAN(XEND) statement to mark the beginning and end of a dynamic transaction. No other LOGTRAN(BEGIN), LOGTRAN(END), LOGTRAN(XBEGIN), or LOGTRAN(XEND) statement can appear between a matching pair of LOGTRAN(XBEGIN) and LOGTRAN(XEND) statements for a specific database access path.

XEND
: Ends a dynamic transaction and writes a record to the log file if user logging is enabled.

Unlocks the database locked by its corresponding LOGTRAN(XBEGIN) statement. The LOGTRAN(XEND) statement must always be preceded by a LOGTRAN(XBEGIN) statement to mark the beginning and end of a dynamic transaction. No other LOGTRAN(BEGIN), LOGTRAN(END), LOGTRAN(XBEGIN), or LOGTRAN(XEND) statement can appear between a matching pair of LOGTRAN(XBEGIN) or LOGTRAN(XEND) statements for a specific database access path. Also, LOGTRAN(XEND) cannot be called after a call has been made to LOGTRAN(XUNDO).

XUNDO
: Rolls back the modifications associated with a dynamic transaction and writes a record to the log file if user logging is enabled.

Unlocks the database locked by its corresponding LOGTRAN(XBEGIN) statement.

The LOGTRAN(XUNDO) statement must always be preceded by a LOGTRAN(XBEGIN) statement to mark the beginning of a dynamic transaction. LOGTRAN(XUNDO) cannot be called to roll back a transaction started by a LOGTRAN(BEGIN) statement. Also, LOGTRAN(XUNDO) cannot be called after a call has been made to LOGTRAN(XEND) for that specific database access path.

base The database to be logged. It must be one of the following:

$HOME This special name indicates that the home database is to be logged.

Note Using the actual home base name in the LOGTRAN statement causes a compiler error.

base-name The name of the database to be logged (when the database is other than the home base).

log-message The *log-message* parameter is required for all LOGTRAN verbs. It must be one of the following:

(*item-name* [(*subscript*)]) The name of a data item that contains the text string (up to 512 bytes long) to be written to the log file. This item must begin on a 16-bit word boundary. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

"*message-string*" The text string (up to 512 bytes long) to be written on the log file.

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

LOCK(*setlist*) This option causes the data sets specified in the *setlist* to be locked. This option is only valid with the LOGTRAN(BEGIN) or LOGTRAN(XBEGIN) statements, with the locks remaining in effect until the corresponding LOGTRAN(END), LOGTRAN(XEND) or LOGTRAN(XUNDO) is encountered.

The *setlist* is of the form:

(*setname*[*cond*][,*setname*[*cond*] ...)

setname The name of the data set to be locked. If the entire database is to be locked the user can substitute @ for *setname*.

cond The lock condition, either COND for conditional lock or UNCOND for unconditional locking. COND is the default.

Note When locking multiple data sets, Multiple Rin (MR) capability must be in effect. You should also list data sets in the order in which they appear in the database for added compatibility with non-Transact Applications. (See the *TurboIMAGE/XL Database Management System Reference Manual* for more information.)

NOMSG — Suppresses the standard error message produced as a result of a database error. It is recommended that STATUS is used with this option.

STATUS — Suppresses the actions defined in Chapter 7 under "Automatic Error Handling." You will need to add code to check the value of STATUS. When STATUS is specified, the effect of a LOGTRAN statement is described by the 32-bit integer value in the status register:

Status Register Value	Meaning
0	The LOGTRAN operation was successful.
<> 0	This is the database error code. (See the *TurboIMAGE/XL Database Management System Reference Manual*.)

See "Using the STATUS Option" in Chapter 7 for more information.

Examples

The first example begins a transaction and locks the entire PERSON database conditionally.

```
LOGTRAN(BEGIN) PERSON, "BEGIN LOGGING DATABASE", LOCK(@);
```

This example begins a transaction, locks the data set NAME unconditionally, and locks the data set ADDRESS unconditionally.

```
LOGTRAN(BEGIN) $HOME, (MSG), LOCK(NAME(UNCOND), ADDRESS(UNCOND));
```

This example begins a transaction and locks the home base conditionally.

```
LOGTRAN(BEGIN) $HOME, (MSG), LOCK(@(COND));
```

This example ends a transaction and unlocks any data sets in $HOME that have been locked.

```
LOGTRAN(END) $HOME, (MSG);
```

This example begins a dynamic transaction and locks the entire home database conditionally.

```
LOGTRAN(XBEGIN) $HOME, "BEGIN DYNAMIC TXN LOGGING", LOCK(@);
```

The next example begins a dynamic transaction for the PEOPLE database and locks the NAME data set conditionally and the ADDRESS data set unconditionally.

```
LOGTRAN(XBEGIN) PEOPLE, (MSG), LOCK(NAME, ADDRESS(UNCOND));
```

This example shows how to begin a dynamic transaction with programmer's control of locking. This would be done if the Transact locking scheme for LOGTRAN(XBEGIN) was not adequate.

```
SET(OPTION) NOLOCK;
LET (MODE) = 1;
:
:
```

```
:
PROC DBLOCK(BASE(CUSTOMERS),
    SET(NAMES),
    (MODE),
    STATUS(DB));
MOVE (CSTATUS) = STATUS(DB);
IF (CSTATUS) <> 0 THEN
    GO TO LOCK-ERROR;
LOGTRAN(XBEGIN) $HOME, (MSG);
```

This example ends a transaction and unlocks any data sets or database locked by the corresponding LOGTRAN(XBEGIN).

```
LOGTRAN(XEND) $HOME, "END OF DYNAMIC TXN LOGGING";
```

The next example shows how to end a dynamic transaction with programmer's control of locking. It assumes that the LOCK option on LOGTRAN(XBEGIN) was NOT used.

```
SET(OPTION) NOLOCK;
LET (MODE) = 1;
:
:
:
LOGTRAN(XEND) $HOME, (MSG);
PROC DBUNLOCK(BASE(CUSTOMERS),
    SET(NAMES),
    (MODE),
    STATUS(DB));
MOVE (CSTATUS) = STATUS(DB);
IF (CSTATUS) <> 0 THEN
   GO TO UNLOCK-ERROR;
```

The next example shows how to end a dynamic transaction when the contents of the logging buffer in memory should be written to disk (Mode 2 of DBXEND). This would be used for critical transactions. It is assumed that locks are held throughout the transaction and that unlocking is the responsibility of the programmer.

Note The DBXEND call must precede the call to DBUNLOCK or TurboIMAGE will return an error.

```
SET(OPTION) NOLOCK;
LET (MODE) = 2;
     :
     :
PROC DBUNLOCK(BASE(CUSTOMERS),
      (MSG),
```

```
      (MODE),
      STATUS(DB),
      (NUMBYTES));
MOVE (CSTATUS) = STATUS(DB);
IF (CSTATUS) <> 0 THEN
   GO TO DBXEND-ERROR;
```

The last example rolls back a transaction that was previously started by LOGTRAN(XBEGIN).

```
LOGTRAN(XUNDO) EMPLOYEES, (MSG);
```

MOVE

Places data into a specified data register space.

Syntax

MOVE (*destination-variable*) = *source-expression*;

MOVE places data into the data register location specified by *destination-variable*. 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 convert data types between the source and the destination, you must use the LET verb to do so. Since MOVE does not check data types during the operation, a destination-variable of type U could contain lowercase alphanumeric characters.

When moving items of different lengths, values are truncated or filled on the right. Numeric data types I, J, Z, P, K, R, E, and 9 are filled with nulls, and alphanumeric data types X and U are filled with blanks.

Note In Transact/iX the fill character for data type 9 is blank.

The display length of the source data item is used to determine the number of characters moved for data types U and X. Storage length is used for all data types when using justification of a literal. For unsubscripted arrays using justification, storage length is used for all data types.

Note The *destination-variable* is used to hold any intermediate results when processing the *source-expression*. See "Special Considerations" later in this verb for potential side effects.

Statement Parts

destination-variable can be the following:

(*item-name* [(*subscript*)])	Specifies that you want the data moved into the data register location identified by *item-name*. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

source-expression is defined below with detailed explanations following:

$$\left\{\begin{array}{l} [-]\,(\textit{item-name}[\,(\textit{subscript})\,]) \\ [-]\,\texttt{"}\textit{character-string}\texttt{"} \\ [-]\,\textit{string-function} \\ \textit{format-function} \\ \textit{source1}\ [\textit{operator source2}]\ \ldots \\ \texttt{STATUS}(\textit{parm}) \end{array}\right\}$$

[−](*item-name* [(*subscript*)])	The value in the data register location for *item-name*. If you include the minus sign (−), 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. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)
[−]"*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 field with blanks, use a space, `" "`, for the character string.

[−]*string-function*	Any of the functions listed below, each of which has a character string as its result. If you include the minus sign (−), 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. (See "Functions" later in the description of this verb for a description of each function and its parameters.)
	
	`CHAR`
	`LOWER`
	`PROPER`
	`STRING`
	`UPPER`

format-function	Either of the two functions listed below, each of which operates on the destination field. A minus sign is not allowed before a format-function. (See "Functions" later in the description of this verb for a description of each function and its parameters.)
	
	`COL`
	`SPACE`

source1 operator source2 ... operator sourcen	The *source* can be an (*item-name*[(*subscript*)]), a "*character-string*", a *string-function*, or a *format-function*. The operator can be +, −, and both operators can be used in the same expression (a minus sign is not allowed before a *format-function*). The plus sign concatenates the items and strips trailing blanks. The minus sign removes the next item. Items and strings to be combined are specified in the order of their intended concatenation or removal. Leading blanks in the strings to be concatenated are not stripped before concatenation.
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 database call to the data register location specified by *destination-variable*. BASE — Moves the database name referenced in the last database call to the data register location specified by *destination-variable*.

FILE	Moves the name of the data set or file referenced by the last database, KSAM, or MPE call to the data register location specified by *destination-variable*.

Special Considerations

The *destination-variable* on the left of the "=" sign will be used as a temporary variable to hold intermediate values necessary when calculating the result of the *source-expression* on the right of the "=" sign. Some important points should be noted:

- The operands in the *source-expression* are processed in the order referenced (i.e. left to right).
- If the *source-expression* contains multiple operators, the *destination-variable* must be defined large enough to hold any intermediate values or truncation will occur. For example,

  ```
  MOVE (A) = (B) + (C) – (D);
  ```

  (A) must be large enough to hold the result of (B) + (C). Failure to do so will cause the intermediate result to be truncated before removing the value of (D). A detailed example follows:

  ```
  MOVE (NAME) = (FNAME) + (LNAME) - "SON";

                            Before       After
       FNAME       X(05)   DAVID        DAVID
       LNAME       X(06)   BENSON       BENSON
       NAME        X(10)   JEFFBENNER   DAVIDBENSO
  ```

- If the *source-expression* used on the right of the "=" sign also contains the *destination-variable*, the value of the *destination-variable* may have changed which could cause unexpected results. For example,

  ```
  MOVE (A) = (B) + (C) + (A);
  ```

  the reference to (A) on the right will have the result of (B) + (C) in it when it is used in the calculation. The best strategy is to avoid using (A) on the right after two operators. A detailed example follows:

  ```
  MOVE (NAME) = (FNAME) + (LNAME) + (NAME);

                            Before                     After
       FNAME       X(08)   John␣␣␣␣                   John␣␣␣␣
       LNAME       X(08)   Paul␣␣␣␣                   Paul␣␣␣␣
       NAME        X(16)   Jones␣␣␣␣␣␣␣␣␣␣␣           JohnPaulJohnPaul
  ```

- If the *source-expression* contains a child item and the destination-variable is an overlapping child item of the same parent, the destination variable may contain unexpected results. For example:

  ```
  DEFINE(ITEM) PARENT X(5):
               CHILD1 X(3) = PARENT(1):
               CHILD2 X(3) = PARENT(3);
  LIST PARENT,INIT;
  MOVE (PARENT) = "AABBB";
  MOVE (CHILD2) = (CHILD1);
  ```

After the move, CHILD2 contains “AAA”, not “AAB”, because the move is done as follows:

1. The first A is moved from position 1 to position 3.
2. The second A is moved from position 2 to position 4.
3. The third character has already been replaced by step 1, and is now A; the third step is therefore to move A from position 3 to position 5.

- When a MOVE contains more than two operands, the Transact compiler will split the MOVE into multiple MOVE statements of two operands each. The following statement:

  ```
  MOVE (A) = (B) + (C) - (D);
  ```

  will be split into the following:

  ```
  MOVE (A) = (B) + (C);
  MOVE (A) = (A) - (D);
  ```

- When a function is the first operand in a MOVE statement, the MOVE will be split into multiple MOVE statements. Consider the following statement:

  ```
  MOVE (A) = UPPER(B) + (C) - (D);
  ```

  The Transact compiler will split the MOVE into the following statements:

  ```
  MOVE (A) = UPPER(B);
  MOVE (A) = (A) + (C);
  MOVE (A) = (A) - (D);
  ```

Functions

The following sections describe the string functions (CHAR, LOWER, PROPER, STRING, and UPPER) and format functions (COL and SPACE) that are only available within the MOVE verb, including parameters and examples.

The string functions return values based on the operation performed on the source-variable. The format functions operate on the destination-variable.

A leading minus sign (−) is not allowed with a format function. A compiler error will be generated when a minus sign immediately precedes a format function.

CHAR

The CHAR function returns the character equivalent of a numerical ASCII code. The argument is a number between 0 and 255 inclusive. Arguments outside the range of 0 to 255 will return a blank.

Syntax

$$\texttt{CHAR(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \textit{numeric-constant}\end{array}\right\}\texttt{)}$$

Examples

```
MOVE (STRING) = CHAR((NUM));

                          Before       After
     NUM          I(4)    65           65
     STRING       X(4)    XYZ␣         A␣␣␣

MOVE (STRING(2)) = CHAR(97);

                          Before       After
     STRING(2)    U(4)    XYZ␣         a␣␣␣

MOVE (STRING(2)) = -CHAR(97);

                          Before       After
     STRING(2)    U(4)    XYZ␣         ␣␣␣a
```

COL

The COL function moves a string into the destination beginning at the specified column position. The first column position is 1. Any bytes in the destination to the left of the column position will be unchanged.

Syntax

$$\texttt{COL(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\Big[\texttt{(}\textit{subscript}\texttt{)}\Big]\texttt{)}\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{,}\textit{position}\texttt{)}$$

where *position* is either a data item name in parentheses or a numeric constant. The position parameter indicates the byte in the destination where the string will begin. If *position* is greater than the number of bytes in the destination, nothing is moved.

Examples

```
MOVE (ADDRESS) = (NUMBER) + COL((STREET),(POS));

                         Before                   After
     ADDRESS      X(16) abcdefghijklmnop         125␣␣Hardwick␣␣␣
     POS          I(4)  6                        6
     NUMBER       X(4)  125␣                      125␣
     STREET       X(10) Hardwick␣␣                Hardwick␣␣

MOVE (ADDRESS) = COL((STREET),(POS));

                         Before                   After
     ADDRESS      X(16) abcdefghijklmnop         abcdeHardwick␣␣␣
     POS          I(4)  6                        6
     STREET       X(10) Hardwick␣␣                Hardwick␣␣
```

Errors

A position value less than 0 is the only error specific to the COL function. If an error is encountered while processing the COL function, the string will be moved to the destination using the default position value of 1. A message describing the error condition is also displayed. Processing continues if Transact is running online, but will stop if Transact is running in batch mode.

LOWER

The LOWER function returns a string in which all letters are converted to lowercase. Any non-alphabetic characters remain unchanged.

Syntax

$$\texttt{LOWER(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
MOVE (NAME) = LOWER((NAME));

                        Before       After
     NAME         X(8)  BROWN␣J␣    brown␣j␣

MOVE (LNAME) = LOWER("SMITH");

                        Before       After
     LNAME        U(4)  ABCD         smit

MOVE (ACTION(2)) = LOWER((VERB((I)))) + "ed";

                        Before       After
     I            I(5)  4            4
     VERB(4)      X(4)  JUMP         JUMP
     ACTION(2)    X(6)  TURNS␣       jumped
```

PROPER

The PROPER function returns a string in which a letter in the first character position and each letter immediately following a special character are converted to uppercase. All other characters remain unchanged.

The default set of special characters as used by PROPER are `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~` and the blank character.

To change the set of characters that cause the next letter to be upshifted, see the SET and RESET verbs later in this chapter.

Syntax

PROPER({ (*item-name*[(*subscript*)]) | "*character-string*" })

Examples

```
MOVE (NAME) = PROPER((NAME));

                        Before       After
     NAME         X(8)  brown␣j␣     Brown␣J␣

MOVE (LNAME) = PROPER("smith,j");

                        Before       After
     LNAME        U(7)  ABCD␣␣␣      Smith,J

MOVE (LNAME) = PROPER("SMITH,J");

                        Before       After
     LNAME        U(7)  ABCD␣␣␣      SMITH,J

MOVE (LNAME) = PROPER((NAME));

                        Before       After
     NAME         X(5)  smith        smith
     LNAME        X(6)  ABCD␣␣       Smith␣

MOVE (LNAME) = PROPER("mr.john smith (hp)");

                        Before       After
     LNAME       X(18)  ABCD␣...␣    Mr.John␣Smith␣(Hp)
```

```
MOVE (LNAME) = PROPER((NAME));

                        Before       After
     NAME         X(7)  a⊔and⊔b      a⊔and⊔b
     LNAME        X(7)  ABCDE⊔⊔      A⊔And⊔B

MOVE (ACTION(2)) = PROPER((VERB((I)))) + "ed";

                        Before       After
     I            I(5)  3            3
     VERB(3)      X(4)  JUMP         JUMP
     ACTION(2)    X(6)  TURNS⊔       JUMPed

MOVE (LNAME) = PROPER("a1b,c.d!e&f g(h]i;");

                        Before       After
     LNAME       X(18)  ABCD⊔...⊔    A1b,C.D!E&F⊔G(H]I;
```

SPACE

The SPACE function moves the specified number of spaces into the destination before moving the string.

Syntax

$$\mathtt{SPACE(}\left\{\begin{array}{l}(\textit{item-name}\left[(\textit{subscript})\right])\\ \textit{"character-string"}\end{array}\right\}, \textit{space-size})$$

where *space-size* is either a data item name in parentheses or a numeric constant. The space-size parameter indicates the number of spaces to be moved to the destination before moving the string.

Examples

```
MOVE (NAME) = (LNAME) + SPACE(FNAME,1) + SPACE(INITIAL,1);

                    Before              After
   LNAME   X(6)     Doe␣␣␣              Doe␣␣␣
   FNAME   X(6)     John␣␣              John␣␣
   INITIAL X(2)     Q␣                  Q␣
   NAME    X(14)    abcdefghijklmn      Doe␣John␣Q␣␣␣␣
```

Errors

A space-size value less than 0 is the only error specific to the SPACE function. If an error is encountered while processing the SPACE function, the string will be moved to the destination using the default space-size value of 0. A message is also displayed describing the error condition. Processing continues if Transact is running online, but will stop if Transact is running in batch mode.

STRING

The STRING function returns a string that is taken from another string beginning at a given position for a given length.

Syntax

$$\texttt{STRING(}\left\{\begin{array}{l}\texttt{(}\textit{item-name}\left[\texttt{(}\textit{subscript}\texttt{)}\right]\texttt{)}\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{,}\textit{position}\texttt{,}\textit{length}\texttt{)}$$

where *position* and *length* are either data item names in parentheses or numeric constants. The *position* parameter indicates the byte at which the substring begins. The *length* parameter indicates the number of bytes to move. If *length* + *position* would extend beyond the end of the source string, the substring returned will be padded a corresponding number of trailing spaces.

Examples

```
MOVE (NAME) = STRING((NAME),1,3);

                         Before       After
     NAME         X(8)   BROWN␣J␣     BRO␣␣␣␣␣

MOVE (LNAME) = STRING("SMITH",(POS),(LEN));

                         Before       After
     POS          I(4)   3            3
     LEN          I(4)   2            2
     LNAME        X(6)   ABCD␣␣       IT␣␣␣␣

MOVE (LNAME) = STRING((NAME),(POS),4);

                         Before       After
     POS          I(5)   2            2
     NAME         X(5)   SMITH        SMITH
     LNAME        X(6)   ABCD␣␣       MITH␣␣

MOVE (ACTION(2)) = STRING((VERB((I))),(POS(3)),(LEN((I)))) + " ";

                         Before       After
     I            I(5)   4            4
     VERB(4)      X(4)   JUMP         JUMP
     POS(3)       I(4)   1            1
     LEN(4)       I(4)   3            3
     ACTION(2)    X(6)   TURNS␣       JUM␣␣␣
```

The next two examples demonstrate the use of functions with concatenation. Removal can produce different results:

```
MOVE (X10) = "Rapid Team" - ("a",20,1) - "p";

                         Before       After
     X10          X(10)  ABC␣␣␣␣␣␣␣            Raid␣Team␣
```

The string function returns a null, therefore nothing is removed.

```
MOVE (X5)  = STRING ("a",20,1);
MOVE (X10) = "Rapid Team" - (X5) - "p";

                        Before       After
     X5           X(5)  ABCDE        ␣␣␣␣␣
     X10          X(10) XYZ␣␣␣␣␣␣␣             RaidTeam␣␣
```

The string function returns a null, however when a null is moved to an X type item, it is converted to blanks. A blank is then removed in the second MOVE statement.

Errors

If an error is encountered while processing the STRING function, an appropriate default string is returned by the function depending on the destination's data type (spaces for X and U types and nulls for all other types). A message is also displayed describing the error condition. Processing continues if Transact is running online but will stop if Transact is running in batch mode. The only errors specific to the STRING function are:

- Position parameter <0
- Length parameter <0

UPPER

The UPPER function returns a string in which all letters are converted to uppercase. Non-alphabetic characters remain unchanged.

Syntax

$$\texttt{UPPER(}\left\{\begin{array}{l}(\textit{item-name}\left[\,(\textit{subscript})\,\right])\\ \texttt{"}\textit{character-string}\texttt{"}\end{array}\right\}\texttt{)}$$

Examples

```
MOVE (NAME) = UPPER((NAME));

                        Before       After
     NAME         X(8)  brown␣j␣    BROWN␣J␣

MOVE (LNAME) = UPPER("smith");

                        Before       After
     LNAME        U(6)  abcd␣␣       SMITH␣

MOVE (LNAME) = UPPER((NAME));

                        Before       After
     NAME         X(5)  smith        smith
     LNAME        X(6)  abcdef       SMITH␣

MOVE (ADDRESS) = UPPER("123␣Main");

                        Before       After
     ADDRESS      X(8)  abcdefgh     123␣MAIN

MOVE (ACTION(2)) = UPPER((VERB((I)))) + "ed";

                        Before       After
     I            I(5)  1            1
     VERB(1)      X(4)  jump         jump
     ACTION(2)    X(6)  turns␣       JUMPed

MOVE (LNAME) = UPPER((NAME));

                        Before       After
     NAME      X(7,,7)  abcdefg      abcdefg
     LNAME     X(6,,7)  JOHN␣J␣     ABCDEFG
```

In the preceding example using a storage length of 7, the item LNAME will contain ABCDEFG in the data register, but when displayed, LNAME will only display the first 6 characters, ABCDEF.

Examples

The first example copies the values for FIELD-A into FIELD-B.

```
MOVE (FIELD-B) = (FIELD-A);

                        Before       After
     FIELD-A      X(4)  SAM␣         SAM␣                  <<no change>>

     FIELD-B      X(5)  CHUCK        SAM␣␣
```

The next example moves the first two characters of DATE into MONTH.

```
MOVE (MONTH) = (DATE);

                        Before       After
     DATE         X(6)  100770       100770                <<no change>>

     MONTH        X(2)  12           10
```

The next example shows concatenation. Note that the trailing blanks in FIELD1 are stripped when the two fields are concatenated.

```
MOVE (NEWFIELD) = (FIELD1) + (FIELD2);

                        Before       After
     FIELD1       X(4)  AB␣␣         AB␣␣                  <<no change>>

     FIELD2       X(3)  CDE          CDE                   <<no change>>

     NEWFIELD     X(6)  123456       ABCDE␣
```

The following example shows the removal of internal characters:

```
MOVE (DATE) = (FDATE) - (SLASH);

                        Before       After
     FDATE        X(8)  01/31/82     01/31/82              <<no change>>

     SLASH        X(1)  /            /                     <<no change>>

     DATE         X(6)  ␣␣␣␣␣␣       013182
```

The next example shows justification:

```
MOVE (FIELDY) = -(FIELDX);

                        Before       After
     FIELDX       X(4)  ABC␣         ABC␣                  <<no change>>

     FIELDY       X(4)  1234         ␣ABC
```

The next examples show justifications using fields of different lengths.

```
MOVE (FIELDB) = –(FIELDA);

                         Before       After
     FIELDA       X(4)   XYZ␣         XYZ␣

     FIELDB       X(8)   12345678     ␣␣␣␣␣XYZ


MOVE (FIELDA) = –(FIELDB);

                         Before       After
     FIELDA       X(4)   XYZ␣         1234

     FIELDB       X(8)   123456␣␣     123456␣␣


MOVE (FIELDA) = –(FIELDB);

                         Before       After
     FIELDA       X(4)   XYZ␣         ␣123

     FIELDB       X(8)   123␣␣␣␣␣     123␣␣␣␣␣
```

The following example demonstrates the use of MOVE with numeric data items of different lengths.

```
SYSTEM T6121;

DEFINE(ITEM) INTARRAY 10 I(4):
             INT         I(4);
LIST INTARRAY: INT;

LET (INT) = 65;
MOVE (INTARRAY) = (INT);
DISPLAY INTARRAY;

EXIT;
```

The result in INTARRAY is the first element has 65 and all others have 0 because MOVE fills numeric type items with zeros when the source length is smaller than the destination. Be sure that the definitions of the source and destinations are the same, since no type conversion is performed by MOVE.

When assigning a value to an array, the MOVE verb treats the array as a simple compound item and moves each byte one at a time until the end of the value or the end of the array, whichever comes first. The remaining elements are filled with blanks (if 9, X, or U data types) or filled with null characters (if numeric data types).

If a subscript is specified, only that element is assigned the value and all other subscripts remain unchanged.

For example, if ARRAY-X is defined as 6X(2) and ARRAY-I is defined as 4I(5,,2).

```
MOVE (ARRAY-X) = "abcdefgh"; <<Sets 1st element to ab; 2nd to cd, etc.>>

MOVE (ARRAY-X(2)) = "ZZ";    <<Sets 2nd element to ZZ.>>

LET (TEMP-I) = 67;
MOVE (ARRAY-I) = (TEMP-I);   <<Sets 1st element to 67 and rest >>
                             <<to nulls (binary 0).>>
LET (TEMP-I) = 78;
MOVE (ARRAY-I(4)) = (TEMP-I);  <<Sets only 4th element to 78.>>
```

See Chapter 3 for more information on handling arrays.

OUTPUT

Performs a multiple data retrieval from a file or data set and displays the data.

Syntax

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

OUTPUT specifies a database 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 PERFORM= statements are executed. This allows you to display the results of 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:

- Displays values in the order in which they appear in data register.
- Accompanies 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 a dictionary definition of the item, or
  - the associated data item name in the list register.
- Displays 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 from the next. If a field cannot fit on the current display line, then the field begins on a new line.

Note

After the first retrieval, Transact uses an asterisk (*) for the call list to optimize subsequent retrievals of that data set.

Statement Parts

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

none	Retrieves an entry from a master set based on the key value in the argument register. This option does not use the match register.
CHAIN	Retrieves entries from a KSAM file key or a detail chain. The entries must meet any match criteria set in the match register in order to be collected. The contents of the key and argument registers specify the chain or KSAM key 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	Retrieves the last entry that was accessed from the MPE or KSAM file or data set.
DIRECT	Retrieves the entry stored at a specified record number from an MPE or KSAM file or a data set. Before using this modifier, store the record number as a 32-bit integer I(10,,4) in the item referenced by the RECNO= option.
PRIMARY	Retrieves the 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	Retrieves entries from a detail set in the same manner as the CHAIN option, only in reverse order. For a KSAM file, this operation is identical to CHAIN.
RSERIAL	Retrieves entries from a 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	Retrieves entries in serial mode from an MPE or KSAM file or a 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 by 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*])	Suppresses 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. The *item-name* must be a parent.

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 Chapter 7.

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 can 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.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

The LIST= option has a limit of 64 individually listed item names and a limit of 255 items specified by a range for a TurboIMAGE data set.

All item names specified must be parent items.

The options for *range-list* and the data items retrieved by OUTPUT include the following:

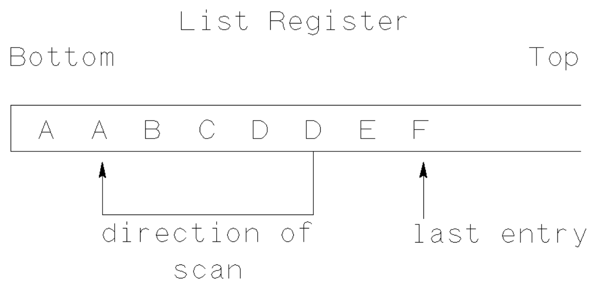
(*item-name*)

A single data item.

(*item-nameX*: *item-nameY*)

All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items, and if there are no data items between the two on the list register, no database access is performed. (See the DEFINE(ITEM) verb description.)

(*item-nameX*)	All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.
(*:item-nameY*)	All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.
(*item-nameX,* *item-nameY,* ... *item-nameZ*)	The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files or for VPLUS forms, data items must be specified in the order of their occurrence in the physical record or form. This order need not match the order of the data items on the list register. Do not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
(@)	Specifies a range of all data items of *file-name* as defined in a dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.
(#)	Specifies an enumeration of all data items of *file-name* as defined in the data

dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the data dictionary. This order need not match the order of the data items in the list register.

() A null data item list. Accesses the file or data set, but does not retrieve any data.

LOCK
: Locks the specified file or database. The lock is active the entire time that the OUTPUT executes. If LOCK is not specified and a TurboIMAGE data set is being accessed, no locking is done.

  When a KSAM or MPE file is being accessed, if LOCK is not specified on the OUTPUT statement but is specified for the file in the SYSTEM statement, then the file 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.

  Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the OUTPUT verb.

  For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on OUTPUT if SET(OPTION) NOLOCK is specified.

  See "Database and File Locking" in Chapter 6 for more information on locking.

NOCOUNT
: Suppresses the message normally generated to indicate the number of entries found.

NOHEAD
: Suppresses default headings for the displayed values.

NOMATCH
: Ignores 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
: Suppresses the standard error message produced as a result of a file or database error. All other error recovery actions occur.

PERFORM=*label*
: Executes the code following the specified label for every entry retrieved by the OUTPUT operation. The entries can 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 having to code loop control logic. You can nest up to 10 PERFORM= options.

RECNO=*item-name* [(*subscript*)]
: With the DIRECT modifier, you must initialize *item-name* to contain the 32-bit integer number (I(10,,4))

of the record to be retrieved. With other modifiers, Transact returns the record number of the retrieved record in *item-name*, a 32-bit integer (I(10,,4)).

The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

SINGLE
: Retrieves and displays only the first entry that satisfies any selection criteria.

SOPT
: Suppresses the optimization of database calls. This option is primarily intended to support a database operation in a performed routine that is called recursively. The option allows a different path for the same detail set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of a call list of "*" after the first call is made. Use SOPT if you are calling TurboIMAGE through the PROC or CALL verbs. For an example of how SOPT is used, see "Examples" at the end of the FIND verb. For more detailed information on SOPT, see "Suppression of Optimization versus WORKFILE" in the FIND statement of this chapter.

```
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, retrieves each selected record,
- then, executes any PERFORM= statements,
- then, writes the specified items to the sort file, and, after writing all the records to the sort file,
- sorts the sort file, and
- displays the sorted output.

The SYSTEM statement determines the size of the sort file.

You can specify either ascending or descending sort order. The default is ascending order. (See the FIND verb description for a different processing sequence.)

STATUS

Suppresses the action defined in Chapter 7 under "Automatic Error Handling." You will need to add code to check the value of STATUS, as shown in the example below. When STATUS is specified, the effect of an OUTPUT statement is described by the 32-bit integer 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 database or MPE/KSAM file system error documentation corresponding to the value.

STATUS causes the following with OUTPUT:

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

See "Using the STATUS Option" in Chapter 7.

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.

Example 1

```
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);
```

Example 2

```
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 displaying 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;
```

PATH

Establishes a chained access path to a data set or a KSAM file.

Syntax

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

PATH uses the key and argument registers that correspond to the KSAM key for setting up chained access for the KSAM file or chained access for a 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 a detail set. The number of entries is not returned for a KSAM file.

You must use a PATH statement to establish the path for chained access to a KSAM file or a data set when the STATUS option is included in a subsequent data access statement. The PATH verb cannot be used with MPE files.

PATH performs file and key validations during program execution. If the attributes do not match the current database or file, an error message is displayed.

Statement Parts

file-name The KSAM file or 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 will not contain the number of entries for a KSAM file.

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

ERROR=*label* ([*item-name*]) Suppresses 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. The *item-name* must be a parent.

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 Chapter 7.

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.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

All item names specified must be parent items.

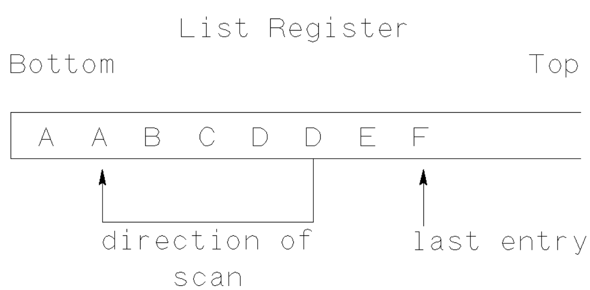
The LIST= option has a limit of 64 individually listed item names and a limit of 255 items specified by a range.

The options for *range-list* and the records upon which they operate include the following:

(*item-name*) A single data item.

(*item-nameX: item-nameY*) All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected.

(*item-nameX:*) All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.

(*:item-nameY*)	All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.
(*item-nameX*, *item-nameY*, ... *item-nameZ*)	The data items are selected from the list register. For KSAM files, data items must be specified in the order of their occurrence in the physical record. This order need not match the order of the data items on the list register. This option incurs some system overhead.
(@)	Specifies a range of all data items of *file-name* as defined in a data dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.
(#)	Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the data dictionary. This order need not match the order of the data items in the list register.
( )	A null data item list. Operates on the file but does not retrieve any data.

NOMSG — Suppresses the standard error message produced by Transact as a result of a file or database error.

STATUS

Suppresses the action defined in Chapter 7 under "Automatic Error Handling." You will need to add code to check the value of STATUS. When STATUS is specified, the effect of a PATH statement is described by the value in a 32-bit integer 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 the database or KSAM file system error documentation that corresponds 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 set is not found in the associated master set. (See Table 7-4 for other status register values.)

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 a detail 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 database 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.

PERFORM

Transfers control to a labeled statement.

Syntax

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 corresponding PERFORM statement.
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.
another *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. 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

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.

```
IF INPUT = "YES", "Y" THEN
   PERFORM ADD-IT
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;
```

PROC

Calls a procedure that has been placed into a segmented library file (SL) for Transact/V or compatibility mode. PROC also calls procedures from an executable library (XL) for native mode programs on Transact/iX.

Syntax

`PROC` *procedure-name* [(*parameter-list*)] [,*option-list*] ;

Transact/V

PROC calls an MPE system intrinsic or other compiled procedure that is resident in an SL file. SL files are searched and procedures and intrinsics are dynamically loaded in the following order: logon group SL, logon 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. Bit maps are always required for any PROC call to an option variable intrinsic or user defined procedure. They are passed by value as the last parameter in a parameter list. A bit map is formed by setting a string of bits to one or zero, depending on whether a parameter is passed or not passed, respectively. The bit string is then right-justified in a 16- or 32-bit word (depending on the number of possible parameters) and converted into an integer value. This value is passed to the option variable procedure as the last parameter. See the *SPL Reference Manual* for more information on Option Variable bit maps.

All system intrinsics called can be declared in a DEFINE(INTRINSIC) statement. When this is done, the intrinsics are resolved only from the system SL.

Transact/iX

The Transact PROC verb is the same, in effect, under MPE V and MPE/iX. It is used to call procedures written in other languages. The primary point to be aware of is that both Transact subprograms and routines written in other languages must reside in an executable library (XL) or be linked into your program if they are to be called by a Transact program under MPE/iX. Switch routines must be written for any user defined subroutines running in compatibility mode, including all SPL routines.

Two features, the PROCALIGNED_16/32/64 compiler options and the *%n* alignment options, allow you to tune applications with respect to the overhead needed for calling external procedures. The PROCALIGNED_16/32/64 compiler options are discussed in detail under "Transact/iX Compiler Options" in Chapter 9. The parameter alignment options are described later in this section.

A third feature, the PROCINTRINSIC compiler option, is designed to ease the migration of programs that call system intrinsics. Compiler options are discussed in Chapter 9.

Another item to note is that no conversion between IEEE floating point (real) numbers and HP3000 floating point numbers is attempted. When passing parameters or data that access real numbers, the called procedure or intrinsic must be compiled with the same real number format as the main program. (See "Floating Point Formats" in Appendix B.)

Statement Parts

procedure-name The name under which the procedure is listed in the SL or XL.

parameter-list The items in the *parameter-list* specify one or more parameters that are passed between the Transact program and the external procedure. The list can contain any number of variables, constants, and literals separated by commas. The order in which you place them 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 "&".

The following special characters can precede any *item-name* parameter or key word parameter:

% Passes the given parameter by byte address (by reference).

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 can be included in the *parameter-list*, and it can appear anywhere in the list.

The default (no special character) passes an *item-name* or key word parameter by a 16-bit address. "*Character-string*" parameters are only passed by reference, and *numeric-constant* parameters are only passed by value.

You can indicate to the called procedure the existence of a null parameter by placing consecutive commas on the list. Transact passes a 16-bit value of zero for this null parameter. Use two commas if the parameter has a 32-bit value, and is passed by value. Use one comma if the parameter is passed by reference.

You can also indicate a null parameter by placing `""` as the parameter.

All addresses specified by the items in *parameter-list* are 16-bit addresses. If you want to specify a byte address, precede the *item-name* with "%". For example, ITEM(NUM) specifies a 16-bit address, whereas %ITEM(NUM) specifies a byte address. PROC does not automatically align data parameters on 16-bit boundaries.

Note Transact does not verify that parameters are correctly set up. You must verify this before attempting to call a procedure.

The *parameter-list* may consist of any of the following:

(*item-name* [(*subscript*)])	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 16-bit boundary in the data register if you want to pass a 16-bit address. The beginning of the data register is on a 16-bit boundary; if you add items with an odd number of bytes, you should add a dummy fill character to retain 16-bit boundaries.
"*character-string*"	A programmer-defined character string. If *character-string* is null, as in `""`, the parameter is filled with binary zeros. Use a space for the character string, `" "` to fill the field with blanks. The default is that the *character-string* is passed by a 16-bit address (by reference). Precede the *character-string* with a "%" sign to pass it by a byte address. You can only pass *character-strings* for those parameters expecting a reference. For more information, see "Character-String and Numeric-Constant Parameters" later in this verb.
numeric-constant	A numeric value. The *numeric-constant* is passed by value. You do not need to precede this parameter with a "#". You can only pass *numeric-constants* for those parameters expecting a value. For more information, see "Character-String and Numeric-Constant Parameters" later in this verb.

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 a 16-bit integer (I(5,,2)) containing the byte length of the argument value.
BASE[(*base-name*)]	Address of a logical array containing the name of the given database preceded by the two-character *base-id* supplied by the database, and followed by a blank character. If no *base-name* is specified, then the home base is assumed. Note, the home base cannot be specified.
BASELNG [(*base-name*)]	Address of a 16-bit integer (I(5,,2)) containing the byte length of the given *base-name*, including the terminating blank.

BYTE(*item-name*)	Address of a 16-bit integer (I(5,,2)) containing the byte length of the value of the given item.
COUNT(*item-name*)	Address of a 16-bit integer (I(5,,2)) containing any subitem occurrence count for the given item. A value of 1 means that the given item is not a compound type containing subitems.
DECIMAL (*item-name*)	Address of a 16-bit integer (I(5,,2)) containing the decimal place count for the given item.
FILEID(*file-name*)	Address of a 16-bit integer (I(5,,2)) 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 a 16-bit integer (I(5,,2)) 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 a 16-bit integer (I(5,,2)) 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 a 16-bit integer (I(5,,2)) containing the byte length of the data item name in the key, including the terminating semicolon.
POSITION (*item-name*)	Address of a 16-bit integer (I(5,,2)) containing the position (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 a 16-bit integer (I(5,,2)) containing the byte length of the given data set name, including the terminating blank.

SIZE(*item-name*)	Address of a 16-bit integer (I(5,,2)) containing the byte length of the display or entry format for the given item.
STATUS	Address of the lower order 16 bits of the 32-bit status register set by Transact. If the STATUS parameter is NOT used, then the 32-bit 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. Upon exiting from PROC, the entire 32 bits of the status register is set to the value in the lower order 16 bits of the status register.
STATUS(DB)	Address of the condition word block returned by the database. (The discussion of MOVE explains how to use this value.)
STATUS(IN)	Address of a 16-bit integer (I(5,,2)) containing the STATUS value following the most recent user input statement (PROMPT, DATA, or INPUT). (See the appropriate verb for the interpretation of the STATUS value.)
TYPE(*item-name*)	Address of a 16-bit integer (I(5,,2)) containing a code that represents the data type of *item-name*. The 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*. (See the discussion of the VPLS option under SET(OPTION) in this chapter.)

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

UNLOAD (*This option is for Transact/V only.*) Unloads the procedure being called following execution; that is, removes 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.

For Transact/iX, all procedures are bound at link time or as a part of the RUN command. If you use the UNLOAD option you will get the compile message:

```
INFO: THE 'UNLOAD' OPTION FOR THE PROC VERB HAS NO MEANING
   ON AN MPE/IX SYSTEM.
```

NOTRAP Ignores 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 Loads 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 for Transact/V only, 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.

Note The following option should not be used with Transact/iX.

---

language Used to specify the language in which the procedure is written: Pascal, COBOL, FORTRAN, BASIC, or SPL. This option is needed to call COBOL procedures to avoid an arithmetic trap when the stack exceeds 16K 16-bit words.

Parameters Passed by Byte Address

An option is available on the PROC verb to specify alignment on individual reference parameters passed by byte address. This option is only available for item variables; it is not used for *character-string* parameters. This option takes the form "*%n*", where *n* can be 8, 16, 32, or 64, as follows:

%8 Align parameter on an 8-bit boundary (this is the default)

%16 Align parameter on a 16-bit boundary

%32 Align parameter on a 32-bit boundary

%64 Align parameter on a 64-bit boundary

The alignment option must precede the parameter affected. For example,

```
PROC GETNAME (%32(NAME));
```

This option takes precedence over the PROCALIGNED_16/32/64 options for the individual parameter. It is only active for the Transact/iX compiler. Under the Transact/V processor, all parameters passed on a greater than 8-bit boundary are treated as 16-bit address parameters. When PROCINTRINSIC is specified, but the alignment check is less than that required by the intrinsic definition in SYSINTR.PUB.SYS, an error occurs at run time.

Parameters Passed by Value or by Reference

Transact/V does not check passed parameters to verify that they are of the same type as the parameters expected by the called procedure. The Transact/iX compiler checks calls to system intrinsics, verifying that reference parameters are passed by reference and value parameters are passed by value. An informational message is reported if a parameter is not passed in the expected way.

For example, the ASCII intrinsic expects the first parameter to be passed by value. If, instead, it is passed by reference using the PROC verb, the Transact/iX compiler issues the following informational messages:

```
*INFO: PROC PARAMETER 1 WAS PASSED BY REFERENCE WHEN VALUE EXPECTED
*INFO: ERRORS IN PROC PARAMETERS TO 'ASCII' WILL CAUSE A RUN TIME ERROR
```

At run time, the following error occurs when the PROC ASCII statement is encountered:

```
*ERROR: PARAMETER SPECIFICATION ERRORS PREVENTED PROC CALL
```

Transact/V programs that take advantage of no type checking may require that you write a procedure to provide the same functionality as the intrinsic being accessed. For example, since no parameter type checking is done on calls to user defined procedures, you can code a procedure which has the same parameters as the intrinsic and which merely calls the intrinsic. In Transact/iX, you would then use the PROC verb to call this procedure rather than the intrinsic, passing the parameters in the same way as when the intrinsic was called directly.

Character-String and Numeric-Constant Parameters

For Transact/V, *numeric-constant* parameters are passed by value. If the number passed is four digits or less (from -9999 to 9999), the number is assumed to be a 16-bit integer such as I(4,,2). If more than four digits, the number is assumed to be a 32-bit integer such as I(9,,4). A number passed with a decimal point is truncated to an integer value.

To pass a 32-bit integer parameter with four digits or less, precede the integer with zeros until the integer is at least five digits long. The following example shows the integer 48 being passed as a 32-bit value because it is preceded by three zeros.

```
PROC DASCII(00048, 10, %(ASCIIEQV), &(NUMCHAR));
              ↑     ↑
           32-bit 16-bit
```

For Transact/iX, *character-strings* and *numeric-constants* passed as parameters for system intrinsics are verified according to the data type for which they are being passed (the data type of the formal parameter of the system intrinsic). This checking of the data type is performed when the PROCINTRINSIC compiler option is specified. When the PROCINTRINSIC compiler option is specified or the DEFINE(INTRINSIC) verb is used, the numeric constant parameters are type-checked to be 16-bit integers such as I(4,,2). You can bypass this limitation by adding the procedure to the SYSINTR file or using variable parameters instead of constants. *Numeric-constants* must be less than 11 characters (a sign plus 10 digits including the decimal point).

Accessing COBOL Subroutines

When the Transact/iX compiler generates the procedure name in the PROC statement, hyphens are left standing—they are neither converted to underscores nor removed. On the other hand, COBOL II/iX converts hyphens to underscores. Therefore, a COBOL II subroutine that is recognized by Transact/V in compatibility mode will not be recognized when it is recompiled using COBOL II/iX and linked or loaded with a Transact/iX program. To make the names consistent, specify the COBOL II/iX compiler option that removes hyphens from COBOL subroutine names or use underscores instead of hyphens when naming COBOL subroutines that will be used under Transact.

Option Variable Procedures

Option variable procedures do not exist under MPE/iX. The Transact/iX compiler only supports calls to option variable intrinsics if the intrinsic is declared in a DEFINE(INTRINSIC) statement or the PROCINTRINSIC option is specified when compiled. The bit map included as part of the parameter list is ignored and the remaining parameters are checked as specified in SYSINTR.PUB.SYS. No implicit type conversions are performed.

User-defined option variable procedures must be accessed either through a switch to a compatibility mode routine or conversion to fixed parameter procedures and recompilation with a native mode compiler. The first option is the only option available to users of SPL procedures who do not want to recode these routines in a native MPE/iX language.

Null Parameters

Under Transact/iX versions prior to A.06.00, all null parameters for option-extensible intrinsics must be designated by single commas. Transact/iX version A.06.00 and later use default parameters for system intrinsics where null parameters appear. Default parameters are also used where the procedure invocation specifies less than the maximum number of parameters for an option-extensible routine. Commas are not necessary after the last specified parameter for system intrinsics that have the EXTENSIBLE option. See the *Pascal/iX Reference Manual* and *Pascal/iX Programmer's Guide* for more information on the EXTENSIBLE option.

The Transact/V convention of using two or four consecutive commas to denote a null 32-bit or 64-bit value parameter is interpreted by Transact/iX as denoting two or four null parameters.

You can avoid this incompatibility by modifying the source so all 32-bit and 64-bit value parameters of option-extensible intrinsics are passed. Another method is to modify the Transact code to use only single commas in place of 32-bit or 64-bit null value parameters, but this method makes the modified source code incompatible with Transact/V.

Null parameters passed to user-defined procedures under Transact/V cause 16-bit zeros to be passed under Transact/iX.

Locating Procedures

Under Transact/V, there are two library search methods for resolving procedures accessed via the PROC verb. If the procedure name has not been included in a DEFINE(INTRINSIC) statement, the SL's are searched as follows: the logon group, the PUB group in the logon account, and finally, the PUB.SYS group. If the procedure name has been included in a DEFINE(INTRINSIC) statement, SL.PUB.SYS will be searched. Under Transact/iX, the libraries and the order in which they are searched must be specified at either link or run time.

The libraries and the order in which they are searched by processes CREATED and ACTIVATED by Transact/iX must be specified in the :RUN command used to run the Transact/iX program. The LIBSEARCH bits on CREATE and ACTIVATE should be set to "NO" to force the create process to use the LIBLIST specified on the :RUN command.

Double Buffering Parameters

By default, Transact/iX generates code to double buffer all reference parameters (parameters passed by address) if they are not preceded by "%", "#", or "&". The double buffered alignment is determined from the type and size of the data item passed via the PROC call. However, since double buffering is inefficient, the compiler options PROCALIGNED_16/32/64 should be used whenever possible to bypass double buffering.

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 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 "%". For additional examples of the PROC verb, see "Migration Examples" in Appendix B.

The example below calls the VPLUS 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 MAP parameter sets up a bit map for an intrinsic that is type OPTION VARIABLE.

The following example calls the intrinsics CREATE and ACTIVATE. (See 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),INIT=(BINARY(1000001)):
                                 <<Flags                              >>
             MAP     I(4),INIT=(BINARY(1010100000));
                                 <<Bit map for optional parameters    >>
$$A:
    LIST ROUTINE,INIT:
         CPIN,INIT:
         CFLAG:
         MAP;

    DATA ROUTINE("WHICH PROCESS?");

    PROC CREATE (%(ROUTINE),,(CPIN),,#(CFLAG),,,,,,#(MAP));

    LET (MAP)   = 3;
    LET (CFLAG) = 3;

    PROC ACTIVATE (#(CPIN),#(CFLAG),#(MAP));

    END;
```

The LET and second PROC statement above can also be replaced with the PROC statement below. This statement passes numeric constants as well as variable parameters.

```
    PROC ACTIVATE (#(CPIN),3,3);
```

The following example shows 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):
             COUNT   I(4):
             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));
```

The next example calls the database 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,
             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   1415 I+(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 next example calls the BRW intrinsic BRWEXEC to execute a report on line.

```
SYSTEM TEST;
DEFINE(ITEM) BRW-COMAREA X(300):
              RETURN-STATUS      I(4)  = BRW-COMAREA(1):
              ERROR-PARM         I(4)  = BRW-COMAREA(3);
DEFINE(ITEM) BRW-PARAMETERS X(176):
              MAX-NUM-PARMS      I(4)  = BRW-PARAMETERS(1):
              ACTUAL-NUM-PARMS   I(4)  = BRW-PARAMETERS(3):
              PARM-NAME-1        X(20) = BRW-PARAMETERS(5):
              PARM-TYPE-1        I(4)  = BRW-PARAMETERS(25):
              PARMRESULT-TYPE1   I(4)  = BRW-PARAMETERS(27):
              RESULT-LENGTH-1    I(4)  = BRW-PARAMETERS(29):
              PARM-MODE-1        I(4)  = BRW-PARAMETERS(31):
              UPSHIFT-1          I(4)  = BRW-PARAMETERS(33):
```

```
                PARM-VALUE-1        X(55) = BRW-PARAMETERS(35):
                RESERVE-1           X(1)  = BRW-PARAMETERS(90):
                PARM-NAME-2         X(20) = BRW-PARAMETERS(91):
                PARM-TYPE-2         I(4)  = BRW-PARAMETERS(111):
                PARMRESULT-TYPE2    I(4)  = BRW-PARAMETERS(113):
                RESULT-LENGTH-2     I(4)  = BRW-PARAMETERS(115):
                PARM-MODE-2         I(4)  = BRW-PARAMETERS(117):
                UPSHIFT-2           I(4)  = BRW-PARAMETERS(119):
                PARM-VALUE-2        X(55) = BRW-PARAMETERS(121):
                RESERVE-2           X(1)  = BRW-PARAMETERS(176);

LIST BRW-COMAREA:
     BRW-PARAMETERS;

LET (MAX-NUM-PARMS) = 2;

LET (ACTUAL-NUM-PARMS) = 1;
MOVE (PARM-NAME-1) = "$REPORT";
    .
    .
    .
PROC BRWEXEC((BRW-COMAREA),(BRW-PARAMETERS));
```

The next example shows a call to the compiler library routine DABS' to determine the absolute value of a number.

```
SYSTEM ABSTST;

DEFINE(ITEM) REALVALUE   R(8,2,8),  INIT=-128.8:
             RESULT      R(8,2,8),  INIT=;

LIST REALVALUE: RESULT;

DISPLAY REALVALUE: RESULT;

PROC DABS'(#(REALVALUE),&(RESULT));

DISPLAY REALVALUE: RESULT;

END;
```

There are two things to check when using the compiler library:

- Make sure you use the PROCINTRINSIC compiler option for Transact/iX and the DEFINE(INTRINSIC) statement for Transact/V.
- Verify that all parameter types match the function parameters and function return.

In addition, be aware that parameters passed by value are preceded by a "#" (pound sign). The last parameter is the function return and it is preceded by a "&" (ampersand).

The last example is a Transact program that calls BRW.

Prior to running the program, we used BRW to design a report and compile the report into a BRW execution file named BRWEXECR. The Transact program uses VPLUS to present the user with a main menu of options. If the user enters option 1, the BRW report is executed. The PROC calls result in the BRW Report Selection menu being displayed with the name of the report requested already filled in. The user then requests the report the same as when running BRW directly.

```
SYSTEM BRW,VPLS=MYFF(MAINMENU(SELECTION));

DEFINE(ITEM) BRW-COMAREA X(106):
             BRW-STATUS     I(4) = BRW-COMAREA:
             BRW-ERROR      I(4) = BRW-COMAREA(3):
             BRW-COM-LENGTH I(4) = BRW-COMAREA(5):
             BRW-EXEC-FILE. X(36)= BRW-COMAREA(7):
             BRW-DEFAULTS   I(4),  INIT=0:
             SELECTION      I+(1);

LIST BRW-COMAREA:
     BRW-DEFAULTS:
     SELECTION;

LET (BRW-COM-LENGTH) = 50;

GET(FORM) MAINMENU;

IF (SELECTION) = 1 THEN
  DO
    PROC BRWINITREQUEST ((BRW-COMAREA));
    MOVE (BRW-EXEC-FILE) = "BRWEXECR";
    PROC BRWSTARTREQUEST ((BRW-COMAREA),
                          (BRW-DEFAULTS));
    PROC BRWSTOPREQUEST ((BRW-COMAREA));
  DOEND;

EXIT;
```

Below are some miscellaneous PROC examples that pass character string parameters and numeric constants as parameters.

```
PROC ASCII (65,10,%(STRING),&(NUMCHAR));

PROC VPRINTFORM (VCOM(CUSTFORM),1,0);

PROC CREATE (%(ROUTINE),"",(CPIN),"",65,,,,,,672);

PROC DBCLOSE (BASE,SET(CUST-MAST),5,STATUS(DB));

PROC FWRITE (#FILEID(TRANOUT), "HELLO THERE WORLD!!", -19, 0);

PROC DABS' (-128,&(RESULT));
```

PROMPT

Accepts input from the user terminal and places the supplied values into the list, data, argument, match, and/or update registers.

Syntax

PROMPT[(*modifier*)] *item-name*[("*prompt-string*")][,*option-list*]
[:*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 can choose from the following:

none	Adds item name to list register and input value to data register. (See Syntax Option 1.)
KEY	Adds item name to key register and adds input value to argument register. (See Syntax Option 2.)
MATCH	Adds item name to list and match registers and adds input value to data register. Also sets up input value in match register as a match criterion. (See Syntax Option 3.)
PATH	Adds item name to list and key registers, and adds input value to data and argument registers. (See Syntax Option 4.)
SET	Adds item name to list register and adds input value to data register, unless response is a carriage return. (See Syntax Option 5.)
UPDATE	Adds item name to list and update registers and input value to data register; also adds 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 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 a dictionary or in the DEFINE(ITEM) definition of the item.

Transact validates the input value as to type, length, or any other characteristics specified in a 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. With native language support, Transact validates numeric data using the thousands and decimal indicators of the language in effect. For more information, see Appendix E, "Native Language Support."

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. If there is no entry text, 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 of the following options (separated by commas) for any syntax option. (See Syntax Option 3, PROMPT(MATCH) for additional options.)

BLANKS — Does not suppress leading blanks supplied in the input value. (Leading and trailing blanks are normally stripped.)

CHECK= *set-name* — Checks the input value against the 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.

Note

The CHECK= or CHECKNOT= options cannot be used to check against MPE or KSAM files, nor can either option be included in a PROMPT(MATCH) statement. Also, if the CHECK= or CHECKNOT= option is used with STATUS, then "]", "]]", or a carriage return suppresses the data set operation and control passes to the next statement.

CHECKNOT= *set-name* — Checks input value against the 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, Transact displays an appropriate error message and reissues the prompt.

NOECHO — Does not echo the input value to the terminal. If omitted, the input value is displayed on the terminal.

RIGHT — Right-justifies the input value within the data register field. By default, the input value is left-justified.

STATUS — Suppresses normal processing of "]" and "]]", which cause an escape to a higher processing or command level.

Status Register Value	Meaning
−1	User entered a "]".
−2	User entered a "]]".
−3	User entered one or more blanks and no non-blank characters.
−4	If timeout is enabled with a FILE(CONTROL) statement, a timeout has occurred.
> 0	Number of characters (includes leading blanks if BLANKS option is specified); no trailing blanks are counted.

The STATUS option allows you to control subsequent processing by testing the contents of the register with an IF statement.

Syntax Options

(1) PROMPT *item-name*[("*prompt-string*")][,*option-list*][:*item-name* ...] ... ;

PROMPT with no modifier adds the *item-name* to the list register and the input value to the data register.

Specifying the ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

Note Only compile time alignment is supported.

(2) PROMPT(KEY) *item-name*[("*prompt-string*")][,*option-list*][:*item-name* ...] ... ;

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. When there is more than one *item-name* in a PROMPT(KEY) statement, only the final *item-name* and value are retained in the key and argument registers.

Note The PROMPT(KEY) statement is the only form of the PROMPT statement that does not update the list register or data register.

(3) PROMPT(MATCH) *item-name*[("*prompt-string*")][,*option-list*][:*item-name* ...] ... ;

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 database or file operation.

The user response to PROMPT(MATCH) can be any of the valid selection criteria described under "Responding to a MATCH Prompt" in Chapter 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.

If the item name is an unsubscripted array, only the value of the first element of the array will be set in the data register. This value from the data register will be set up as a match criterion in the match register.

The MATCH modifier allows one or more of the *option-list* items listed under "Statement Parts", except for CHECK= and CHECKNOT=, which are not allowed in a PROMPT(MATCH) statement. Additionally, you can select one of the following options to specify that a match selection is to be performed on a basis other than equality.

If you specify one of the options listed below, the entire user input is treated as a single value. The match specification characters described in Chapter 5 are not allowed as user input with the options listed below.

option-list: Any of the following options can be selected:

ALIGN	Forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX
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 database 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*][:*item-name* ...] ... ;

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. When there is more than one *item-name* in a PROMPT(PATH) statement, only the final *item-name* and value are retained in the key and argument registers.

Specifying the ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

(5) PROMPT(SET) *item-name* [("*prompt-string*")][,*option-list*][:*item-name* ...] ... ;

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. 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 with RESET(MATCH) before you reissue the prompt.

```
PROMPT(SET) EMPL:
            DEPT:
            PHONE:
            ROOM:
            LOCATION;


    EMPL>
    DEPT>
   PHONE> 278
    ROOM> 312
LOCATION>
```

Specifying the ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

(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.

Specifying the ALIGN option forces the item to be aligned on a 16-bit boundary in Transact/V and on a 32-bit boundary in Transact/iX.

Examples

This example causes a sequence of prompts to be displayed:

```
$$ADD:                        <<Add a new record                    >>
   $CUSTOMER:
      PROMPT CUST-NAME("CUSTOMER'S NAME"):
             CUST-ADDR:
             CUST-CITY:
             CUST-PHONE;
```

This example is a result of the above code.

```
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;
```

The last example shows how the ALIGN option word-aligns SELECT-CODE in the list register.

```
PROMPT(MATCH) SELECT-CODE, ALIGN;
```

PUT

Moves data from the data register to a file, data set, or a VPLUS form.

Syntax

PUT[(*modifier*)]*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

modifier To specify the type of access from the data set or file, choose one of the following modifiers.

none	Adds an entry, based on the list and data registers, into a file or a data set.
FORM	Displays a VPLUS form on any VPLUS compatible terminal, and moves data to the form from the data register. If this modifier is not used, the destination must be a file or data set.

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* can be specified as any of the following:

form-name	Name of the form to be displayed by PUT(FORM).
(*item-name* [(*subscript*)])	Name of an item that contains the name of the form to be displayed by PUT(FORM). The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)
*	Displays 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*) that indicates the currently displayed form.
&	Displays 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 and the STATUS option are always available. The other options described below, may be used only without or only with the FORM modifier.

The list of items from the list register to be used for the PUT operation. For data sets, no child items can be specified in the range list. For PUT(FORM) only, items in the range list can be child items.

If the LIST= option is omitted with any modifier except FORM, all the items named in the list register are used. If the LIST option is omitted for PUT(FORM), the list of items in the list register, and either in the SYSTEM statement or the data dictionary for the form are used.

The LIST= option should not be used when specifying an asterisk (*) as the source.

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.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

The LIST= option has a limit of 64 individually listed item names. A range limitation of 255 items for TurboIMAGE data sets and 128 items for VPLUS forms also exists.

All item names specified must be parent items for files or data sets.

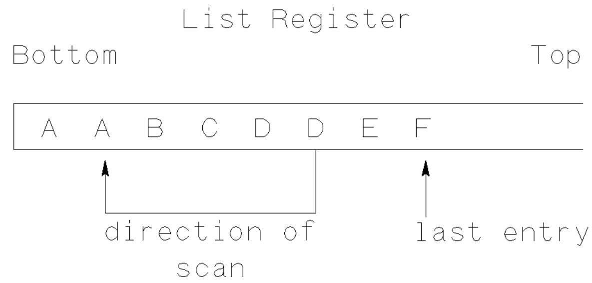
The options for *range-list* and the records upon which they operate include the following:

(*item-name*) A single data item.

(*item-nameX*: *item-nameY*) All the data items in the range from *item-nameX* through the last occurrence of *item-nameY*.

In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as follows,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items (see DEFINE(ITEM) verb) and if there are no data items between the two in the list register, no database access is performed.

(*item-nameX:*)	All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.
(:*item-nameY*)	All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register
(*item-nameX*, *item-nameY*, ... *item-nameZ*)	The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files and VPLUS forms, data items must be specified in the order of their occurrence in the physical record or form. This order need not match the order of the data items on the list register. Do not include child items in the list unless they are defined in the VPLUS form. This option incurs some system overhead.
(@)	Specifies a range list of all data items of *destination* as defined in a dictionary. This range is defined as *item-name-1:item-name-n* for the file.
(#)	Specifies an enumeration of all data items of *destination* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the dictionary. This order need not match the order of the data items in the list register.

( )	A null data item list. That is, accesses the file or data set, or displays the form, but does not transfer any data.

STATUS — Suppresses the action defined in Chapter 7 under "Automatic Error Handling." If you use this option, you should program your own error handling procedures.

When STATUS is specified, the effect of a PUT statement is described by the 32-bit integer 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 the condition word or MPE/KSAM file system error documentation corresponding to the value.

When the STATUS option is placed on the PUT(FORM), the PUT(FORM) only writes to the form. It does not read the function keys or the data from the form to the VPLUS buffer. For additional information, see the PUT flowcharts in Appendix A.

PUT with the STATUS option could be used as shown in one of the examples shown in the Examples section.

Options Available Without the Form Modifier

ERROR=*label* ([*item-name*]) — Suppresses 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. The *item-name* must be a parent.

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 Chapter 7.

LOCK — Locks the specified file or database. If a data set is being accessed, the lock is set the whole time that PUT executes. If the LOCK option is not specified but the database is opened in mode 1, then automatic locking will execute the lock.

For a KSAM or MPE file, if LOCK is not specified on PUT but is specified for the file in the SYSTEM statement, then the file 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.

Including the LOCK option will override the SET(OPTION) NOLOCK for the execution of the PUT verb.

A database opened in mode 1 must be locked while PUT executes. For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on PUT if SET(OPTION) NOLOCK is specified. If a lock is not specified (for a database opened in mode 1) an error is returned.

See "Database and File Locking" in Chapter 6 for more information on locking.

NOMSG	Suppresses the standard error message produced by Transact as a result of a file or database error.
RECNO=*item-name* [(*subscript*)]	Places the record number of the new entry into the data register space for *item-name*. *Item-name* must be defined as a 32-bit integer, such as I(10,,4). It can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

Options Available Only With the Form Modifier

APPEND	Appends 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	Clears the previously displayed form when the requested 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	Uses the form currently displayed on the terminal screen. That is, performs 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.
CURSOR=*field-name*| *item-name*[(*subscript*)]	Positions the cursor within the specified field. The *field-name* identifies the field and the *item-name* identifies the item which names the field. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

Note

To ensure that the cursor will be positioned on the correct field, you must have a one to one correspondence between the fields defined in VPLUS. Transact determines where to position the cursor by counting the fields.

FEDIT	Performs the field edits defined in the FORMSPEC definition for the form immediately before displaying it.
FKEY=*item-name* [(*subscript*)]	Moves the number of the function key pressed by the user in this operation to a 16-bit 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. The *item-name* may be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

F*n*=*label*
: Control passes to the labeled statement if the user presses function key *n*. *n* can have a value of 0 through 8, inclusive, where zero indicates the ENTER key. This option can be repeated as many times as necessary in a single PUT(FORM) statement.

FREEZE
: Freezes the specified form on the screen and appends 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
: Initializes the fields in the displayed form to any initial values defined for the form by FORMSPEC, or performs 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*]
: Does not return control to the program until the terminal user has pressed the function key *n*. *n* can 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 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 any key other than f8, the f8 exit function is disabled.

If F*n* is omitted, PUT(FORM) waits until any function key is pressed. If the user presses any of the function keys f1 through f7, the next record will be PUT; f8 retains its exit function.

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*)
: Places a message in the window area of the screen and, optionally, enhances a field in the form. The fields *field* and *message* can be specified as follows:

field
: Either the name of the data item for the field to be enhanced, or an *item-name* within parentheses which will contain the data item of the field to be enhanced at run time.

message — Either a "*string*" in quotes 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 shows 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 are determined by the LIST= option. In this example, the data in the data register is retrieved from a 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 database      >>
     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;
```

This example shows how the LIST=(#) option works, given a data set defined as follows:

```
NAME:          SUP-MASTER, MANUAL(13/12,18), DISC1;
ENTRY:         SUPPLIER(1),
               STREET-ADD,
               CITY,
               STATE,
               ZIP;
CAPACITY:      200;
```

The statement:

```
PUT SUP-MASTER,LIST=(#);
```

is equivalent to the statement:

```
PUT SUP-MASTER,LIST=(SUPPLIER,STREET-ADD,CITY,STATE,ZIP);
```

Use PUT with the STATUS option to check for error conditions. This example writes to an overflow file and issues an error message when the data set is full.

```
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;
```

REPEAT

Repeats execution of a simple or compound statement until a specified condition is true.

Syntax

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 one or more conditions, each made up of a *test-variable*, a *relational-operator*, and one or more *values*. Multiple conditions are joined by AND or OR. Execution of the statement following REPEAT continues until the test gives a value of true.

Statement Parts

statement — A simple or compound Transact statement can follow REPEAT. A compound statement is bracketed with a DO/DOEND pair.

condition-clause — One or more conditions, connected by AND or OR, where

AND — is a logical conjunction. The condition clause is true if all of the conditions are true; it is false if one of the conditions is false.

OR — is a logical inclusive OR. The condition clause is true if any of the conditions is true; it is false if all of the conditions are false.

Each condition contains a *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value[,*value*] ...

test-variable — Can be one or more of the following:

(*item-name* [(*subscript*)]) — The value in the data register that corresponds to *item-name*. The *item-name* may be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

[*arithmetic expression*] — An arithmetic expression containing item names and/or constants. The expression is evaluated before the comparison is made. (See LET verb for more information.)

Note An *arithmetic-expression* must be enclosed in square brackets ([ ]).

EXCLAMATION — Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. (See "Data Entry Control Characters" in Chapter 5.) If the null response is set, the EXCLAMATION test variable is a positive integer; if not set, it is zero. The default is 0.

FIELD	Current status of FIELD command qualifier. If a user qualifies a command with FIELD, the FIELD test variable is a positive integer. Otherwise, it is a negative integer. The default is <0.
INPUT	The last value input in response to the INPUT prompt.
PRINT	Current status of PRINT or TPRINT command qualifier. If the 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. The default is < 0.
REPEAT	Current status of REPEAT command qualifier. If the user qualifies a command with REPEAT, the REPEAT test variable is a positive integer; otherwise, REPEAT is a negative integer. The default is < 0.
SORT	Current status of SORT command qualifier. If the user qualifies a command with SORT, the value of the SORT test variable is a positive integer; otherwise SORT is a negative integer. The default is < 0.
STATUS	The value of the 32-bit 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 can 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 Any of test variable values or the value against which the *test-variable* is compared. The value may be an arithmetic expression, which will be evaluated before the comparison is made. The allowed value depends on the test variable, as shown in the comparison below. Alphanumeric strings must be enclosed in quotation marks.

If the *test-variable* is:	The *value* must be:
item-name	An alphanumeric string, a numeric value, an arithmetic expression, a reference to a variable as in (*item-name*) or a class condition as described below.
[*arithmetic expression*]	A numeric value, an arithmetic expression, or an expression, or a reference to a variable as in (*item-name*). (See the LET verb for more information.)
INPUT	An alphanumeric string.

EXCLAMATION FIELD PRINT REPEAT SORT	A positive or negative integer, or an expression.
STATUS	A 32-bit integer or an expression.

If more than one value is given, then:

- The *relational-operator* can be only "=" or "<>".
- When the relational operator is "=", 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.
- When the relational operator is "<>", the action is taken if the *test-variable* is not equal to *value1* AND *value2* AND ... *valuen*. In other words, a comma in a series of values is interpreted as an AND when the operator is "<>".

When the test variable is an *item-name*, the *value* can be one of the following class conditionals, which are used to determine whether a string is all numeric or alphabetic. The operator can only be "=" or "<>".

NUMERIC	This class condition includes the ASCII characters 0 through 9 and a single operational leading sign. Leading and trailing blanks around both the number and sign are ignored. Decimal points are not allowed in NUMERIC data. This class test is only valid when the item has the type X, U, 9, or Z, or when the item is in the input register.
ALPHABETIC	This class condition includes all ASCII native language alphabetic characters (upper and lowercase) and space. This class test is only valid for item names of type X or U.
ALPHABETIC-LOWER	This class condition includes all ASCII lowercase native language alphabetic characters and space. This class test is only valid for type X or U.
ALPHABETIC-UPPER	This class condition includes all ASCII uppercase native language alphabetic characters and space. This class test is only valid for item names of type X or U.

Order of Evaluation

When complex conditions are included, the operator precedence is:

- Arithmetic expressions are evaluated.
- Truth values are established for simple relational conditions.
- Truth values are established for simple class conditions.
- Multiple value conditions are evaluated.
- Truth values are established for complex AND conditions.
- Truth values are established for complex OR conditions.

Parentheses can be used to control the order of precedence when conditional clauses are being evaluated. In multiple value conditions, evaluation terminates as soon as a truth value is determined.

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;
```

The following are two examples of using the REPEAT verb:

```
REPEAT
   DO
      LET (TOTAL-OVERDUE) = (TOTAL-OVERDUE) + (AMT-OVERDUE);
      FIND(SERIAL) CUST-INVOICE,STATUS;
   DOEND
UNTIL (TOTAL-OVERDUE) > 999999.99 OR
      (TOTAL-OVERDUE) > (MIN-OVERDUE) AND
      (CUST-CODE) = "NEW";

REPEAT
   FIND(SERIAL) STK-ON-HAND,STATUS
UNTIL ((WEIGHT) > [(KILO-PER-METER) * (METERS)] AND
       (METERS) > (MIN-LENGTH) OR
       (PRICE) > [(UNIT-PRICE) * (KILO-PER-METER) * (METERS)]);
```

REPLACE

Changes the values contained in a KSAM or MPE record or a data set entry.

Syntax

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

REPLACE allows you to replace one or more records or entries in a file or data set. REPLACE uses 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 a 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 only replaces key (search) items in a manual master set if there are no detail set entries linked to that key. It does not replace detail set entries with search items that do not exist in manual master sets associated with that detail.

The REPLACE operation does 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 a 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.

To use REPLACE effectively, do the followings:

1. Specify the entries to update. Set up the key and argument registers if you are using 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 item in a master set, then delete all chains linked to that item from associated detail sets.

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 can get the new values from a user with a DATA(UPDATE) or PROMPT(UPDATE) statement, or you can 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 or data 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.

Note Before using REPLACE, you must first set the SYSTEM statement access mode to "UPDATE."

REPLACE 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.

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 with a new entry. Thus, for this option, only update items, not the whole record, need be present in the list register.

If you are performing dynamic transactions (Transact/iX only), be aware that transactions have a length limit. For a discussion about how REPLACE is affected by this limitation, see "Limitations" under "Dynamic Roll-back" in Chapter 6.

Note After the first retrieval, Transact uses an asterisk (*) for the call list to optimize subsequent retrievals of that data set.

Statement Parts

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

none — Updates an entry in a master set based on the key value in the argument register; this option does not use the match register. If the manual master key is to be changed, there must not be any entries in detail sets linked to the old manual master key item.

CHAIN — Updates entries in a 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 a manual master set, the new item must exist in the associated master set.

CURRENT
: Updates 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
: Updates the entry stored at the specified record number. The entry may not be defined as a child item. Before using this modifier, you must store the record number as a 32-bit integer I(10,,4) in the item referenced by the RECNO option.

PRIMARY
: Updates the 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
: Updates entries in a 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
: Updates 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
: Updates 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 the master set. This modifier forces the UPDATE option on a master set if you are not matching on key items.

file-name
: The KSAM or MPE file or the data set to be accessed by 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*])
  : Suppresses 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. The *item-name* must be a parent.

    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 Chapter 7.

  LIST=(*range-list*)
  : The list of items from the list register to be used for the REPLACE operation. For data sets, no child items can be specified in the range list.

If the LIST= option is omitted with any modifier, all the items named in the list register are used.

When the LIST= option is used, only the items specified in a LIST= option have their match conditions applied when the items are included in the match register. When the LIST= option is omitted, items which appear in the list register and the match register have their match conditions applied. Otherwise, the match conditions for an item are ignored.

The match register can be used only with the modifiers CHAIN, RCHAIN, SERIAL, or RSERIAL.

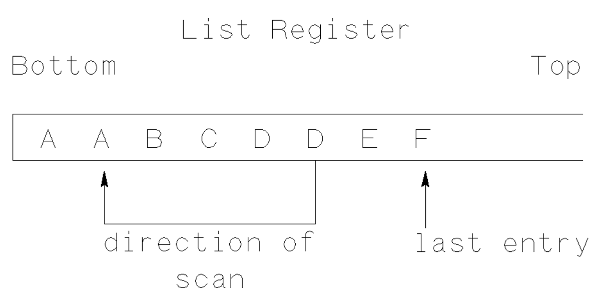
All item names specified must be parent items.

The options for *range-list* include the following:

(*item-name*) A single data item.

(*item-nameX: item-nameY*) All the data items range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items (see the DEFINE(ITEM) verb), and if there are no data items between the two on the list register, no database access is performed.

(*item-nameX:*)	All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.
(:*item-nameY*)	All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.
(*item-nameX,* *item-nameY,* ... *item-nameZ*)	The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files, data items must be specified in the order of their occurrence in the physical record. This order need not match the order of the data items on the list register. Does not include child items in the list unless they are associated with a VPLUS forms file. This option incurs some system overhead.
(@)	Specifies a range of all data items of *file-name* as defined in a data dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.
(#)	Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the dictionary. This order need not match the order of the data items in the list register.
( )	A null data item list. That is, accesses the file or data set, but does not transfer any data.

LOCK

Locks the specified file or database. If a data set is being accessed, the lock is set the entire time that REPLACE executes. If the LOCK option is not specified but the database is opened in mode 1, the lock specified by the type of automatic locking in effect is active while the entry is processed by any PERFORM= statements, but is unlocked briefly before the next entry is retrieved.

For a KSAM or MPE file, if LOCK is not specified on REPLACE but is specified for the file in the SYSTEM statement, then the file 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.

Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the REPLACE verb.

A database opened in mode 1 must be locked while REPLACE executes. For transaction locking, you can use

	the LOCK option on the LOGTRAN verb instead of the LOCK option on REPLACE if SET(OPTION) NOLOCK is specified. If a lock is not specified (for a database opened in mode 1) an error is returned. See "Database and File Locking" in Chapter 6 for more information.
NOCOUNT	Suppresses the message normally generated by Transact to indicate the number of updated entries.
NOMATCH	Ignores any match criteria set up in the match register.
NOMSG	Suppresses the standard error message produced by Transact as a result of a file or database error.
PERFORM=*label*	Executes the code following the specified label for every entry retrieved by the REPLACE verb before replacing the values in the entry. The entries can 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 can nest up to 10 PERFORM= options. The use of PERFORM forces application of the UPDATE option on master sets.
RECNO= *item-name* [(*subscript*)]	The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.) With the DIRECT modifier, you must define *item-name* to contain the 32-bit integer number I(10,,4) of the record to be updated. With other modifiers, Transact returns the record number of the replaced record in the 32-bit integer I(10,,4) *item-name*.
SINGLE	Updates only the first selected entry, and then proceeds with the statement following REPLACE.
SOPT	Suppresses Transact optimization of database calls. This option is primarily intended to support a database operation in a performed routine that is called recursively. The option allows a different path to the same detail set to be used at each recursive entry, rather than optimizing to the same path. It also suppresses generation of a call list of "*" after the first call is made. Use SOPT if you are calling TurboIMAGE through the PROC or CALL verbs. For an example of how SOPT is used, see "Examples" at the end of the FIND verb description. For more detailed information about SOPT, see "Suppression of Optimization versus WORKFILE" under the FIND verb in this chapter.

STATUS

Suppresses the actions defined in Chapter 7 under "Automatic Error Handling." Use of this option requires that you program your own error handling procedures.

When STATUS is specified, the effect of a REPLACE statement is described by the value in the 32-bit 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 the database or MPE/KSAM file system error documentation that corresponds 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). (See the example below.)

UPDATE

When this option is used with Transact/iX versions that are prior to A.04.00, 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 a manual master set. Otherwise, a DUPLICATE KEY IN MASTER error occurs when REPLACE adds the new entry.

When UPDATE is used on Transact/iX versions A.04.00 and later and the database is enabled for critical item update, search and sort items are updated. If critical item update is not enabled, UPDATE operates as it did prior to version A.04.00. See the *TurboIMAGE/XL Database Management System Reference Manual* for more information.

Examples

The first example replaces a search item value in a 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 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;
```

The following example replaces each occurrence of a non-key item, ZIP, with a new value. It asks the 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 of specified>>
     LIST=(LAST-NAME:ZIP),        <<zip code, a non-key item.           >>
     UPDATE,
     PERFORM=GET-ZIP;
EXIT;

GET-ZIP:
    DISPLAY;
    DATA(UPDATE) ZIP ("Enter new ZIP code");
    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 the associated detail set>>
```

```
DISPLAY "Updating product number in PROD-DETL", LINE2;
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=();
```

RESET

Resets execution control parameters, the match or update registers, the list register stack pointer, or delimiter values.

Syntax

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	Clears user responses from the input buffer. (See Syntax Option 1.)
DELIMITER	Resets delimiter values to Transact defaults. (See Syntax Option 2.)
LANGUAGE	Resets any SET(LANGUAGE) commands issued in the program.
OPTION	Resets various execution control parameters or the match and update registers. (See Syntax Option 3.)
PROPER	Resets delimiters for upshifting the next letter. (See Syntax Option 4.)
STACK	Resets the stack pointer for the list register. (See Syntax Option 5.)

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 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 program is run, suppose the prompt and response are:

```
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 "=". (See "Field Delimiters" in Chapter 5.)

(3) RESET(OPTION) *option-list*;

RESET(OPTION) is used to reset any options that have been changed by means of the SET verb. It is also used to reset the match and update registers.

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

AUTOLOAD	Resets the AUTOLOAD option. Forms are not automatically loaded into local form storage before they are displayed.
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. See SET(OPTION) in the SET verb description for more information.
FORMSTORE= *(form-store-list)*	Unloads the VPLUS forms in *form-store-list* from the local form storage memory of a forms caching terminal. *Form-store-list* can either be a list of VPLUS forms separated by commas. Or, it can be the name, enclosed in an additional set of parentheses, of a data item containing such a list. Forms belonging to different families can appear in the same list. To use local form storage, you must include the FSTORESIZE parameter in the SYSTEM verb. (See the FSTORESIZE parameter in the SYSTEM verb entry in this chapter.) The RESET(OPTION) FORMSTORE statement should only be used when lookahead loading is disabled and only to make room in local storage for new forms. For example, if you know that one form is significantly larger than the others and is not used later in the program, you can explicitly unload it to make room for new forms, rather than relying on lookahead loading to choose the best form to unload. The RESET(OPTION) FORMSTORE statement is not required in any other situation. Chapter 5 contains more information about the SET(OPTION) FORMSTORE statement under "Local Form Storage". The following example unloads four forms:

```
RESET(OPTION) FORMSTORE=(MENU,ADDPROD,CHGPROD,DELPROD);
```

The following commands do the same as above with a data name specified as *form-store-list*:

```
DEFINE(ITEM) FORMLIST X(40);
:
LIST FORMLIST;
MOVE (FORMLIST) = "MENU,ADDPROD,CHGPROD,DELPROD";
RESET(OPTION) FORMSTORE=((FORMLIST));
```

Note When local form storage is enabled, VPLUS automatically configures the HP 2626A and HP 2626W terminals to use datacomm port 1 and removes the HPWORD configuration from the HP 2626W terminal.

MATCH *item-list* Clears the MATCH register so that you can set up new match criteria. This option can also be used to selectively delete item entries. Here is the format you would use:

```
RESET(OPTION) MATCH [LIST({[item-name]})];
                          {     *     }
```

If there is an entry in the match register with the specified name, it will be deleted. An asterisk (*) can be used in place of the item name to delete the last entry added to the list register. In either case, if more than one such entry exists in the match register (such as multiple selection criterion in an OR chain), all will be deleted.

Only entries that were created in the current level can be deleted. The error message: `ITEM TO BE DELETED NOT FOUND IN MATCH REGISTER` is issued at run time if the item specified is not found in the set of entries for the current level.

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(OPTION) NOLOCK.

NOLOOKAHEAD Re-enables lookahead loading. VPLUS forms are loaded into local form storage according to the sequence defined in FORMSPEC. Lookahead is the default loading option for local form storage in Transact.

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 *item-list* — Clears the UPDATE register so you can set up new update parameters. This option can also be used to selectively delete item entries. Here is the format you would use:

```
RESET(OPTION) UPDATE [LIST({[item-name})];
                           {    *     }
```

If there is an entry in the update register with the specified name, it will be deleted. An asterisk (*) can be used in place of the item name to delete the last entry added to the list register.

Only entries that were created in the current level can be deleted. The error message `ITEM TO BE DELETED NOT FOUND IN UPDATE REGISTER` will be issued at run time if the item specified is not found in the set of entries for the current level.

VPLS — Indicates to Transact that the terminal is no longer in block mode. Error messages are no longer sent to the window. (See the 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(PROPER);

RESET(PROPER) resets the delimiters back to the default characters that cause the next letter to be upshifted by the PROPER function of the MOVE verb. The default set of special characters as used by PROPER function are `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~` and the blank character.

(5) 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.

(6) RESET(LANGUAGE);

RESET(LANGUAGE) resets any SET(LANGUAGE) commands issued in the program.

Examples

This example removes all current match criteria and item update values from the match and update registers.

```
RESET(OPTION)
    MATCH,
    UPDATE;
```

This example resets the list register to its beginning so you can use the same area for new list items.

```
RESET(STACK) LIST;
```

The following examples show how to use the MATCH option to delete specific items from the match register. The first example sets up the match register.

```
MOVE (name) = "Fred";
SET(MATCH) LIST(NAME);
MOVE (name) = "Bud";
SET(MATCH) LIST(NAME);
SET(MATCH) LIST(ADDRESS);
SET(MATCH) LIST(ZIP);
```

This example deletes “ADDRESS” from the match register.

```
RESET(OPTION) MATCH LIST(ADDRESS);
```

This example deletes both “NAME” entries from the match register.

```
RESET(OPTION) MATCH LIST(NAME);
```

This example causes the error message ITEM TO BE DELETED NOT FOUND IN MATCH REGISTER to be issued, because “AGE” is an item in the match register.

```
RESET(OPTION) MATCH LIST(AGE);
```

The following example shows what happens when using the UPDATE option to delete an item not added in the entries for the current level. This example will result in an error since “NAME” was not added in the current level.

```
SET(UPDATE) LIST(NAME);
LEVEL;
     SET(UPDATE) LIST(ADDRESS);
     RESET(OPTION) UPDATE LIST(NAME);

END(LEVEL);
```

The following example shows how to use the RESET(PROPER) option to reset the delimiters back to the default characters.

```
SET(PROPER) " -;,:0123456789";
:
RESET(PROPER);
```

```
MOVE (NAME) = PROPER((NAME));

                        Before         After
     NAME         X(12) 1doe's␣joe,p   1doe'S␣Joe,P

SET(PROPER) ". & -";
:
RESET(PROPER);
MOVE (LNAME) = PROPER("mr.&ms.smith-jones");

                        Before               After
     LNAME       X(18)  Mr.␣John␣Smith,jr.  Mr.&Ms.Smith-Jones
```

RETURN

Terminates a PERFORM block.

Syntax

```
RETURN [ (level) ] ;
```

RETURN transfers control from a PERFORM block to another statement. RETURN is also used to return to a database 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 database access loop called with the PERFORM 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.

Examples on the next page show how the RETURN verb works.

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 first         >>
                                  <<statement following PERFORM E;    >>
IF (VALUE)="ALLAN" THEN

   RETURN(1);                     <<Transfers control to first        >>
                                  <<statement following PERFORM D;    >>
IF (VALUE)="BROWN" THEN

   RETURN(@);                     <<Transfers control to first        >>
                                  <<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.

Syntax

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	Specifies Transact commands. (See Syntax Option 1.)
DELIMITER	Specifies Transact delimiters. (See Syntax Option 2.)
FORM	Specifies data transfer to a VPLUS form buffer for subsequent display. (See Syntax Option 3.)
KEY	Sets the value of the key and argument registers. (See Syntax Option 4.)
LANGUAGE	Specifies the native language used by Transact. (See Syntax Option 5.)
MATCH	Sets up match selection criteria in the match register. (See Syntax Option 6.)
OPTION	Specifies various execution control parameters. (See Syntax Option 7.)
PROPER	Specifies delimiters for upshifting the next letter. (See Syntax Option 8.)
STACK	Changes the value of the stack pointer for the list register. (See Syntax Option 9.)
UPDATE	Sets the value of the update register. (See Syntax Option 10.)

The DELIMITER, KEY, OPTION, and UPDATE modifiers are restored at the end of a LEVEL.

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 can 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. The maximum length of the input-string is 256 characters.

Examples of SET(COMMAND)

This statement lists all the commands in the current program and returns to the next statement.

```
SET(COMMAND) COMMAND;
```

This statement lists all the subcommands in the command sequence beginning with $$ADD and returns to the next statement.

```
SET(COMMAND) COMMAND(ADD);
```

This statement executes ADD ELEMENT until the user enters “]” or “]]”. It then returns to command mode and issues the “>” prompt for another command.

```
SET(COMMAND) "REPEAT ADD ELEMENT";
```

This statement executes the code associated with the command/subcommand:

```
SET(COMMAND) "ADD CUSTOMER";
```

and results in:

```
$$ADD:
    $CUSTOMER:
```

It does not return.

(2) SET(DELIMITER) “*delimiter-string*”;

SET(DELIMITER) replaces Transact’s input field delimiters (“,” and “=” described in Chapter 5) with the delimiter characters specified in the delimiter string. 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.

However, even though the SET(FORM) statement performs a VGETBUFFER (when there are items to transfer), the data returned from the VPLUS form buffer is not made available to the programmer. This is because the data is not directly transferred to the data register, but to an internal buffer.

Used with the LIST= option, SET(FORM) allows you to initialize fields in a VPLUS form with values from the data register rather than with values specified through FORMSPEC. The internal buffer holding the data from the VPLUS form buffer is partially or completely overlaid with data from the data register, depending on the items specified in the LIST= option. Once the overlay is complete, the VPUTBUFFER intrinsic is used to move the data back to the VPLUS form buffer.

With the inclusion of other 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. By default, Transact gets records formatted for a 264X terminal. If a different terminal is being used, a verb which opens the terminal (e.g., GET(FORM) or PUT(FORM)) should precede the SET. Information will therefore be available to tell SET to use a different format.

form — A form in the VPLUS forms file that is used for the subsequent display. It can be specified as one of the following:

form-name	Name of the form as defined by FORMSPEC.
(*item-name* [(*subscript*)])	Name of an item that contains the form name. It can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)
*	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 specification before this SET operation) and the current 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 current form will be CLEARed when the next form is displayed. This happens regardless of the FORMSPEC definitions of the two forms.

APPEND — Appends 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 — Clears 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.

CURSOR= *field-name*| *item-name* [(*subscript*)] — Positions the cursor within the specified field. The *field-name* identifies the field and the *item-name* identifies the item which names the field. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

If this option is omitted, the cursor is positioned in the form's default field.

Note

To ensure that the cursor will be positioned on the correct field, you must have a one to one correspondence between the fields defined in VPLUS. Transact determines where to position the cursor by counting the fields.

FEDIT — After transferring data to the form, perform any field edits specified in the FORMSPEC definition for the form.

FREEZE — Freezes 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 — Initializes the fields in the specified form to any initial values defined for the forms by FORMSPEC, or performs 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. The list can include child items. If this option is omitted, items that appear in both the list register and SYSTEM definition for the form are transferred.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

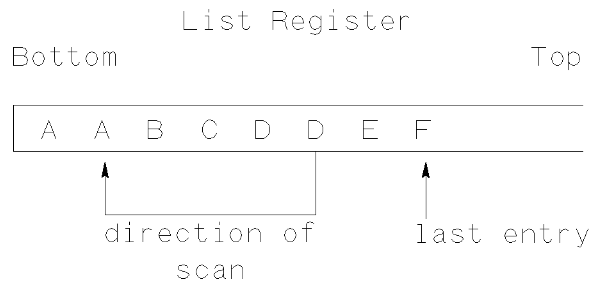
The LIST= option has a limit of 64 individually listed item names and a limit of 128 items specified by a range.

The options for *range-list* and the records upon which they operate include the following:

(*item-name*) — A single data item.

(*item-nameX*: *item-nameY*) — All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected.

(*item-nameX:*) All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.

(:*item-nameY*) All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.

(*item-nameX, item-nameY, ... item-nameZ*) The data items are selected from the list register. For VPLUS forms, data items must be specified in the order of their occurrence in the form. This order need not match the order of the data items on the list register. Child items can be included in the list as long as they are defined in the VPLUS form. This option incurs some system overhead.

(@) Specifies a range of all data items of *form* as defined in a dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.

(#) Specifies an enumeration of all data items of *form* as defined in the data dictionary. The data items are specified in the order of their occurrence in the form as defined in the dictionary. This order need not match the order of the data items in the list register.

( ) A null data item list. Does not retrieve any data.

WINDOW= ([*field*], *message*)	Places a message in the window area of the screen and, optionally, enhances 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*[(*subscript*)] within parentheses whose data register value is the name of the field to be enhanced. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)
message	Either a *"string"* of characters within quotes that comprises the message to be displayed, or an *item-name*[(*subscript*)] within parentheses whose data register value is the message string to be displayed in the window. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

Examples of SET(FORM)

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) MENU,
   CLEAR;
```

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.

```
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 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.

```
PUT(FORM) (FORMNAME), FREEZE;
SET(FORM) &,
   LIST=(ITEM-A),
   WINDOW=((ITEM-A), (MESSAGE));
PUT(FORM) *,
   WAIT=F1;
```

(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.

Examples of SET(KEY)

The example below 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.

```
SET(KEY) LIST(ACCT-NO);
OUTPUT(CHAIN) ORDER-DETAIL,
  LIST=(ACCT-NO:QTY-ON-HAND);
```

(5) SET(LANGUAGE) [*language*[,STATUS]];

The SET(LANGUAGE) statement allows the programmer to specify or change the native language at run time. The user can either specify a literal language name or number in quotes (which is checked at compile time) or give the name of an item which will contain the language number at run time. This item must begin on a 32-bit storage boundary. It can be subscripted if an array item is being referenced.

If the operation is successful, Transact sets the status register to the number of the language in effect before the language is changed. If an error results, Transact returns the error message to the user, sets the status register to −1, and leaves the native language unchanged. If STATUS is specified, Transact suppresses the error message, and the contents of the status register is the same as described above.

If you omit *language*, Transact sets the status register to the number of the current language and then resets the language number to 0 (NATIVE-3000). A compiler error results if the STATUS option is specified without *language*. For more information see Appendix E, "Native Language Support."

This is a spacer page added to retain proper pagination.

This is the end of the spacer page.

```
MOVE (STATE) = "CA";
SET(MATCH) LIST(STATE);
MOVE (STATE) = "NM";
SET(MATCH) LIST(STATE);
LET (DATE) = 010192;
SET(MATCH) LIST(DATE), GE;

+-----------------------------------------------+
|  STATE          STATE          DATE           |
| equal to   OR  equal to   AND  greater than   |
|  "CA"           "NM"           010192         |
+-----------------------------------------------+
```

(7) SET(OPTION) *option-list*;

SET(OPTION) and one or more option fields included in *option-list* 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:

AUTOLOAD — Causes VPLUS forms to be loaded automatically into the local form storage of the terminal at the time the form is displayed if the FSTORESIZE parameter is specified in the SYSTEM statement. Chapter 5 contains more information about the AUTOLOAD option under "Local Form Storage".

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. By default, the list register is reset before the END sequence block executes. However, if a REPEAT option or command is in effect, the list register is not reset until the END block is executed.Once the END block is executed, this option is automatically reset.

FIELD[="*ab*"] — Enhances or changes the prompts for data item fields on the terminal display. (This option with no parameter has the same effect as the FIELD command qualifier, described in Chapter 5.) 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, "ˆ", 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 "<".

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 forms caching terminal user handbooks. For example:

```
SET(OPTION) FIELD= " J";
```

This statement enhances the response field with half bright inverse video.

Transact enhances (or delimits) the number of spaces that represent the field length, except in two circumstances:

- If the field being prompted for is longer than the space available on the current line on the screen, Transact only enhances (or marks with a right delimiter) the rest of the line.
- The length of the response to a DATA(MATCH) or PROMPT(MATCH) prompt cannot be known in advance, since the response might include operators and multiple values. Therefore, Transact highlights or delimits only the rest of the line.

FORMSTORE= (*form-store-list*)

Loads the VPLUS forms in *form-store-list* into local storage memory of a forms caching terminal. This reduces datacomm overhead with frequently used forms and causes the form to appear all at once instead of being painted line by line on the screen.

Form-store-list can either be a list of VPLUS forms separated by commas or the name, enclosed in an additional set of parentheses, of a data item containing such a list. The data item can be subscripted. Forms belonging to different families can appear in the same list.

To use local form storage, you must include the FSTORESIZE parameter in the SYSTEM verb. (See the FSTORESIZE parameter in the SYSTEM verb entry in this chapter.)

The RESET(OPTION) FORMSTORE statement is not required with the SET(OPTION) FORMSTORE statement. (See the explanation of the RESET(OPTION) FORMSTORE statement in the RESET verb description in this chapter. Chapter 5 contains more information about the SET(OPTION) FORMSTORE statement under "Local Form Storage".)

The following example loads four forms.

```
SET(OPTION) FORMSTORE=(MENU,ADDPROD,CHGPROD,DELPROD);
```

The following commands do the same as above with a data name specified as *form-store-list*.

```
DEFINE(ITEM) FORMLIST X(40);
  :
LIST FORMLIST;
MOVE (FORMLIST) = "MENU,ADDPROD,CHGPROD,DELPROD";
SET(OPTION) FORMSTORE=((FORMLIST));
```

Note When local form storage is enabled, VPLUS automatically configures the 2626A and 2626W terminals to use datacomm port 1 and removes the HPWORD configuration from the 2626W terminal.

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 database opened in mode 1 for a DELETE, PUT, REPLACE, or UPDATE operation. NOLOCK does not reset the LOCK option specified with a database 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 or when you are locking with the LOCK option on

the LOGTRAN verb. (See Chapter 6 for more information on locking.) The NOLOCK option is turned off when processing crosses a barrier between command sequences. Therefore, NOLOCK must be set in each command sequence to which it applies.

NOLOOKAHEAD Disables look-ahead loading, which is the default option when local form storage is used. Setting the NOLOOKAHEAD option has the effect of protecting explicitly loaded forms from being overwritten by automatically loaded forms. Chapter 5 contains more information about look-ahead loading under "Local Form Storage".

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 Chapter 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 for PWIDTH is 132 and the maximum is 152.

REPEAT Sets the REPEAT option. At execution time, Transact repeats the associated statement sequence until the user enters one of the following special characters:

] Terminates execution of the current command sequence and passes control to the first statement in the sequence. However, if there is an active SET(OPTION) END= *label*, the block introduced by *label* is executed before control is passed to the first statement of the command sequence.

]]	Terminates repeated execution of this command sequence and passes control to command mode regardless of the command level or subcommand level. However, if there is an active SET(OPTION) END= *label* statement, the block introduced by *label* is executed before control is passed to command mode. The list register is reset before the current command sequence is repeated.

The 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 in Chapter 5 under "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 "Command Qualifiers" in Chapter 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 "Command Qualifiers" in Chapter 5.)

VPLS=*item-name* [(*subscript*)]
: 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 identified by *item-name*. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

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 can then examine or change comarea fields.

A SET(OPTION) VPLS statement must be followed by a RESET(OPTION) VPLS statement before any Transact statements can be used to manipulate the forms within the same Transact system and level. Otherwise, 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 block mode 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):    <<First eight words, comarea          >>
                WINDOW-ENH X(1)   <<Right byte of eighth word           >>
                = 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)

This statement aligns the prompt character on column 25, with two blank lines between the prompt lines.

```
SET(OPTION) PALIGN25,PROMPT=2;
```

This statement sorts subsequent OUTPUT listings to the terminal. It suppresses item headings and suppresses the usually automatic “CONTINUE (Y/N)?” prompt.

```
SET(OPTION) NOHEAD,SORT,DEPTH=0;
```

(8) SET(PROPER) "*delimiter-string*";

SET(PROPER) replaces the default characters that cause the next letter to be upshifted with the delimiter characters specified in the delimiter string. This statement is used in conjunction with the PROPER function on the MOVE verb. A maximum of 256 characters can be defined as the *delimiter-string*. The double quote character (") can be made one of these delimiter characters by including 2 consecutive double quotes ("") anywhere in the *delimiter-string*. Use the RESET(PROPER) verb to reset the delimiter which was set to the default set.

Examples of SET(PROPER)

```
SET(PROPER) " -;,:""0123456789";
MOVE (NAME) = PROPER((NAME));

                    Before                   After
     NAME     X(12) 1doe's␣joe,p            1Doe's␣Joe,P

SET(PROPER) " .&";
MOVE (LNAME) = PROPER("mr.&ms.smith-jones");

                    Before                   After
     LNAME   X(18)  Mr.␣John␣Smith,jr.      Mr.&Ms.Smith-jones
```

(9) 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 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. Transact generates an error at execution time if it cannot find the data item in the list register. The scan does not move the stack pointer, which 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 the stack pointer moves down the list register, the items above the new current item are removed from 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 can choose to redefine the data register contents for the following reasons:

- To transfer values from one data item to another in a different set,
- To access subfields of a data item by adding several item names in place of the original item name, or
- 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(*);
```

The next statement moves the stack pointer back to the item PROD-NO and removes all items above it. 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.

```
SET(STACK) LIST(PROD-NO);
```

(10) 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.

Note

A child item value placed in the UPDATE register is overridden by its parent's value if the parent value was placed in the update register before it.

SYSTEM

Names the Transact program and any databases, files, or forms files that are used by the program.

Syntax

`SYSTEM` *program-name*[,*definition-list*] ;

The SYSTEM statement names the program and describes databases, 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/V stores the output from the compiler in a file called "IP*xxxxxx*" 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/V's `SYSTEM NAME>` prompt.

definition-list Description of the files or data sets used during execution. Each definition list describes a file. Within the definition list, 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*`]`
`[,[`*optlock*`][,[`*basetype*`]]]])]`
`[,`*base-name2*`[([`"*password*"`][,[`*mode*`]`
`[,[`*optlock*`][,[`*basetype*`]]]])]]` ...

base-name The name of a database used in the program. This database has the attributes described in the *TurboIMAGE/V* or *XL Database Management System Reference Manual*. The *base-name1* is termed the **home base** and any references in the program to this database must not include a base qualifier. The name of the home base is stored in the system variable $HOME.

The BASE description opens the database. The home base can be opened a second time by repeating its name in the database list in the SYSTEM statement. This feature allows two independent and concurrent access paths to the same detail 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 data set.

References to data sets in bases other than the home base must be qualified by including the name of the database in parentheses following the data set name:

set-name(*base-name*)

If one or more of the following three qualifiers are used, they must all be enclosed in parentheses.

base-type The floating-point type specification for the database. The valid types are HP3000_16 and HP3000_32.

HP3000_16 specifies that the database requires HP floating point format. HP3000_32 specifies that the database requires IEEE floating point format. If no type is specified here or as a Transact/iX compiler option, HP3000_32 is assumed by Transact/iX. For Transact/V, HP3000_16 is assumed.

password Used by Transact for opening the database. If no password is provided, at execution time Transact prompts with

`PASSWORD FOR` *base-name*`>`

If the user enters an incorrect password, Transact issues an error message and then prompts again for the password.

For Transact/iX, up to three password prompts are issued. If the password is still invalid, the program will end. In batch mode for both Transact/V and Transact/iX, if the password is invalid on the first response, the batch job ends.

mode Used by Transact for opening the database. For Transact/V, this specification overrides any mode given by the user at execution time in response to the `SYSTEM NAME>` prompt. For Transact/iX, this specification overrides a mode specified by the TRANDBMODE environment variable. The default is 1.

If dynamic transactions are being performed (Transact/iX only), DBOPEN mode 2 cannot be used.

For more information about access modes, see "Database Access" in Chapter 5.

For example, to specify the database STORE to be opened with the password "MANAGER" in mode 1:

```
SYSTEM MYPROG,
   BASESTORE("MANAGER",1);
```

optlock
: Specifies whether or not optimized database locking is to be used. It can be a value of 0 or 1. The default = 0. (See Chapter 6 for more details.)

    0 — Tells Transact to always lock unconditionally at the database level.

    1 — Tells Transact to lock conditionally at the optimum level which avoids a deadlock with other Transact programs.

`DATA=`*data-length, data-count*

data-length
: The maximum 16-bit word size of the data register. The DATA=*data-length* specifications given in a main program establish the maximum data register size used by all called programs and take precedence over any DATA=*data-length* specifications in called programs. The default is 1024 16-bit words.

data-count
: The maximum number of items allowed in the list register. The DATA=,*data-count* specifications given in a main program do not establish the number of entries in the list register used by all called programs nor does it take precedence over any "DATA=,*data-count*" specifications in called programs. Default=256 items.

`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 UPDATE 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. It is recommended that you generally use UPDATE rather than READ or WRITE as this access is required to use either the REPLACE or UPDATE statements. 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, HP3000_16, HP3000_32. (See FOPEN in *MPE or*

MPE/iX Intrinsics Manual for a detailed explanation of these options and terms.)

The default is OLD (old file), binary, no carriage control, and file equation permitted.

A temporary MPE file defined for WRITE access with the option TEMP is purged when Transact exits if Transact automatically opens and closes the file. However, it is not purged when Transact exits if the CLOSE verb is used programmatically. It is purged immediately whenever the FILE(CLOSE) verb is used.

HP3000_16 specifies that the file requires HP floating point format. HP3000_32 specifies that the file requires IEEE floating point format. If no type is specified here or as a Transact/iX compile option, HP3000_32 is assumed by Transact/iX. For Transact/V, HP3000_16 is assumed.

record-length Record length of records in file. A positive value indicates words, a negative value indicates bytes. Default is byte length required by file operation.

blocking-factor Blocking factor used to block records. The default is 1 record/block.

file-size Size of the file in records. The default is 10000 records.

extents Number of extents used by the file. The default is 10 extents.

initial-allocation Initial allocation of extents. The default is 1 extent.

file-code MPE file code for the file. The default is 0.

For example, to define a file with Read/Write access, 40 words per record, a blocking factor of 3 records per block, and a file size of 100 records:

```
SYSTEM FREC,
   FILEWORK(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;
      .
      .
      .
```

FSTORESIZE=*formstoresize*

formstoresize The number of forms allowed to be stored in the terminal, specified as a number from −1 to 255. The 2626A terminal can store up to four forms. The forms directory on the 2624B can contain up to 255 depending on the form size, the type of datacomm network, and the memory capacity of the individual terminal.

If *formstoresize* is 0 to 255, VPLUS automatically configures the 2626A and 2626W terminals to use datacomm port 1 and removes the HPWORD configuration from the 2626W terminal.

If 0 is specified, local form storage is not performed. VPLUS configures the 2626A and 2626W terminals as explained above.

If −1 is specified, no local form storage is performed. VPLUS does not change any terminal configuration, and either terminal port can be used.

If the FSTORESIZE parameter is not specified, the FORM'STOR'SIZE field in the VPLUS comarea is set to −1, so that no local form storage is performed. VPLUS does not change any terminal configuration, and either terminal port can be used. See "Local Form Storage" in Chapter 5 for more information.

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. The default is 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 the *KSAM/3000 Reference Manual* for a detailed explanation of these options and terms.)

Defaults are OLD (old file), binary, no carriage control, and file equation permitted.

OPTION=*option* — For Transact/V, either enable or disable the test facility for this program execution; *option* can be either one of the following:

- TEST — Enables the TEST facility during execution of the Transact/V program.
- NOTEST — Disables the TEST facility during execution of the Transact/V program. The default is TEST.

This option is ignored by Transact/iX.

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. The default is 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 can 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 FORM2 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 16-bit word size of the work area containing the match, update, and input registers. This work area is used by Transact/V to set up temporary values used during execution of the program. The default is 256. Transact/iX automatically allocates enough room for all temporary variables, so the work-length option has no affect on a Transact/iX program.

work-count — The maximum number of entries allowed in the work area for Transact/V. The default is 64. Transact/iX automatically allocates entries for the work area, so work-count has no effect on a Transact/iX program.

WORKFILE=*number* — Specifies the number of records in the work file. The default is 10,000 records. This option replaces the SORT=*number* option which remains available for backward compatibility.

UPDATE

Modifies a single entry in a KSAM or MPE file or in a data set, or modifies a VPLUS form.

Syntax

UPDATE [(FORM)] *destination* [,*option-list*] ;

UPDATE modifies data items that are not key search or sort items in a master or detail set entry. 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.

In versions of Transact/iX A.04.00 and later, UPDATE modifies key search or sort items in a master or detail data set entry when critical item update is enabled for the database. 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 can be retrieved from a user, or from a data set or file.

To update a non-key value with UPDATE, do the following:

1. Fetch the record or entry to update and place it in the data register. You can 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 the user. If you are getting a value from the user, a PROMPT(SET) or DATA(SET) statement is useful, since it allows the 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.

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. (See the REPLACE verb description.)

Note Before using UPDATE, you must first set the SYSTEM statement access mode to "UPDATE".

Statement Parts

FORM — Causes this verb to transfer data from the data register to a VPLUS form displayed at a VPLUS compatible terminal by PUT(FORM) or GET(FORM). If the requested form is not currently displayed on the terminal, an error results. If this modifier is not specified, the *destination* must be a data set or file.

destination — The name of a file, data set, or form to be updated.

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* can be specified as any of the following:

form-name	Name of a form to be updated by UPDATE(FORM).
(*item-name* [(*subscript*)])	Name of an item whose data register location contains the name of the form to be updated by UPDATE(FORM). The *item-name* can be subscripted if an array item is referenced. (See "Array Subscripting" in Chapter 3.)
*	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 always available. Other options, described below, can 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. For data sets, no child items can be specified in the range list. For UPDATE(FORM) only, items in the range list can be child items.

If the LIST= option is omitted with any modifier except UPDATE(FORM), all the items in the list register, and either in the SYSTEM statement or the data dictionary for the form are used.

The LIST= option should not be used when specifying an asterisk (*) as the source.

For all options of *range-list*, the data items selected are the result of scanning the data items in the list register from top to bottom, where top is the last or most recent entry. (See Chapter 4 for more information on registers.)

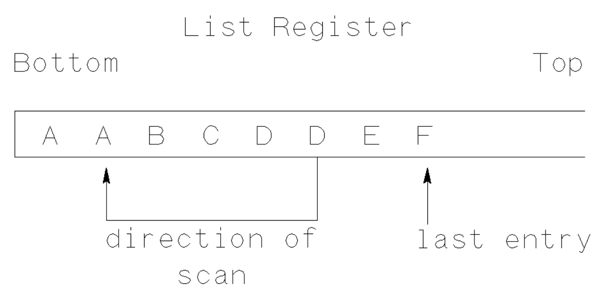
The LIST= option has a limit of 64 individually listed item names. A range limitation of 255 items for TurboIMAGE data sets and 128 items for VPLUS forms also exists.

All item names specified must be parent items when not using the FORM modifier. The options for *range-list* and the records or forms they update include the following:

(*item-name*) A single data item.

(*item-nameX:* *item-nameY*) All the data items in the range from *item-nameX* through *item-nameY*. In other words, the list register is scanned for the occurrence of *item-nameY* closest to the top of the list register. From that entry, the list register is scanned for *item-nameX*. All data items between are selected. An error is returned if *item-nameX* is between *item-nameY* and the top of the list register.

Duplicate data items can be included or excluded from the range, depending on their position on the list register. For example, if *range-list* is `A:D` and the list register is as shown,



then data items A, B, C, D, and D are selected. For database files, an error is returned if duplicate entries are selected.

If *item-nameX* and *item-nameY* are marker items (see the DEFINE(ITEM) verb), and if there are no data items between the two on the list register, no database access is performed.

(*item-nameX:*) All data items in the range from the last entry through the occurrence of *item-nameX* closest to the top of the list register.

(:*item-nameY*) All data items in the range from the occurrence of *item-nameY* closest to the top through the bottom of the list register.

(*item-nameX,* *item-nameY,* ... *item-nameZ*) The data items are selected from the list register. For databases, data items can be specified in any order. For KSAM and MPE files or for VPLUS forms, data items must be specified in the order of their occurrence in the physical record or form. This order need not match the order of the data items on the list register. Do not include child

	items in the list unless they are defined in the VPLUS form. This option incurs some system overhead.
(@)	Specifies a range of all data items of *file-name* as defined in a dictionary. The *range-list* is defined as *item-name1:item-namen* for the file.
(#)	Specifies an enumeration of all data items of *file-name* as defined in the data dictionary. The data items are specified in the order of their occurrence in the physical record or form as defined in the dictionary. This order need not match the order of the data items in the list register.
( )	A null data item list. That is, access the file or data set, but do not transfer any data.

Options Available Without the Form Modifier

ERROR=*label* ([*item-name*])
: Suppresses 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. The *item-name* must be a parent.

  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 discussion "Automatic Error Handling" in Chapter 7.

LOCK
: Locks the specified file or database for the duration of the UPDATE. For databases, if this option is not specified on UPDATE when the database has been opened with mode 1, then automatic locking will execute the lock.

  For a KSAM or MPE file, if LOCK is not specified on UPDATE but is specified for the file in the SYSTEM statement, then the file 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.

  Including the LOCK option overrides SET(OPTION) NOLOCK for the execution of the UPDATE verb.

  For transaction locking, you can use the LOCK option on the LOGTRAN verb instead of the LOCK option on UPDATE if SET(OPTION) NOLOCK is specified.

  See "Database and File Locking" in Chapter 6 for more information on locking.

NOMSG
: The standard error message produced by Transact as a result of a file or database error is to be suppressed.

STATUS

Suppresses the actions defined in Chapter 7 under "Automatic Error Handling." This option allows you to program your own error handling procedures. When STATUS is specified, the effect of an UPDATE statement is described by the value in the 32-bit integer 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 database or MPE/KSAM file system error documentation that corresponds to the value.

See "Using the STATUS Option" in Chapter 7 for details on how to use the STATUS data.

Options Available Only With the Form Modifier

APPEND

Appends 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

Clears the previously displayed form when the requested 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.

CURSOR= field-name |*item-name* [(*subscript*)]

Positions the cursor within the specified field. Field-name identifies the field and the item-name identifies the item which names the field. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

Note

To ensure that the cursor will be positioned on the correct field, you must have a one to one correspondence between the fields defined in VPLUS. Transact determines where to position the cursor by counting the fields.

FEDIT

Performs any field edits defined in the FORMSPEC definition immediately before redisplaying the form.

FKEY= *item-name* [(*subscript*)]

Moves the number of the function key pressed by the operator in this operation to a 16-bit integer I(5,,2) *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. The item name can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)

F*n*=*label*

Control passes to the labeled statement if the operator presses function key *n*. *n* can have a value of 0 through 8, inclusive, where zero indicates the ENTER key. This option can be repeated as many times as necessary in a single UPDATE(FORM) statement.

FREEZE
: Freezes the specified form on the screen and appends 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
: Initializes 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*]
: Does not return control to the program until the terminal user has pressed function key *n*. *n* can have a value of 0 through 8, where 1 through 8 indicate the keys f1 through f8 and 0 indicates the ENTER key. If F*n* is any key other than f8, the f8 exit function is disabled.

  If the 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*)
: Places a message in the window area of the screen and, optionally, enhances a field on 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*[(*subscript*)] within parentheses that will contain the data item 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*[(*subscript*)] within parentheses containing the message string to be displayed in the window.

Examples

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.

```
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);
```

The next example is similar, except that it allows the 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 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         <<Log and complete update             >>
  ELSE
     DO
       PUT LOG-FILE,
           LIST=(MARKER1:MARKER2);
       DISPLAY "key update attempted";
     DOEND;
```

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 must be numeric" in the window area of that form.

WHILE

Repeatedly tests a condition clause and executes a simple or compound statement while the condition is true.

Syntax

`WHILE` *condition-clause statement*;

WHILE causes Transact to test a *condition-clause*. The condition clause includes one or more conditions, each made up of a *test-variable*, a *relational-operator*, and one or more *values*; multiple conditions are joined by AND or OR. 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 — One or more conditions, connected by AND or OR, where

- AND — A logical conjunction. The condition clause is true if all of the conditions are true; it is false if one of the conditions is false.
- OR — A logical inclusive OR. The condition clause is true if any of the conditions is true; it is false if all of the conditions are false.

Each condition contains a *test-variable*, *relational-operator*, and one or more *values* in the following format:

test-variable relational-operator value [,*value*] ...

test-variable — Can be one or more of the following:

- (*item-name* [(*subscript*)]) — The value in the data register that corresponds to *item-name*. The *item-name* can be subscripted if an array item is being referenced. (See "Array Subscripting" in Chapter 3.)
- [*arithmetic expression*] — An arithmetic expression containing item names and/or constants. The expression is evaluated before the comparison is made. (See the LET verb for more information.)

Note An *arithmetic-expression* must be enclosed in square brackets ([ ]).

- EXCLAMATION — Current status of the automatic null response to a prompt set by a user responding with an exclamation point (!) to a prompt. (See "Data Entry Control Characters" in Chapter 5.) If the null response is set, the

	EXCLAMATION test variable is a positive integer; if not set, it is zero. The default is 0.
FIELD	Current status of FIELD command qualifier. If a user qualifies a command with FIELD, the FIELD test variable is a positive integer. Otherwise, it is a negative integer. The default is < 0.
INPUT	The last value input in response to the INPUT prompt.
PRINT	Current status of PRINT or TPRINT command qualifier. If a 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. The default is <0.
REPEAT	Current status of REPEAT command qualifier. If a user qualifies a command with REPEAT, the REPEAT test variable is a positive integer; otherwise, REPEAT is a negative integer. The default is < 0.
SORT	Current status of SORT command qualifier. If a user qualifies a command with SORT, the value of the SORT test variable is a positive integer; otherwise SORT is a negative integer. The default is < 0.
STATUS	The value of a 32-bit integer 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 can 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 value can be an arithmetic expression, which will be evaluated before the comparison is made. The allowed value depends on the test variable, as shown in the comparison below. Alphanumeric strings must be enclosed in quotation marks.

If the *test-variable* is:	The *value* must be:

item-name An alphanumeric string, a numeric value, an arithmetic expression, a reference to a variable as in (*item-name*), or a class condition as described below.

[*arithmetic expression*] A numeric value, an arithmetic expression, or an expression, or a reference to a variable as in (*item-name*).

INPUT	An alphanumeric string.
EXCLAMATION FIELD PRINT REPEAT SORT	A positive or negative integer, or an expression.
STATUS	A 32-bit integer or expression.

Alphanumeric strings must be enclosed in quotation marks. If more than one value is given, then:

- The *relational-operator* can be only "=" or "<>".
- When the relational operator is "=", 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.
- When the relational operator is "<>", the action is taken if the *test-variable* is not equal to *value1* AND *value2* AND ... *valuen*.

  In other words, a comma in a series of values is interpreted as an AND when the operator is "<>".

When the test variable is an *item-name*, the *value* can be one of the following class conditionals, which are used to determine whether a string is all numeric or alphabetic. The operator can only be "=" or "<>".

NUMERIC	This class condition includes the ASCII characters 0 through 9 and a single operational leading sign. Leading and trailing blanks around both the number and sign are ignored. Decimal points are not allowed in NUMERIC data. This class test is only valid when the item has the type X, U, 9, or Z, or when the item is in the input register.
ALPHABETIC	This class condition includes all ASCII native language alphabetic characters (upper and lowercase) and space. This class test is only valid for item names of type X or U.
ALPHABETIC-LOWER	This class condition includes all ASCII lowercase native language alphabetic characters and space. This class test is only valid for item names of type X or U.
ALPHABETIC-UPPER	This class condition includes all ASCII uppercase native language alphabetic characters and space. This class test is only valid for item names of type X or U.

statement Any simple or compound Transact statement; a compound statement is one or more statements bracketed by a DO/DOEND pair.

Order of Evaluation

When complex conditions are included, the operator precedence is:

- Arithmetic expressions are evaluated.
- Truth values are established for simple relational conditions.
- Truth values are established for simple class conditions.
- Multiple value conditions are evaluated.
- Truth values are established for complex AND conditions.
- Truth values are established for complex OR conditions.

Parentheses can be used to control the order of precedence when conditional clauses are being evaluated. In multiple value conditions, evaluation terminates as soon as a truth value is determined.

Examples

```
WHILE (SUB-TOTAL) >= 0
  DO
   GET(CHAIN) ORDERS;
    .
    .
    .
  LET (SUB-TOTAL)=(SUB-TOTAL) - (OUT-BAL);
  DOEND;

WHILE (BALANCE) < 0 AND STATUS  0
    DO
      GET(CHAIN) CUST-DETAIL,STATUS;
      LET (BALANCE) = (BALANCE) + (AMOUNT);
    DOEND;

WHILE (PART-NO-PREFIX) <> (PROTOTYPE),(DEVELOPMENT)
      GET(CHAIN) PART-DETAIL,STATUS;
```

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),
        PERFORMDISPLAY;
      .
      .
DISPLAY:
   DISPLAY "Order List by Product Number", LINE2:
            ORD-NO, NOHEAD, COL5:
            PROD-NO, NOHEAD, COL20:
            QTY-ORD, NOHEAD, COL35:
            SHIP-DATE, NOHEAD, COL50;
```

9

Running Transact

A Transact program must be compiled before it can be executed. On MPE V systems, the Transact/V compiler must convert the source code into intermediate processor code *(p-code)* which is interpreted by the Transact/V processor at run time. On MPE/iX systems, the Transact/iX compiler generates a native mode program file. The Transact/V compiler and processor may be used in compatibility mode on MPE/iX systems.

This chapter explains how to compile and run Transact programs using Transact/V and Transact/iX, including

- Compiler commands
- Program segmentation
- Reserved file names
- The Transact/V compiler
- Executing Transact/V programs
- The Transact/iX compiler
- Controlling Transact/iX program execution
- Compiling and executing Transact/iX programs
- Compiler listings

The key differences between Transact/V and Transact/iX are detailed in Appendix B, "Native Mode Transact/iX Migration Guide."

Compiler Commands

You can place any of the following commands between any two statements in the source program to control the compiled output, to conditionally compile blocks or code, or to control which data dictionary is used. Because these commands are not language statements, do not terminate them with a semicolon.

Compiler Output Commands

`!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* can be up to 500 characters long. This command can only be specified once; usually, 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. The *file-name* can be a fully qualified name with file group and account. Up to 5 files can be nested with !INCLUDE commands.
`!LIST`	Writes subsequent source statements to the list file. If LIST is specified in response to the CONTROL> prompt, !LIST has no effect.
`!NOLIST`	Suppresses 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.
`!PRECISION ([`*decimal-places*`])`	The value of *decimal-places* determines the minimum number of decimal places (i.e. digits to the right of the decimal point) that is maintained for *all* packed-decimal intermediate results when calculating the value of an arithmetic expression. This value is used only if an expression's "default" evaluation method would have maintained fewer decimal places. If *decimal-places* is not included within the parentheses, the decimal precision is reset to the "default" evaluation method. The discussion of decimal precision later in this chapter provides additional information about using this command. Also see the discussion of rounding when using packed-decimal arithmetic under the LET verb in Chapter 8.
`!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 can be up to 500 characters long. The discussion of segmentation later in this chapter tells why and how to segment programs.

Conditional Compilation Commands

There are 10 conditional compilation switches that can be set to ON or OFF by the !SET compiler command. The switches can then be queried by the !IF compiler command, and compilation of the following block of code will depend on the value of the switch. The end of the conditional block is marked by !ELSE or !ENDIF.

The following compiler commands are used to control conditional compilation:

`!SET X`*n*`={ON/OFF}`
: Sets the compilation switch to ON or OFF. The default is OFF. X*n* is any member of the set `X0, X1, X2, X3, X4, X5, X6, X7, X8, and X9.`

`!IF X`*n*`={ON/OFF}`
: Queries the named switch to determine its value. If the condition is true, the following block of code is compiled. If the condition is false, the following block is not compiled and control passes to the next !ELSE or !ENDIF.

`!ELSE`
: Marks the beginning of a block of code that will or will not be compiled, depending on the condition of the preceding `!IF`. If the condition is false, the following code is compiled. If the condition is true, the following code is not compiled. This optional command allows you to define an "either-or" situation, in which either one block of code or another is compiled, depending on the value of a switch.

`!ENDIF`
: Terminates the influence of an !IF. This command is required if an !IF is used.

Other compiler commands can occur between !IF and !ELSE or !ENDIF.

For example,

```
!SET X1=ON
 :
!IF X1=ON
DISPLAY "THIS LITERAL WILL BE DISPLAYED BECAUSE X1 IS ON";
!SET X2=OFF
!ELSE
DISPLAY "THIS LITERAL WILL BE DISPLAYED IF X1 IS OFF";
!ENDIF
```

Besides switches X0-X9, there is an eleventh switch, XL. It is set to OFF automatically when code is compiled with Transact/V and to ON when code is compiled with Transact/iX. You can test this switch with the !IF command to control compilation. For example:

```
!IF XL=ON
   SYSTEM MYPROG,             << Compile these lines if using Transact/iX. >>
      BASE=MYBASE(,,,HP3000_16),
      FILE=MYFILE((HP3000_16));

!ELSE
   SYSTEM MYPROG,             << Compile these lines if using Transact/V.  >>
      BASE=MYBASE,
      FILE=MYFILE;

!ENDIF
```

System Dictionary Compiler Commands

The default data dictionary used by Transact is Dictionary/V. If you want to access System Dictionary, use the following compiler commands:

`!SYSDIC[(`*dictionary.group.account*`)]`	Causes the compiler to use the named System Dictionary to resolve all forms files, forms, file definitions, and data items not defined in DEFINE statements. Defaults to SYSDIC in logon group and account. If System Dictionary is to be used, this command is required and it must be the first System Dictionary command included in the program.
`!NOSYSDIC`	Ends access to System Dictionary and returns to Dictionary/V.
`!DOMAIN[(`*domain*`)]`	Names the System Dictionary domain to be used. Defaults to common domain.
`!VERSIONSTATUS[(P/T/A)]`	Refers to the version to be used (production, test, or archive). Defaults to P (production version).
`!VERSION[(`*version*`)]`	Names the version to be used. Defaults to production version. This parameter overrides the VERSIONSTATUS parameter.
`!SCOPE[(`*scope*`[,"`*password*`"])]`	Names the scope and the password to be used. Defaults to DA scope and prompting for the password.

You can change System Dictionary, DOMAIN, VERSIONSTATUS, VERSION, or SCOPE in the middle of compilation by reissuing the appropriate compiler commands in the Transact source. System Dictionary compiler commands can go between statements and even within one statement—the SYSTEM statement (see example below). All of the System Dictionary commands that are used to effect a single change should appear contiguously. Comments should precede or follow the entire group of commands.

The command !NOSYSDIC causes the compiler to end access to the System Dictionary and return to using Dictionary/V for any following data items not defined in the program.

For example, if you want to change domains while extracting forms-file definitions, you can embed compiler commands in the SYSTEM statement as follows:

```
!SYSDIC(SYSDIC.PUB.SYS)
!SCOPE(Transact,"password")
SYSTEM APPL1, VPLS = FORMF1,FORMF2,        <--uses common domain
!DOMAIN(TEST)
                     FORMF3,               <--uses test domain
!NOSYSDIC
                     FORMF4;               <--uses Dictionary/V
```

Controlling Decimal Precision

The !PRECISION compiler command is used to force a minimum number of decimal places (i.e. digits to the right of the decimal point) that will be maintained for *all* packed-decimal intermediate results when calculating the value of an arithmetic expression. This value is used only if the "default" evaluation method of an expression would have maintained fewer decimal places. See the discussion of rounding when using packed-decimal arithmetic under the LET verb in Chapter 8.

The !PRECISION compiler command in effect at the end of a compiled program will be used for controlling the decimal precision used throughout the entire program. A warning is issued for each additional !PRECISION compiler command *after* the first in a source file indicating that the decimal precision is being changed. At the end of the compile, a message is issued if the "default" decimal precision is not in effect.

If a subprogram is called, the decimal precision with which that subprogram was compiled will apply while it is executing. Upon return to the calling program, the decimal precision is reset to the calling program's decimal precision.

A total of 27 digits are maintained for numbers in packed-decimal format (whole and decimal parts). Care should be taken not to request too many digits of decimal precision. The result of a multiplication temporarily uses at least double the number of decimal digits before any rounding takes place. If enough digits are not available for the whole number portion of the value, an overflow condition occurs.

Use any of the following methods as a guide for taking advantage of the !PRECISION compiler option.

- The simplest method is to select a small number (2 to 6), which should ensure reasonable decimal precision in the types of calculations used in the program. This should be sufficient for the majority of programs.
- A more precise method is to set the minimum decimal precision to at least one more than the maximum number of digits defined to the right of the decimal point for all the items and constants used in the program. You can refine this method by reviewing arithmetic expressions throughout the program.
- To take full advantage of the decimal precision available, use the following steps:
  1. Determine the maximum number of digits to the left of the decimal point that might be stored internally for *any* intermediate value created using packed-decimal arithmetic. For example, if you multiply three values together that have the following maximum values: 99, 999.9, and 9999, then the maximum number of digits (to the left of the decimal point) that the result stores is 9 (= 2 + 3 + 4). This value is referred to as LEFT.
  2. Then the maximum decimal precision that the program allows is the result of the following equation:

     ```
     (27 - LEFT) / 2          (discard any remainder)
     ```

Note

Regardless of the method you use, test the recompiled program for the possibility of overflow conditions.

Program Segmentation

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

You can solve this problem by segmenting your program. Transact allows you to divide your program into as many as 126 separate segments.

If you choose to segment your program, 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 the size needed by an entire program.

You divide a program into segments by including the !SEGMENT compiler command in your source 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, for example, when a FIND or OUTPUT statement requires a PERFORM block, the statement and the PERFORM block should be within the same segment. Program control cannot automatically cross segment boundaries, unless you specifically define entry points or use command structures.

One way to force Transact to cross segment boundaries is to use a GO TO or PERFORM statement to transfer control to a program control label in a different segment and to define that label as an entry point. Entry point labels are necessary for transfers into any segment other than 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.

Another way to control the use of segments is with command labels. When a user enters a command, control transfers to the associated command label. As far as the 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.

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. 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 can be referenced from any point.
- Any program control label or data item declared after a !SEGMENT command is local to that segment. A data item of the same name can be declared in another segment and its separate definition is maintained.

- If an item is defined in a data dictionary, but not in a DEFINE(ITEM) statement, it must be referenced in the root segment in order to be used in any segment. If the program references a child item that is defined in the data dictionary, then the parent must be referenced either in the root segment or in the same segment as that in which the child is referenced.

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.

When using local items in a segmented program, you need to explicitly clear the list, match, and update registers at the end of the segment. Transact normally checks them when it loads a new segment and issues a warning message if it finds items. It does not clear them. Furthermore, 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 any routines that are 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.

Reserved File Names

Transact uses the following files. These file names must not be used in a Transact program or in a file equation while Transact is running. Any file using the following file-name conventions could be overwritten without warning when Transact is used.

File Name	Purpose
`IP`*xxxxxx*	p-code file
`IT`*xxxxxx*	Trandebug file (version A.04.00 and earlier)
`IU`*xxxxxx*	Trandebug file (version A.04.02 and later)
`OUTPUT`	Used internally by Transact

where: *xxxxxx* is the SYSTEM name of the Transact program.

For example:

```
:file output=myfile
:tranxl myfile
```

When `tranxl` is executed, `myfile` will be overwritten.

The Transact/V Compiler

This section explains how to run the Transact/V compiler under MPE V and MPE/iX compatibility modes and describes the control options you can choose. It also describes a compiler listing, tells how you can control listings, discusses program segmentation, and describes how to control input sources to and output destinations from the compiler.

Figure 9-1 illustrates the steps used to compile and run a Transact program under MPE V.

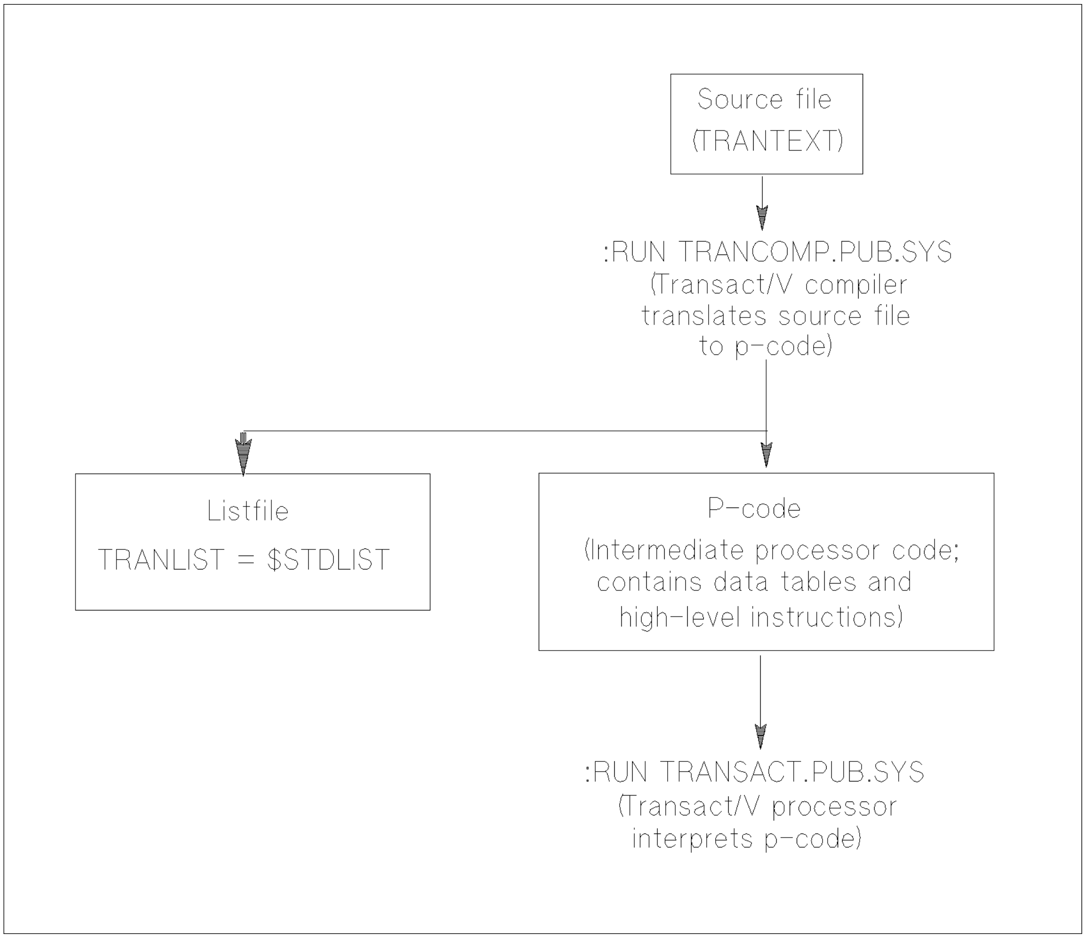


Figure 9-1. Compiling and Executing a Transact Program under MPE V

You create Transact source programs using EDIT/3000 or another text editor. The source code file can be either numbered or unnumbered. Source statements are limited to 72 characters per line and can span multiple lines.

You request the Transact compiler to translate the source code into *p-code* with the following command:

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

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

`SOURCE FILE>` Enter the file name under which the source code was saved.

`LIST FILE>` Enter a carriage return to direct the listing to your terminal ($STDLIST). You can direct the listing to a line printer by responding with LP or you can suppress the listing altogether by responding with NULL. These are the more common responses. For other possible responses, see the discussion of "Controlling Output Destinations from the Compiler."

The compiler will then prompt you to specify which control options are to be applied to the translation:

`CONTROL>` Respond to this prompt by entering one or more of the following options separated by commas. Any option can be preceded by NO to reverse its effect.

`LIST` Generates a listing of the compiled source code. The default is LIST.

`DICT` References a data dictionary (either Dictionary/V or System Dictionary) to resolve data item definitions. The default is DICT.

When this option is in effect, Transact uses Dictionary/V by default. If you want to use System Dictionary, use the dictionary commands described earlier in this chapter.

`CODE` Creates the p-code file that is executed by the Transact processor. The p-code file is created only if no errors occur during compilation. (See option XERR.) The default is CODE.

`ERRS` Lists compilation errors on $STDLIST, even if you direct a listing elsewhere. The default is ERRS.

`CHCK` Causes Transact to check that all items referenced have been put in the LIST register by either a LIST or PROMPT statement. A warning at the end of each segment is generated for all items that were not put in the LIST register. The default is NOCHCK.

Note

The use of the CHCK option does not guarantee that all run-time errors will be eliminated for items not in the LIST register. The compiler does not know the order of execution. This compiler option will only notify the programmer of items that are never used in a LIST or PROMPT statement within the segment the items are referenced.

`DEFN` Produces a listing of data-item definitions as part of the compiler list output. The list covers all data items defined in your source code and in a data dictionary. If LIST(AUTO) is included in your program, the compiler listing includes the name and relative list register position of each item placed in the list register.

The location of the items in the listing depends on the form of LIST(AUTO) used and on whether the program is segmented.

For LIST(AUTO) *filename*, the items are always listed right after the verb. For LIST(AUTO)@ in single segment programs, items are listed at the end of the program listing. For LIST(AUTO)@ in a multiple segment program, items are listed at the end of each segment, except that items in the root segment are listed at the end of the program. The default is NODEFN.

`OBJT` Produces a listing of the p-code. The default is NOOBJT.

`OPT@` Causes Transact *not* to store heading text, edit text, or entry/prompt text of data items that are defined in a data dictionary and are being used in the program to be compiled. This optimizes the tables in the p-code file so that the data segment stack is reduced at execution time. This option is the same as specifying OPTE, OPTH, and OPTP. If conflicting control options are specified, then the last control option is in effect. For example: OPT@,NOOPTH eliminates all text except the heading. In contrast, NOOPTH,OPT@ eliminates all text. See the option descriptions below for more information regarding the individual options. Appendix C, "Optimizing Transact/V Applications," provides additional information on this option in conjunction with data stack optimization. The default is NOOPT@.

`OPTE` Causes Transact *not* to store edit text of data items that are defined in a data dictionary and are being used in the program to be compiled. This optimizes the tables in the p-code file so that the data segment stack is reduced at execution time.

Note that the OPTE option should not be used if the edit mask from a data dictionary is needed in the program. Appendix C provides additional information on this option in conjunction with data stack optimization. The default is NOOPTE.

`OPTH` Causes Transact *not* to store heading text of data items that are defined in a data dictionary and are being used in the program to be compiled. This optimizes the tables in the p-code file so that the data segment stack is reduced at execution time.

Note that the OPTH option should not be used if the data item's heading text from a data dictionary is needed in the program. Appendix C provides additional information on this option in conjunction with data stack optimization. The default is NOOPTH.

`OPTI` Causes Transact *not* to store the text name of the data item defined using DEFINE(ITEM) with OPT option in the program.

Note that unlike OPT@, OPTE, OPTH, and OPTP options for data items defined in a data dictionary, OPTI requires OPT to be used with DEFINE(ITEM). This optimizes the tables in the p-code file so that the data segment stack is reduced at execution time. Note also that the OPTI option should not be used if the data item names are needed for prompt strings, display item headings, SET(KEY) lists, and LIST= constructs. Appendix C provides additional information on this option in conjunction with data stack optimization. The default is NOOPTI.

`OPTP`
: Causes Transact *not* to store prompt text of data items that are defined in a data dictionary and are being used in the program to be compiled. This optimizes the tables in the p-code file so that the data segment stack is reduced at execution time.

Note that the OPTP option should not be used if the data item names are needed for prompt strings and LIST= constructs. In the absence of the prompt string from a dictionary, the item name is used for prompting. Appendix C provides additional information on this option in conjunction with data stack optimization. The default is NOOPTP.

`OPTS`
: Optimizes multiple segment 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. The default is NOOPTS.

`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 C provides additional information on this option in conjunction with data stack optimization. The default is NOSTAT.

`XERR`
: Creates a p-code file even if errors are encountered in the compilation. (See the CODE option.) The default is NOXERR.

`XREF`
: Generates a listing to provide a cross-reference to locations of label definitions and their references. The default is NOXREF.

Bypassing Transact/V Compiler Prompts

Two RUN command options can be used to bypass the Transact compiler prompts. These are the PARM= and INFO= options that are specified in the compiler invocation statement. The PARM= option parameters identify your source file and/or list file:

Value	Formal Designator	Meaning
1	`TRANTEXT`	Formal file designator for source file. If specified, the SOURCE FILE> prompt does not appear.
2	`TRANLIST`	Formal file designator for list file. If specified, the LIST FILE> prompt does not appear. TRANLIST may be equated to any file.
3	`TRANTEXT` `TRANLIST`	If used, neither the SOURCE FILE> nor the LIST FILE> prompt appears.

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

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"
```

You can direct the compiler to a file for answers to its prompts. See "Controlling Input Sources to the Compiler" later in this chapter. 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 V commands:

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

Controlling Input Sources to the Transact/V Compiler

TRANIN is the formal file designator that TRANCOMP uses when compiling with Transact/V for responses to prompts such as system name, options, and list. The default setting for TRANIN is $STDINX, but you can change the default using a file equation. 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 EDIT/V 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 Transact/V Compiler

TRANLIST is the formal file designator for the destination of compiler listings when you set PARM=2 for the Transact compiler. When LP is the response to the LIST FILE> prompt, the default device for TRANLIST is LP. You can, however, use a file equation to change the device. A file equation or the destination default is activated when you respond to the LIST FILE> prompt with LP.

If you simply want to redirect your compiler listing and no other compiler output, you can respond to the LIST FILE> prompt with any of the following:

- A carriage return or $STDLIST directs the compiler listing to the terminal in a session or 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 batch job if ERRS is specified, but other parts of the listing are suppressed.
- $NULL directs the listing to a null file, in effect suppressing the listing. (The preferred response is NULL.)
- A file name directs the listing to a new disk 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.

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>, the compiler listing, and any requested statistics or data item definitions appear on that device. (Note that TRANOUT also controls processor output, including the SYSTEM NAME> prompt.)

TRANCODE is the name of the *p-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
```

To direct the compiled program to another group, use:

```
:FILE TRANCODE=TRANCODE.GROUP
```

Executing Transact/V Programs

This section describes how to execute Transact programs and explains how to control input to and output from the Transact processor.

Transact programs are executed (the p-code is interpreted) by running the Transact processor with the MPE V RUN command:

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

After an acknowledgement message, Transact issues the following prompt:

```
SYSTEM NAME>
```

Respond by entering the program's name as specified in the SYSTEM statement of the program you want to execute. In addition to this required response, you can specify one or more optional responses separated by commas. These optional responses specify the mode with which you want to open a database, 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 — The name of the program as it appears in the SYSTEM statement in the source program (required).

mode — The mode to be used in opening any databases specified in the program. The mode consists of a single digit indicating one of the open modes to be specified for DBOPEN. If you do not specify a mode here or in the SYSTEM statement of your program, Transact opens the databases in mode 1. Mode 1 requires locking and allows concurrent modifications to be made to a database. Any mode specified in the SYSTEM statement of the program takes precedence over a mode specified here. See the discussions in Chapter 6 on database access and understanding locking.

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 Chapter 10.)

start — The location where you want testing to begin. This is the *internal location* number of a line of processor code, optionally preceded by a segment number if it is in a segment other than segment 0. (See "Compiler Listing" in this chapter.)

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, in the format:

segment number.*end*.

For example, suppose you want to open any databases 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 a p-code file associated with the program name ("IP*xxxxxx*", where "*xxxxxx*" is the program name), it generates an error message and reissues the SYSTEM NAME> prompt. If you respond with a carriage return to the original or reissued prompt, control returns to the MPE operating system.

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

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

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

Note Unlike the programs developed and executed under MPE control, a Transact/V program can only be executed by running the Transact processor. You cannot execute a Transact p-code file with the MPE RUN command.

After it locates the p-code file, the processor generates the following prompt if databases have been defined in the SYSTEM statement and no password supplied:

PASSWORD FOR *databasename*>

You must enter the correct password to open any databases 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. Be sure to enter the password exactly as it is defined. For example, if it is defined with all uppercase letters, enter it in exactly that way.

Once your program is executing, you can redisplay the SYSTEM NAME> prompt by pressing the Ctrl Y key to stop execution and get the > prompt.

Controlling Input Sources to the Transact/V 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, use a file equation to change that. The processor will then read input from the specified file or device until it encounters an end-of-file condition. If it reaches end-of-file before all of the prompts are answered, it returns to $STDINX.

When a file equation is set for TRANIN and a Transact program calls Report, Inform, or another Transact program, TRANIN will continue to be read. (When Report or Inform is initiated from the command interpreter, it reads from REPIN or INFOIN, respectively.) The input device is only opened once at the beginning of the main Transact program. Any called system will share the input device or input file.

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. The size of this file can be altered by using the SORT= or WORKFILE= options with the SYSTEM statement.

If a larger or smaller sort file is desired after the program has been compiled, use a file equation to change the size. This will override the settings in the SYSTEM statement. For example, to reduce the sort file size to 5,000 records, use the following MPE FILE command:

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

Controlling Output Destinations from the Transact/V 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 can, 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 during a session or to the line printer during a batch 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 and 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
```

If VPLUS is used in batch mode, TRANVPLS should be directed to a device capable of handling VPLUS escape sequences or to $NULL.

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 or to the line printer in a batch job (TRANOUT). To send test mode output to another device, 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
```

You can also direct the test mode output to a disk 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 with 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 SPOOK5.PUB.SYS for MPE/V or a text editor for MPE/iX to examine the test mode information saved in a spool file.

The Transact/iX Compiler

Compiling and executing a Transact program under MPE/iX requires three sets of procedures. These sets can be accomplished one at a time by using three separate commands, or they can be combined and accomplished by using two separate commands, or even by using a single command. The commands you choose to use depend primarily on how you want to invoke subroutines or subprograms.

The three sets of procedures you can use for compiling and executing Transact programs under MPE/iX are as follows:

1. The Transact/iX compiler translates either a source code or p-code file into binary form and stores it as a Series 900 object module in a relocatable file. Note that Transact/iX can accept as input either an ASCII source file or a p-code file produced by Transact/V's TRANCOMP program. If your input is an ASCII file, the Transact/iX compiler first calls Transact/V's TRANCOMP to create p-code, then produces the relocatable file. This relocatable Series 900 object module file is called an RSOM file.
2. The MPE/iX linker must prepare the RSOM file for execution by binding procedures in the RSOM together. The linker also performs other tasks such as defining the initial requirements of the user data area. The MPE/iX LINKEDIT program may also be required at this stage if procedures external to the RSOM are to be added to the RSOM.
3. The MPE/iX operating system must allocate and initiate the execution of the program. External procedures referenced and stored in an executable library (XL) are bound to the program at this time.

You can advance through each of these procedures independently, controlling the specifics of each process along the way. In particular, it is possible to use the command TRANXL for the first set of procedures, the command LINK for the second set, and the MPE/iX command RUN *progname* for the third set. Alternatively, you can combine procedures with a single command. For example, the command file TRANXLLK performs the first and second sets of procedures; the command file TRANXLGO performs the first, second, and third sets of procedures.

You can also use the MPE/iX RUN command to execute the Transact/iX compiler, which is a program file called TRAN.PUB.SYS. This command accomplishes only the first set of procedures. It requires preceding file equations and specification of PARM values if you choose to change any defaults.

Another MPE/iX program, LINKEDIT.PUB.SYS, is required for including subprograms that are external to the RSOM file into the RSOM file.

Transact/iX Compiler Options

Like compatibility mode TRANCOMP, the Transact/iX compiler allows you to control certain compilation features by supplying compiler options via the INFO= parameter. These options can be included on any of the commands that are used to invoke the Transact/iX compiler: TRANXL, TRANXLLK, TRANXLGO, and RUN TRAN.PUB.SYS.

The Transact/iX compiler has the following options:

DYNAMIC_CALLS
: Generates dynamic calls for all CALL statements in the program. This allows a program to be executed even if some of the programs that it calls are not available at load-time. Dynamic calls are described in detail later in this chapter. The default is NODYNAMIC_CALLS.

HP3000_16
: Uses the HP floating point format for all the files and databases used by this program. If the NOHP3000_16 option is specified, then all the files are expected to use the IEEE floating point format. See the "Floating Point Formats" section in Appendix B. The default is NOHP3000_16.

PROCALIGNED_16
: Causes the compiler to assume that all 16-bit aligned parameters are correctly aligned on 16-bit boundaries and prevents it from double buffering them on 16-bit boundaries. Using this option improves run-time efficiency, since the compiler only generates a run-time check to ensure that these parameters are correctly aligned on a 16-bit boundary. Double buffering still occurs if the following conditions are all met: the procedure called is an intrinsic, the PROCINTRINSIC option is specified, and a greater than 16-bit alignment is required.

  This is the recommended option for existing Transact programs, since correct 16-bit alignment is already assured.

  The default is NOPROCALIGNED, if PROCALIGNED_16, PROCALIGNED_32, or PROCALIGNED_64 is not specified.

PROCALIGNED_32
: Causes the compiler to assume that all 16-bit and 32-bit aligned parameters are correctly aligned on 16-bit or 32-bit boundaries and prevents it from double buffering them on 16-bit or 32-bit boundaries. Using this option improves run-time efficiency, since the compiler only generates a run-time check to ensure that these parameters are correctly aligned on a 16-bit or 32-bit boundary. Double buffering still occurs if the following conditions are all met: the procedure called is an intrinsic, the PROCINTRINSIC option is specified, and a greater than 16-bit or 32-bit alignment is required.

  This option is primarily recommended for use with new Transact programs in which the correct alignment of all 16-bit and 32-bit procedure parameters is assured.

  The default is NOPROCALIGNED, if PROCALIGNED_16, PROCALIGNED_32, or PROCALIGNED_64 is not specified.

PROCALIGNED_64
: Causes the compiler to assume that all 16-bit, 32-bit, and 64-bit aligned parameters are correctly aligned on 16-bit, 32-bit, 64-bit boundaries and inhibits double buffering. Using this option improves

run-time efficiency, since the compiler only generates a run-time check to ensure that these parameters are correctly aligned on a 16-bit, 32-bit, or 64-bit boundary.

This option is primarily recommended for use with new Transact programs in which the correct alignment of all 16-bit, 32-bit, and 64-bit procedure parameters is assured.

The default is NOPROCALIGNED, if PROCALIGNED_16, PROCALIGNED_32, or PROCALIGNED_64 is not specified.

PROCINTRINSIC

Aids the migration of Transact/V systems containing intrinsic calls to MPE/iX. This option is identical in effect to declaring intrinsics within a DEFINE(INTRINSIC) statement. The Transact/iX compiler checks all procedures referenced by the PROC verb to determine whether or not they are defined in SYSINTR.PUB.SYS. When it finds an intrinsic, the compiler extracts from SYSINTR.PUB.SYS the intrinsic name (corrected for case) and the number, type, and alignment of the parameters used by the intrinsic.

This information is used at run time to set up the procedure call. If this option is not used, the Transact/iX compiler downshifts all procedure names not in the SYSINTR.PUB.SYS file (in accordance with the Series 900 procedure calling convention) and may result in the procedure not being found in the executable library (XL).

A warning message is generated each time the compiler locates an intrinsic that has not been declared in a DEFINE(INTRINSIC) statement or if it does not find an intrinsic in SYSINTR.PUB.SYS that was declared in a DEFINE(INTRINSIC) statement. PROCINTRINSIC must be used if the program calls an intrinsic and the intrinsic is not declared with DEFINE(INTRINSIC).

The default is NOPROCINTRINSIC.

TRANDEBUG

Causes TRANDEBUG, the symbolic debugger, to be enabled when the program is executed. See Chapter 11 for instructions using TRANDEBUG.

SUBPROGRAM

This option is used when compiling a program that will be called by another Transact/iX compiled program. No outer block is generated when the SUBPROGRAM option is specified. The processing that is normally done by the outer block is done by the calling program.

The Transact/iX compiler creates a single RSOM file regardless of how many SYSTEM statements are in a source file. When a source file contains more than one system, the default is to compile the first SYSTEM encountered with option NOSUBPROGRAM and those remaining with the option SUBPROGRAM as they are assumed to be subprograms called by the first system. Using the SUBPROGRAM compiler option causes all the systems in the file to be compiled with the SUBPROGRAM option.

OPTIMIZE

This option directs the compiler to generate level 1 optimized code. Using this option causes the compile to be slower but produces object modules that are more efficient at run time.

The default is NOOPTIMIZE.

TRANCOMP Options Available to the Transact/iX Compiler

In addition to the compiler options described above, the Transact/iX compiler also accepts many of the compiler options available for TRANCOMP/V, described earlier in this chapter. These are:

LIST	OPTE
ERRS	OPTP
CHCK	OPTI
DEFN	OPTS
OPT@	XREF
OPTH	

The following TRANCOMP/V options are ignored by the Transact/iX compiler. Their default values are shown to the right.

Ignored Option	Default Value
DICT	DICT
CODE	CODE
OBJT	NOOBJT
STAT	NOSTAT
XERR	NOXERR
OBJO	NOOBJO
OBJH	NOOBJH

When one of these compiler options is specified, it is ignored if it specifies the default value. If it does not specify the default value, an informational message is generated by the compiler. For example, if option NODICT were specified, the following informational message would be reported by the compiler:

```
*INFO: OPTION 'NODICT' IGNORED
```

Compiling Programs for Static Calls

The steps for compiling subprograms for use with static calls are as follows:

1. Compile the subprogram with either the :TRANXL command or :RUN TRAN.PUB.SYS. Specify the SUBPROGRAM compiler option in the INFO string.
2. Use the LINKEDIT program to build the subprogram and add it to either an RL, an XL, or the RSOM file of the calling program.

These steps are shown in the examples that follow.

The steps for compiling and executing main programs are slightly different, depending on whether the subprogram is in an RL or an XL (i.e., depending on whether it will be accessed at link time or run time). Subprograms in an XL require slightly more load time than comparable programs in an RL, but they provide the same fast run-time performance.

When subprograms are in an RL, the compilation and linking are separate steps, so that the main program can be linked to the RL at link time.

When subprograms are in an XL, the compilation and linking can be combined and done with :TRANXLLK; then when the program is executed, the RUN command must include the name of the XL file.

The steps are shown in the following examples.

Example of Static Calls with Link-Time Linking

The first example shows how to compile and run programs using static calls with link-time linking. Assume a main program called MAIN, which calls another program called PROG using static calls. First, compile the subprogram PROG into PROGOBJ using the SUBPROGRAM compiler option.

```
:TRANXL PROG, PROGOBJ; INFO="SUBPROGRAM"
```

Second, add the compiled program to an RL file using LINKEDIT/XL.

```
:RUN LINKEDIT.PUB.SYS
LinkEd> BUILDRL PROGRL
LinkEd> ADDRL PROGOBJ
LinkEd> EXIT
```

Third, compile the main program like any other Transact/iX program.

```
:TRANXL MAIN, MAINOBJ
```

Fourth, link the main program with the RL file containing the subprogram.

```
:LINK FROM=MAINOBJ; TO=MAINPROG; RL=PROGRL
```

Last, run the main program.

```
:RUN MAINPROG
```

Example of Static Calls with Load-Time Linking

This example shows how to use load-time linking with the same two programs used in the example above. The steps for the subprogram are the same, except the responses to the LINKEDIT prompts specify XL instead of RL.

First, compile the subprogram into PROGOBJ using the SUBPROGRAM compiler option.

```
:TRANXL PROG, PROGOBJ; INFO="SUBPROGRAM"
```

Second, add the compiled subprogram to an XL file using LINKEDIT/XL.

```
:RUN LINKEDIT.PUB.SYS
LinkEd> BUILDXL PROGXL
LinkEd> ADDXL PROGOBJ
LinkEd> EXIT
```

Third, compile and link the main program like any other Transact/iX program. You can combine these steps by using TRANXLLK, no special options are needed in this step.

```
:TRANXLLK MAIN, MAINPROG
```

Last, run the main program with the XL= option to name the XL file containing the subprogram.

```
:RUN MAINPROG; XL='PROGXL'
```

Dynamic Calls

No special techniques or parameters are required to compile or link a Transact compiled program which uses only dynamic calls. However, the command RUN *progname* must include the XL = option, and an XL (the preceding example shows how to create an XL) containing all the called programs must be available at run time.

Controlling Transact/iX Program Execution

Both Transact/V and Transact/iX use the formal file designator, TRANIN, at run time to respond to input prompts and database passwords. The default setting for TRANIN is $STDINX. The program reads input from TRANIN until it encounters an end-of-file condition. If it reaches the end-of-file before all prompts are answered, it returns to $STDINX for additional input.

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. The size of this file can be altered by using the SORT= or WORKFILE= options on the SYSTEM statement. If a larger or smaller sort file is desired after the program has been compiled, use a file equation to change its size. This will override the settings in the SYSTEM statement. For example, to reduce the sort file size to 5,000 records use the following MPE FILE command:

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

Transact/iX Environment Variables

Two environment variables, TRANDBMODE and TRANDEBUG, are available with Transact/iX:

TRANDBMODE

This environment variable provides a method for specifying the database open mode at run time. Transact/V allows this same feature when responding to the system prompt.

The mode consists of a single digit that indicates one of the open modes to be specified for DBOPEN. If you do not specify a mode in the SYSTEM statement of your program or use this environment variable, Transact opens the databases in mode 1. Any mode specified in the SYSTEM statement of the program takes precedence over a mode specified by this environment variable.

To use this feature, do the following:

- At the MPE/iX system prompt, set the environment variable to contain the desired open mode for the database at the time DBOPEN is called.

  ```
  :SETVAR TRANDBMODE 5
  ```

  (where 5 is the open mode)

- Run the native mode Transact program as usual.

TRANDEBUG

For Transact programs that were compiled with the TRANDEBUG option, this environment variable allows the user to disable and enable the TRANDEBUG debugger without recompiling the program. (See "Disabling the Debugger" in Chapter 11.)

Compiling and Executing Transact/iX Programs

The following MPE/iX commands are used to compile and execute Transact/iX programs:

RUN TRAN.PUB.SYS	Performs the same function as TRANXL but allows complete user control over all optional features.
TRANXL	Uses either an ASCII source file or p-code as input; produces an intermediate binary RSOM file.
TRANXLLK	Combines the functions of TRANXL and LINK.
TRANXLGO	Combines the functions of TRANXL, LINK, and RUN.
LINK	Uses an intermediate RSOM file; produces a linked program file.
LINKEDIT	Adds procedures to the RSOM file and produces a linked program file.
RUN *progname*	Executes the program.

These commands are described on the following pages, in the order shown above.

RUN TRAN.PUB.SYS

Invokes the Transact/iX compiler and produces an RSOM file.

```
RUN TRAN.PUB.SYS [;PARM=parmnum][;INFO="text"]
```

For complete syntax of the RUN command, see the *MPE/iX Commands Reference Manual*.

The Transact/iX compiler is a program file called TRAN.PUB.SYS. You can therefore use the MPE/iX command RUN to invoke it and compile your program.

When you compile with the RUN command, The default source, object, and listing files for the compiler are $STDIN, $NEWPASS, and $STDLIST, respectively. To override these default values, you must perform two steps:

1. Equate the non-default file with its formal designator using an MPE/iX FILE command;
2. Select an appropriate value for the PARM parameter of the RUN command. This value indicates which files are not defaulted.

The compiler recognizes these formal file designators:

Formal Designator	File Usage
`TRANTEXT`	Source file
`TRANOBJ`	Object file (RSOM)
`TRANLIST`	Listing file

The PARM parameter of the RUN command indicates which files appeared in the file equation. The compiler opens these files instead of the default files. The PARM parameter accepts an integer value in the range 0 ... 7. The integer value have the following meanings:

Value	Files Present in `FILE` Commands
0	none
1	source
2	listing
3	listing, source
4	object
5	object, source
6	object, listing
7	object, listing, source

An error occurs if the PARM value indicates a file for which no file equation exists. On the other hand, if a file equation exists and the PARM value doesn't indicate that file, the compiler will use the default file.

The RUN command also has an optional INFO parameter. The INFO string consists of compiler options for the Transact/iX compiler. Valid compiler options are described earlier in this chapter. The options can be arranged in any order.

TRANTEXT can be either an ASCII source file or a p-code file. When TRANTEXT contains ASCII text, TRANCOMP is called to create p-code from the source file, then it compiles the p-code.

The default size of the RSOM file is 4,000 records. For very large Transact programs, you should increase the default with an MPE/iX FILE command before compiling the program. For example, the following command increases the size of the RSOM file to 15,000 records:

```
:FILE TRANOBJ=MYSOM;DISC=15000
```

If the RSOM file size is not large enough, the following error is displayed:

```
*ERROR: error in writing to output file. (7204)
```

TRANXL

Invokes the Transact/iX compiler and produces an RSOM file.

```
TRANXL [textfile] [,[rlfile] [,[listfile]]] [;INFO = "text"]
```

The command TRANXL invokes the Transact/iX compiler and causes it to process the specified source file and generate object code to a binary file. All of the parameters of the TRANXL command are optional; their default values are given below. If you do not include an optional parameter, its default value is used automatically. TRANXL does not prompt for missing parameters.

Statement Parts

textfile The name of the input file read by the Transact/iX compiler. This can be any p-code or ASCII file. If this parameter is omitted, the file $STDIN, the current input device, is the default file. In a session, this is the terminal and you can enter source code interactively. To signal the end of source code, enter a colon (:) as the first character on a new line.

rlfile The name of the relocatable SOM (RSOM) file on which the compiler writes the object code. If this parameter is omitted, the file $NEWPASS is the default file.

listfile The name of the file on which the compiler writes the program listing. This can be any ASCII file. If this parameter is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job. If the listfile is $STDLIST, the listing is echoed back to the terminal. If the list file is $NULL or a file other than $STDLIST, the compiler displays lines with errors on $STDLIST as well as in the list file. If *textfile* is p-code, *listfile* contains only error messages.

text The text string consists of compiler options for the Transact/iX compiler. Valid compiler options are described earlier in this chapter. The options can be arranged in any order.

The default size of the RSOM file is 4,000 records. For very large Transact programs, you should increase the default with an MPE/iX BUILD command before compiling the program. For example, the following command increases the size of the RSOM file to 15,000 records.

```
:BUILD MYSOM;DISC=15000;CODE=NMOBJ
```

If the RSOM file size is not large enough, the following error is displayed:

```
*ERROR: error in writing to output file. (7204)
```

TRANXLLK

Compiles and links a source file into an executable program file.

```
TRANXLLK [textfile ][,[progfile] [,[listfile]]] [;INFO = "text"]
```

The command file TRANXLLK compiles a Transact or p-code program into an RSOM file and then links that file into a program file. All of the parameters of the TRANXLLK command are optional; the default values are given below.

Statement Parts

textfile The name of the input file that the Transact/iX compiler reads. This can be any p-code or ASCII file. If this parameter is omitted, the file $STDIN, the current input device, is the default file. In a session, this is the terminal and you can enter source code interactively. To signal the end of source code, enter a colon (:) as the first character on a new line.

progfile The name of the program file on which the linker writes the linked program. If this parameter is omitted, the file $NEWPASS is the default file.

listfile The name of the file on which the compiler writes the program listing. This can be any ASCII file. If this parameter is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job. If the listfile is $STDLIST, the listing is echoed back to the terminal. If the list file is $NULL or a file other than $STDLIST, the compiler displays lines with errors on $STDLIST as well as in the list file.

text The text string consists of compiler options for the Transact/iX compiler. Valid compiler options are described earlier in this chapter. The options can be arranged in any order.

The default size of the intermediate RSOM file, which is created by the Transact/iX compiler, is 4,000 records. This file size can not be altered when using the TRANXLLK command. You must use TRANXL or run TRAN.PUB.SYS in these cases.

TRANXLGO

Compiles, links, and executes a source file.

```
TRANXLGO [textfile] [,[listfile]] [;INFO = "text"]
```

The command file TRANXLGO compiles, links, and executes a Transact or p-code program. All of the parameters of the TRANXLGO command are optional; the default values are given below. After successful completion of TRANXLGO, the program file is in the temporary file $OLDPASS that you can save using the MPE/iX SAVE command.

Statement Parts

textfile The name of the input file that the Transact/iX compiler reads. This can be any p-code or ASCII file. If this parameter is omitted, the file $STDIN, the current input device, is the default file. In a session, this is the terminal and you can enter source code interactively. To signal the end of source code, enter a colon (:) as the first character on a new line.

listfile The name of the file on which the compiler writes the program listing. This can be any ASCII file. If this parameter is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job. If the listfile is $STDLIST, the listing is echoed back to the terminal. If the list file is $NULL or a file other than $STDLIST, the compiler displays lines with errors on $STDLIST as well as in the list file.

text The text string consists of compiler options for the Transact/iX compiler. Valid compiler options are described earlier in this chapter. The options can be arranged in any order.

The default size of the intermediate RSOM file, which is created by the Transact/iX compiler, is 4,000 records. This file size can not be altered when using the TRANXLLK command. You must use TRANXL or run TRAN.PUB.SYS in these cases.

LINK

Creates an executable program file.

```
LINK [FROM=file [,file]...] [;TO=destfile]
     [;RL=rlfile]
     [;XL=xlfile]
     [;CAP=caplist]
     [;STACK=maxstacksize]
     [;HEAP=maxheapsize]
     [;UNSAT=unsatname]
     [;PARMCHECK=integer]
     [;PRIVLEV=integer]
     [;XLEAST=integer]
     [;ENTRY=entryname]
     [;NODEBUG]
     [;NOSYM]
     [;MAP]
     [;SHOW]
```

For input, the LINK command uses the RSOM file(s) produced by the Transact/iX compiler. It prepares this binary code for execution by binding procedures together and defining the requirements for the data area.

If the program is going to be accessing a subprogram in an RL, use the RL option to name the library that contains the subprogram.

For complete documentation of the LINK command and all its parameters, see the *MPE/iX Commands Reference Manual*.

LINKEDIT

Accesses the Link Editor subsystem, where you can create program libraries and add routines to them.

```
LINKEDIT
```

When you compile and execute Transact/iX programs, the Link Editor is used to build subprograms and to add them to either an XL or RL. The Link Editor commands that are most likely to be used are

```
BUILDRL
BUILDXL
ADDRL
ADDXL
```

For a complete description of all Link Editor commands, see the *Link Editor XL Reference Manual*.

RUN progname

Executes the program file produced by the MPE/iX linker.

```
RUN progname; [XL = "xlname[, xlname, ...]"]
```

The MPE/iX RUN command executes the linked program file produced by the linker. Any external procedures referenced and stored in an executable library are bound to the program at this time.

If subprograms are stored in an XL, use the XL= option to reference the library that contains the subprograms.

For complete syntax and details, see the *MPE/iX Commands Reference Manual*.

Transact Compiler Listings

The following example shows the listing of a source program produced by the compiler using all four default control options. The three columns of numbers on the left side of the listing are described below.

```
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
```

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 Chapter 10.)

Nesting Level — Nesting level indicator that is incremented by one when the compiler encounters the start of a compound statement or a new level. It is decremented by one when the compiler reaches the end of such a compound statement or level.

The nesting level number changes at the line after the IF statement that introduces a new level. If you have trouble tracking level numbers, it helps to include DO/DOEND pairs at every level change, even though they are only required if you have compound statements. The following example shows how DO/DOEND pairs clarify the structure of a program:

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

     1.000
     2.000              SYSTEM TST04;
     3.000  0000        DEFINE(ITEM) TEMP01 I(2):
     4.000  0000                     TEMP02 I(2):
     5.000  0000                     TEMP03 I(2);
     6.000  0000        PROMPT TEMP01:TEMP02:TEMP03;
     7.000  0003        IF (TEMP01) = 1 THEN
     8.000  0003 1        DO
     9.000  0003 1        IF (TEMP02) = 1 THEN
    10.000  0006 2          DO
    11.000  0006 2          IF (TEMP03) = 1 THEN
    12.000  0009 3            GO TO OUT
    13.000  0009 3          ELSE
    14.000  0012 3            DO
    15.000  0012 3            DISPLAY "AT LEVEL 3";
    16.000  0014 3            LET (TEMP01) = 3;
    17.000  0016 3            DOEND;
    18.000  0016 2          DOEND;
    19.000  0016 1        DOEND;
    20.000  0016          IF (TEMP01) = 3 THEN
    21.000  0016 1          DO
    22.000  0016 1          DISPLAY "AT SECOND LEVEL 1"
    23.000  0019 1          DOEND;
    24.000  0021
    25.000  0021        OUT:
    26.000  0021        DISPLAY "AT THE END";
    27.000  0023        EXIT;

 CODE FILE STATUS: REPLACED

 0 COMPILATION ERRORS
 PROCESSOR TIME=00:00:02
   ELAPSED TIME=00:00:03
```

Transact Compiler Listings

The compiler listing generated by the Transact/iX and Transact/V compilers are the same with two exceptions:

- Transact/iX does not create a permanent p-code file and hence the compiler listing does not report the "CODE FILE STATUS".
- The summation of compiler warnings is provided with Transact/iX and the warning/error/compilation time message is formatted differently.

DISPLAY ITEM

Displays the value of a single item, several items, or all items in the data register.

Syntax

$$\left\{\begin{array}{l}\texttt{DISPLAY ITEM}\\ \texttt{DIT}\end{array}\right\}\left[\,\mathit{item\_name\_list}\,\right]$$

$$\mathit{item\_name\_list} = \left\{\begin{array}{l}\mathit{item1}\left[\,(\mathit{subscript})\,\right]:\mathit{item2}\left[\,(\mathit{subscript})\,\right]\\ \mathit{item1}\left[\,(\mathit{subscript})\,\right]\left[\,,\ \ldots\ \mathit{itemN}\left[\,(\mathit{subscript})\,\right]\,\right]\\ \mathit{item1}\left[\,(\mathit{subscript})\,\right]:\\ :\mathit{item2}\left[\,(\mathit{subscript})\,\right]\end{array}\right\}$$

Parameters

item — The names of the valid items or range of items in the data register that are to be displayed. These items can be child items.

subscript — A list of numerical values of the form *val1*, *val2*, ... *valn* used to select a particular element in an array. If this parameter is omitted and the item being displayed is an array, the entire array is displayed. If this parameter is specified and the item is not an array, an error message is displayed.

Discussion

This command allows you to display either selected items in the data register or all the items in the data register. The item values are converted to their ASCII equivalents prior to display. If an item cannot be successfully converted to ASCII or an overflow occurs, the item value appears as #s.

If an item cannot be displayed entirely on one line, it is formatted for multiple lines. If the items being displayed cannot fit on a single screen, a CONTINUE(Y/N)? prompt is displayed at the page breaks. If an item name is not specified, all the items in the data register are displayed.

Note Child items cannot be used when specifying a range of items to display.

Examples

The following examples display a single item, a range of items, and all items in the data register.

```
TRANDEBUG> DISPLAY ITEM item1

ITEM1:   ABCD
```

```
TRANDEBUG> DISPLAY ITEM item1:item3

ITEM1:   ABCD
ITEM2:   1234
ITEM3:   56.78

TRANDEBUG> DISPLAY ITEM

ITEM1:   ABCD
ITEM2:   1234
ITEM3:   56.78
ITEM4:   XYZ
```

INITIALIZE

The Transact/iX compiler does not support the INITIALIZE built-in command nor the INITIALIZE option of the SET(COMMAND) statement. To quit one program and begin another, you must EXIT from the first program, then invoke the next program at the MPE/iX command level.

If the INITIALIZE option is encountered during compilation, the following informational message is issued:

```
*INFO: UNSUPPORTED COMMAND: SET(COMMAND) INITIALIZE
```

If the INITIALIZE option is encountered at run time, the following error message is issued:

```
* ERROR: UNSUPPORTED COMMAND: SET(COMMAND) INITIALIZE
```

The INITIALIZE option should be replaced with a program exit. You must then specify the new program to be run at the MPE/iX command level.

Calls to Transact/V Subprograms

Calls to compatibility mode Transact/V subprograms are not supported by native mode Transact/iX.

UNLOAD and NOLOAD Options in the PROC Verb

The UNLOAD and NOLOAD options on the PROC verb are inappropriate in Transact/iX since procedures cannot be unloaded. If the UNLOAD option is encountered during compilation, an informational message is generated. If the options are encountered at run time, they are ignored. See the PROC verb in Chapter 8 for more information.

TRANIN

TRANIN is the formal file designator used by TRANCOMP for responses to prompts by the Transact/V compiler for the source file, options, and list. This file is not used for the Transact/iX native mode compiler.

TRANIN is the format file descriptor used at run time by both Transact/V and Transact/iX to respond to input prompts and database passwords. TRANIN is used differently during run time in Transact/iX. In Transact/V, the system name, database open mode, and test mode can also be included in the TRANIN file. Transact/iX does not have these additional features.

Features that Differ Between Transact/V and Transact/iX

The following features differ in usage or in effect between Transact/V and native mode Transact/iX:

- Multiple systems in one file
- Parameters passed by value or by reference in the PROC verb
- Parent and child values in SET(UPDATE)
- ALIGN option of LIST and PROMPT verbs
- Fill characters used for data type 9 with the MOVE verb
- Arithmetic operations

Multiple Systems in One File

The Transact/V compiler creates a separate p-code file for each SYSTEM statement in a source file. The native mode Transact/iX compiler creates a single RSOM file regardless of how many SYSTEM statements are in a source file. The first system is compiled as a main program and the remaining systems are compiled with the SUBPROGRAM option.

If the SUBPROGRAM option is provided in the INFO string when running the native mode Transact/iX compiler, all systems in the source file are compiled with the SUBPROGRAM option. (See "Transact/iX Compiler Options" in Chapter 9.)

Parameters Passed by Value or by Reference in the PROC Verb

Transact/V does not do type checking on passed parameters. Transact/iX checks the calls to system intrinsics to verify that reference parameters and value parameters are passed as expected. For more information see the PROC verb in Chapter 8.

Parent and Child Values in SET(UPDATE)

In Transact/V, if a parent-item value is placed in the update register before a child-item value, the parent value overrides the child value. In Transact/iX, however, the child value overrides the parent value.

ALIGN Option of LIST and PROMPT Verbs.

In Transact/V, alignment is on 16-bit word boundaries. In MPE/iX, alignment is on 32-bit word boundaries.

Fill Characters Used for Data Type 9 with the MOVE Verb

Null is the fill character used for the 9 data type in a Transact/V MOVE. In a Transact/iX MOVE, the fill character is blank.

Arithmetic Operations

If a problem occurs within a LET statement, the results obtained with Transact/V and Transact/iX may differ. For additional information see the LET verb in Chapter 8.

Migration Examples

This section contains several examples of the typical kinds of migration changes.

Data File Real Number Conversion

The following program shows the conversion of floating point (real) numbers from the MPE V format to the MPE/iX standard format. Note that the HP3000_16 option is applied to the input file and the HP3000_32 option is applied to the output file. This causes item-name R4, which is a real number, to be read as an MPE V format real number and to be written as an MPE/iX standard format real number.

```
SYSTEM CONVRT,FILE=IN(READ(HP3000_16))
             ,FILE=OUT(WRITE(HP3000_32));
DEFINE(ITEM) X2 X(2):
             I4 I(4):
             I8 I(8):
             R4 R(4);

LIST X2:I4:I8:R4;

FIND(SERIAL) IN,PERFORM=100-CONVERT;
EXIT;

100-CONVERT:
  PUT OUT;
  RETURN;
```

Procedures with Null 32 Bit Parameters

The following fragment of Transact/V code illustrates the Transact/V convention of two commas to indicate a null 32-bit parameter.

```
SYSTEM EXAM1;

DEFINE(ITEM) FILE-NAME X(20):
             FOPTION   I(4):
             AOPTION   I(4):
             FILENUM   I(4):
             BITMAP    I(4);

DEFINE(INTRINSIC) FOPEN;

LIST FILE-NAME:
     FOPTION:
     AOPTION:
     FILENUM:
     BITMAP;
```

```
   MOVE (FILE-NAME) = "OLDFILE";
   LET (FOPTION) = 5;                         <<old ascii file>>
   LET (AOPTION) = 0;                         <<read access>>
   LET (BITMAP) = 7168;                       <<1110000000000 passing the first>>
                                              <<three parameters              >>
   PROC FOPEN(%(FILE-NAME),
              #(FOPTION),
              #(AOPTION),
              ,,,,,,,,,,,                     <<note extra commas to denote null>>
              &(FILENUM),                     <<values                          >>
              #(BITMAP));
```

To modify this source program so that it is still compatible with Transact/V, you must pass the *filesize* parameter and replace the two commas currently used to denote a null filesize with the *filesize* parameter and a single comma. The code fragment for this is shown below.

```
SYSTEM EXAM1;

DEFINE(ITEM) FILE-NAME X(20):
             FOPTION   I(4):
             AOPTION   I(4):
             FILENUM   I(4):
             FILESIZE  I(9):              <<32 bit integer>>
             BITMAP    I(4);

DEFINE(INTRINSIC) FOPEN;

LIST FILE-NAME:
     FOPTION:
     AOPTION:
     FILENUM:
     FILESIZE,INIT:
     BITMAP;

MOVE (FILE-NAME) = "OLDFILE";
LET (FOPTION) = 5;                        <<old ascii file>>
LET (AOPTION) = 0;                        <<read access>>
LET (BITMAP) = 7176;                      <<1110000001000 passing the first   >>
                                          <<three parameters and filesize     >>

LET (FILESIZE) = 1023;
PROC FOPEN(%(FILE-NAME),
           #(FOPTION),
           #(AOPTION),
           ,,,,,,
           #(FILESIZE),
           ,,,                            <<each comma denotes a parameter;    >>
           &(FILENUM),                    <<note that there is 1 fewer comma   >>
           #(BITMAP));                    <<then there is in the above example.>>
```

Data Area Allocation

There are a few requirements for the *data_buffer* parameter that the calling program must address. It must allocate the entire Transact/iX data register in the calling program before the call. The data buffer must be at least as large as the data register used in the subprogram. If the buffer is smaller than the amount of bytes that are placed in the LIST and DATA register in the called Transact/iX subprogram, an error message will be issued in the subprogram.

The values placed in the data buffer by the calling program must ensure that the formats are correct for Transact/iX as listed in Table 3-3 in Chapter 3. Values placed in the *data_buffer* by the calling program should be double byte-aligned (16-bit) or the values will not be interpreted correctly by the called Transact/iX code.

The *data_length* parameter should be the same (or larger) as the size of the *data_buffer*.

Database and File Handling

When calling a Transact/iX subprogram, all the databases and files specified in the SYSTEM statement are opened, regardless of whether or not they are accessed by the calling program.

When calling a Transact/iX subprogram, it is invoked by the main program. Only the *data_buffer* is shared between the two programs. This intrinsic cannot preserve any database or file information during the call to the Transact/iX subprogram, such as current record numbers. The calling program has sole responsibility for managing these issues.

VPLUS Forms

When the called Transact/iX subprogram uses VPLUS forms, the calling program must ensure that the terminal is in character mode. The VPLUS comarea is not available to the Transact/iX subprogram when it is called from a program written in a different language. The Transact/iX subprogram always assumes that the terminal is in character mode and returns the terminal to character mode after finishing execution.

Trap Handling

During the invocation of the ACI call, arithmetic trapping is enabled for the Transact/iX subprogram with calls to HPENBLTRAP, XARITRAP, and XLIBTRAP. On returning from the called system, the arithmetic trapping is reset to the state it was in prior to calling the Transact/iX subprogram.

You should keep in mind that the trap handling in the Transact/iX subprogram may not be the same as the trap handling in the main program.

Examples

The following examples illustrate how ACI is used to call a Transact/iX subprogram from a Pascal program and from a COBOL program.

Pascal Code

```
program pastest(output);
$standard_level 'OS_FEATURES'$

type
   system_name_type = packed array[1..7] of char; {system name plus blank}
   nibble = 0..15;                                {for P types  }
   data_record = packed record                    {Data register}
      x_item     : packed array[1..8] of char;    {X(8)  }
      i4_item    : integer;                       {I(9)  }
      i2_item    : shortint;                      {I(4)  }
      nine_item  : packed array[1..6] of char;    {9(6)  }
      j4_item    : integer;                       {J(9)  }
      r4_item    : real;                          {R(6)  }
      packed_item: packed array[1..6] of nibble;  {P(5)  }
      filler     : packed array[1..2014] of char; {Rest of Data reg. used}
   end;                                           {by called subprogram. }

var
   data_buffer     : data_record;
   return_status   : integer;
   system_name     : system_name_type;

procedure tl_call_transact
   (
   var   system_name  : system_name_type;
         data_buffer  : localanyptr;
         data_length  : integer;
   var   return_status: integer
   ); external;
```

Index

D

E

F

G

H

I

J

K

L

O

P

R

S

T

U

V