HP 3000 Computer Systems

HP Transact Reference Manual

HP Part No. 32247-60003
Printed in U.S.A.

Seventh Edition
E0494

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or use of this material.

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

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright ©1981, 1982–1985, 1986, 1987, 1988, 1990, 1992, 1994 by Hewlett-Packard Company

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

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

Where to Find More Information

The following manuals and courses are recommended for additional reference or to practice using Transact.

Reference Manuals

Title	Part Number
MPE/V Commands Reference Manual	30000-90009
MPE/V Intrinsics Reference Manual	30000-90010
SPL/V Reference Manual	30000-90024
KSAM/3000 Reference Manual	30000-90079
TurboIMAGE/V Database Management System Reference Manual	32215-90050
VPLUS/3000 Reference Manual	32209-90001
Dictionary/3000 Reference Manual	32244-90001
HP System Dictionary/V User's Guide	32254-90001
HP System Dictionary/V Utilities Reference Manual	32254-90003
Report/V User's Guide	32245-90001
Inform/V User's Guide	32246-90001
Getting Started with Transact	32247-90007
MPE/iX Commands Reference Manual	32650-90003
MPE/iX Intrinsics Reference Manual	32650-90028
TurboIMAGE/XL Database Management System Reference Manual	30391-90001
HP System Dictionary/XL General Reference Manual	32256-90004
HP System Dictionary/XL Utilities Reference Manual	32256-90003

Self-Paced Courses

Title	Part Number
Using Dictionary/V	22843B
Programming in Transact	22842A

Conventions

UPPERCASE	Within syntax statements, characters in uppercase must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example:

```
SHOWJOB
```

Valid entries:	showjob	ShowJob	SHOWJOB
Invalid entries:	shojwob	ShoJob	SHOW_JOB

italics — Within syntax statements, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release:

RELEASE *filename*

punctuation — Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.

{ } — Within syntax statements, braces enclose required elements. When several elements within braces are stacked, you must select one. In the following example, you must select ON or OFF:

```
SETMSG { ON  }
       { OFF }
```

[ ] — Within syntax statements, brackets enclose optional elements. In the following example, brackets around ,TEMP indicate that the parameter and its delimiter are optional:

PURGE {*filename*} [,TEMP]

When several elements with brackets are stacked, you can select any one of the elements or none. In the following example, you can select *devicename* or *deviceclass* or neither:

SHOWDEV [*devicename*]
        [*deviceclass*]

Conventions (continued)

`[ ... ]`
: Within syntax statements, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:

  [,*itemname*][...]

  If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:

  [*itemname*][,...]

`| ... |`
: Within syntax statements, a horizontal ellipsis enclosed in parallel vertical lines indicates that you can select more than one element that appears within the immediately preceding pair of brackets or braces. However, each element can be selected only one time. In the following example, you must select `,A` or `,B` or `,A,B` or `,B,A` :

  $\left\{ \begin{array}{l} \texttt{,A} \\ \texttt{,B} \end{array} \right\} \texttt{| ... |}$

  If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, you must select `A` or `B` or `AB` or `BA`. The first element cannot be preceded by a comma:

  $\left\{ \begin{array}{l} \texttt{A} \\ \texttt{B} \end{array} \right\} \texttt{|, ... |}$

`...`
: Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.

⊔
: Within syntax statements, the space symbol ⊔ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank:

  SET[(*modifier*)]⊔(*variable*);

underlining
: User input is underlined. For example:

  PROMPT? response

  In a syntax statement, brackets, braces, or ellipses are underlined if you must enter them. For example:

  COMMAND [[*ParameterA*]] = *ParameterB*

Conventions (continued)

shading — Within an example of interactive dialog, `shaded` characters indicate user input or responses to prompts. In the following example, `OMEGA` is the user's response to the NEW NAME prompt:

```
NEW NAME? OMEGA
```

[ ] — The symbol [ ] indicates a key on the terminal's keyboard. For example, [Ctrl] indicates the Control key.

[Ctrl] *char* — [Ctrl] *char* indicates a control character. For example, [Ctrl] Y means you have to simultaneously press the Control key and the Y key on the keyboard.

base prefixes — The prefixes %, #, and $ specify the numerical base of the value that follows:

%*num* specifies an octal number.
#*num* specifies a decimal number.
$*num* specifies a hexadecimal number.

When no base is specified, decimal is assumed.

Bit (*bit:length*) — When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format bit (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, Bits (13:3) indicates bits 13, 14, and 15:

```
most significant                     least significant

|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0|  |  |  |  |  |  |  |  |  |  |  |  |13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Bit (0:1)                                 Bits(13:3)
```

Contents

1. Introduction to Transact

Transact Features and Benefits . . . 1-2
- Consistent Interface to HP File Structures . . . 1-2
- Integration with System Dictionary and Dictionary/V . . . 1-2
- Command Structure Reduces Coding Time . . . 1-2
- Development of Block-Mode or Character-Mode Applications . . . 1-2
- Examples . . . 1-3
- Built-in Debugging Facilities . . . 1-3
- Options for Producing Reports . . . 1-3
- Additional Features . . . 1-4

2. Program Structure

SYSTEM Statement . . . 2-3
DEFINE(ITEM) Statement . . . 2-3
LIST Statement . . . 2-3
END and EXIT Statements . . . 2-4
Statements . . . 2-4
- Labels . . . 2-4
- Verbs . . . 2-4
- Modifiers . . . 2-4
- Target . . . 2-5
- Option-List . . . 2-5
- Compound Statements . . . 2-5
- Statement Formatting . . . 2-6

Comments . . . 2-7
Delimiters . . . 2-7
Reserved Words . . . 2-8

3. Data Items

Data Item Definitions . . . 3-2
Data Item Names . . . 3-2
Data Item Types . . . 3-3
Data Item Sizes . . . 3-3
Data Type Compatibility . . . 3-8
- Data Types and VPLUS . . . 3-8
- Data Types and Databases . . . 3-9
- Data Types and Data Dictionaries . . . 3-9

Parent Items and Child Items . . . 3-10
Compound Items . . . 3-11
Array Subscripting . . . 3-11
Defining and Handling Arrays . . . 3-12
- Simple Arrays . . . 3-12

Complex Arrays . . . 3-14
Child Identified as Simple Items . . . 3-14
Child Items Defined as Compound Items . . . 3-15
Special Considerations . . . 3-16
Alias Items . . . 3-17
Using Dictionary Definitions with Data Item Relations . . . 3-17

4. Transact Registers
List and Data Registers . . . 4-2
Managing the List and Data Registers . . . 4-3
Key and Argument Registers . . . 4-3
Key Register . . . 4-4
Argument Register . . . 4-4
Match Register . . . 4-5
Update Register . . . 4-6
Input Register . . . 4-6
Status Register . . . 4-7
How Registers Work . . . 4-7
Verbs and Registers . . . 4-7
Sample of Transact Coding . . . 4-8

5. User Interface
Command Sequences . . . 5-2
Processing Command Sequences . . . 5-4
Command and Subcommand Labels . . . 5-4
User-Entered Passwords for Commands and Subcommands . . . 5-5
Built-in Commands . . . 5-6
Command Qualifiers . . . 5-6
DATA, INPUT, and PROMPT . . . 5-8
Responding to a MATCH Prompt . . . 5-8
VPLUS Interface . . . 5-11
Local Form Storage . . . 5-11
Look-Ahead Loading . . . 5-12
Automatic Form Loading . . . 5-12
Local Form Storage Example . . . 5-13
Special Notes . . . 5-13
Special Characters and Keys That Control Execution . . . 5-14
Control Y . . . 5-14
Data Entry Control Characters . . . 5-14
MATCH Specification Characters . . . 5-15
Field Delimiters . . . 5-15
Special Keys for Use with VPLUS Forms . . . 5-16

6. Accessing Databases and Files

Using Databases 6-2
Access Mode 6-2
Database Close 6-2
Database and File Locking 6-3
Locking Options Available with Transact 6-3
Avoiding Deadlocks in Transact Programs 6-4
Understanding the Optimized Locking Scheme 6-5
Using the LOCK Option with the Database Access Verbs 6-7
Using the LOCK Option with the LOGTRAN Statement 6-8
Dynamic Roll-Back 6-9
Locking 6-9
Examples of Locking Strategy With LOGTRAN 6-10
Limitations 6-13
DELETE Verb 6-13
REPLACE Verb 6-14
Database Unlocking 6-14
Using KSAM and MPE Files 6-15
Defining a Buffer Record 6-15
General Format for Key-Driven Access 6-16
Traversing a KSAM File by Primary Key 6-17
Traversal by Alternate Key 6-17
General Format for Generic Keys 6-18
Search with Generic Key 6-18
Simulating an Approximate Key Search 6-19
Chronological Traversal of a KSAM File 6-20
IPC Files 6-21

7. Error Handling

Automatic Error Handling 7-2
Data Entry Errors 7-2
Database or File Operation Errors 7-2
Arithmetic Calculations 7-4
Using the STATUS Option 7-5
Data Entry Errors 7-5
Database or File Operation Errors 7-6
Compiler Error Messages 7-8
Transact/V Error Message Formats 7-8
Transact/iX Error Message Formats 7-8
Error-Info 7-8
Run-time Error Messages 7-9
Error Message Format 7-9
Error-Info 7-9
Program-Name 7-10
Using EXPLAIN 7-10
Example 7-11

8. Transact Verbs

CALL . . . 8-2
CLOSE . . . 8-10
DATA . . . 8-12
DEFINE . . . 8-19
DELETE . . . 8-27
DISPLAY . . . 8-34
END . . . 8-44
EXIT . . . 8-46
FILE . . . 8-47
FIND . . . 8-51
FORMAT . . . 8-64
GET . . . 8-72
GO TO . . . 8-82
IF . . . 8-83
INPUT . . . 8-90
ITEM . . . 8-92
LET . . . 8-93
LEVEL . . . 8-113
LIST . . . 8-115
LOGTRAN . . . 8-122
MOVE . . . 8-128
OUTPUT . . . 8-144
PATH . . . 8-152
PERFORM . . . 8-157
PROC . . . 8-158
PROMPT . . . 8-171
PUT . . . 8-177
REPEAT . . . 8-186
REPLACE . . . 8-190
RESET . . . 8-199
RETURN . . . 8-205
SET . . . 8-207
SYSTEM . . . 8-223
UPDATE . . . 8-230
WHILE . . . 8-237

9. Running Transact

Compiler Commands . . . 9-2
Compiler Output Commands . . . 9-2
Conditional Compilation Commands . . . 9-2
System Dictionary Compiler Commands . . . 9-3
Program Segmentation . . . 9-5
Reserved File Names . . . 9-6
The Transact/V Compiler . . . 9-7
Bypassing Transact/V Compiler Prompts . . . 9-11
Controlling Input Sources to the Transact/V Compiler . . . 9-11
Controlling Output Destinations from the Transact/V Compiler . . . 9-12
Executing Transact/V Programs . . . 9-13
Controlling Input Sources to the Transact/V Processor . . . 9-14
Controlling Output Destinations from the Transact/V Processor . . . 9-15

The Transact/iX Compiler 9-16
Transact/iX Compiler Options 9-17
TRANCOMP Options Available to the Transact/iX Compiler 9-19
Compiling Programs for Static Calls 9-19
Example of Static Calls with Link-Time Linking 9-20
Example of Static Calls with Load-Time Linking 9-20
Dynamic Calls 9-21
Controlling Transact/iX Program Execution 9-21
Transact/iX Environment Variables 9-21
TRANDBMODE 9-21
TRANDEBUG 9-22
Compiling and Executing Transact/iX Programs 9-22
RUN TRAN.PUB.SYS 9-23
TRANXL 9-25
Statement Parts 9-25
TRANXLLK 9-26
Statement Parts 9-26
TRANXLGO 9-27
Statement Parts 9-27
LINK 9-28
LINKEDIT 9-29
RUN progname 9-30
Transact Compiler Listings 9-31
Transact Compiler Listings 9-32

10. Transact/V Test Facility
Statement Parts 10-1
TEST Output 10-2
Examples 10-7
Test Mode 1 10-8
Test Mode 3 10-9
Test Mode 4 10-10
Direct Test Output to File 10-11
Test Modes 22 through 25 10-12
Test Mode 25 10-13
Test Mode 34 10-15
Test Mode 42 10-18
Test Modes 101 and 102 10-20

11. Transact/iX Symbolic Debugger: TRANDEBUG
Overview 11-1
Features and Benefits 11-1
Symbolic Debugger 11-1
Breakpoints 11-1
Transact Display Functions 11-2
Transact Modify Functions 11-2
Program Execution Control 11-2
MPE/iX Subsystem Support 11-2
Source Code Window 11-2
TRANDEBUG Log and Command Files 11-3
Arithmetic Trapping 11-3

Control Y Trapping 11-3
Online Help 11-3
Compatibility 11-3
Using TRANDEBUG 11-4
Compiling with the TRANDEBUG Option 11-4
Starting and Ending TRANDEBUG Sessions 11-5
Displaying Source Code in TRANDEBUG 11-5
Setting a Breakpoint 11-6
Continuing Program Execution 11-8
Displaying the Values of Data Items 11-9
Modifying the Values of Data Items 11-9
Stepping Through a Program 11-9
TRANDEBUG Startup Initialization File 11-11
Redirecting VPLUS Input and Output 11-11
Disabling the Debugger 11-12
Alternative Debug Entry Points 11-12
TRANDEBUG Run-Time Environment 11-12
Arithmetic Traps 11-13
TRANDEBUG Commands 11-13
: 11-14
ABORT 11-15
AUTORPT 11-16
BREAK DELETE 11-18
BREAK LIST 11-20
BREAK SET 11-21
CONTINUE 11-24
DATA BREAK DELETE 11-25
DATA BREAK LIST 11-27
DATA BREAK REGISTER 11-28
DATA BREAK SET 11-30
DATA LOG 11-33
DEFN 11-35
DISPLAY BASE 11-36
DISPLAY CALLS 11-37
DISPLAY COMAREA 11-38
DISPLAY FILE 11-39
DISPLAY INPUT 11-40
DISPLAY ITEM 11-41
DISPLAY KEY 11-43
DISPLAY MATCH 11-44
DISPLAY PERFORMS 11-45
DISPLAY STATUS 11-46
DISPLAY STATUSDB 11-47
DISPLAY STATUSIN 11-48
DISPLAY UPDATE 11-49
EDIT 11-50
HELP 11-51
LABEL BREAK SET 11-53
LABEL JUMP 11-55
LOC 11-56
LOG 11-57

MODIFY INPUT 11-59
MODIFY ITEM 11-60
MODIFY KEY 11-61
MODIFY MATCH 11-62
MODIFY STATUS 11-64
MODIFY UPDATE 11-65
NMDEBUG 11-67
PAGE BACK 11-68
PAGE FORWARD 11-69
PAGE JUMP 11-70
PRINT 11-71
REPEAT 11-73
SORT 11-75
STEP 11-77
TPRINT 11-79
TRACE 11-81
USE 11-83
VERSION 11-85
WINDOW LENGTH 11-86
WINDOW OFF 11-87
WINDOW ON 11-88

A. Flowcharts of File and Database Operations
DELETE Charts A-2
FIND Charts A-4
GET Charts A-9
OUTPUT Charts A-14
PATH Charts A-17
PUT Charts A-18
REPLACE Charts A-21
SET Charts A-24
UPDATE Charts A-26

B. Native Mode Transact/iX Migration Guide
Exclusive Transact/iX Features B-2
Additional Compiler Options B-2
Options on the PROC Verb: Parameters Passed by Byte Address B-2
Floating Point Formats B-2
Dynamic Roll-Back B-3
Critical Item Update B-3
Symbolic Debugger B-3
Exclusive Transact/V Features B-4
MPE V-Related Compiler Options B-4
Run-Time Item Attribute Resolution (Binding) B-4
Test Modes B-4
INITIALIZE B-5
Calls to Transact/V Subprograms B-5
UNLOAD and NOLOAD Options in the PROC Verb B-5
TRANIN B-5
Features that Differ Between Transact/V and Transact/iX B-6
Multiple Systems in One File B-6

Parameters Passed by Value or by Reference in the PROC Verb B-6
Parent and Child Values in SET(UPDATE) B-6
ALIGN Option of LIST and PROMPT Verbs. B-6
Fill Characters Used for Data Type 9 with the MOVE Verb B-6
Source Program Migration B-7
Conversion B-7
Data File Migration B-8
File Format Conversion B-8
Migration Examples B-9
Data File Real Number Conversion B-9
Procedures with Null 32 Bit Parameters B-9
Migration Checklist B-11

C. Optimizing Transact Applications
Run-Time Stack C-2
Compiler Statistics C-6
Single-Segment Programs C-10
Multiple-Segment Programs C-14
Programs Using CALLs Without the SWAP Option C-20
Programs Using CALLs with the SWAP Option C-28
Stack Usage Comparison C-32
Processing Time Optimization C-33

D. Architected Call Interface (ACI)
Introduction D-1
Syntax D-1
Parameters D-1
Data Area Allocation D-3
Database and File Handling D-3
VPLUS Forms D-3
Trap Handling D-3
Examples D-4
Pascal Code D-4
Pascal Commands D-5
COBOL Code D-6
COBOL Commands D-7
Pascal Code With Status D-7

E. Native Language Support
The SET(LANGUAGE) Statement E-1
The RESET(LANGUAGE) Statement E-1
Specifying the Language for the Compiler and Processor E-2
Called Programs E-2
Numeric Input E-2
Numeric Output E-2
Date and Time E-3
IF and MATCH Changes E-3
Upshifting and Character Types E-3
Intrinsics That Support Native Languages E-3
Examples: E-3

Index

Figures

2-1. Sample Transact Program 2-2
5-1. Program Using Command Sequences 5-3
8-1. Complex Array of Sales Figures. 8-106
9-1. Compiling and Executing a Transact Program under MPE V 9-7
11-1. The TRANDEBUG Screen 11-5
11-2. TRANDEBUG Source Code Window 11-6
11-3. Sample Transact Program 11-7
C-1. Data Stack Layout for a Single-Segment Transact Program C-3
C-2. Transact Compiler Statistics C-8
C-3. Compiler Statistics Fields and Data Stack Components C-9
C-4. Compiler Statistics for a Single-Segment Program C-11
C-5. Data Stack of a Single-Segment Program C-12
C-6. Table Register Entities of a Single-Segment Program C-13
C-7. Compiler Statistics for a Multiple-Segment Program (1 of 3) C-15
C-8. Data Stack of a Multiple-Segment Program C-18
C-9. Table Register Entities of a Multiple-Segment Program C-19
C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (1 of 5) C-21
C-11. Data Stack of Program Using CALLs Without the SWAP Option C-26
C-12. Table Register Entities of Main Program Using CALLs Without the SWAP Option C-27
C-13. Data Stack of Program Using CALLs With the SWAP Option (CALLed Program is on the Stack) C-29
C-14. Table Register Subsets for Main Program After CALLing Subprogram . . . C-30
C-15. Table Register Entities of Subprogram 4 . . . . C-31

Tables

2-1. Transact Delimiters and Their Functions 2-7
3-1. Transact Data Item Types 3-3
3-2. Data Item Size 3-4
3-3. Compatibility of VPLUS and Transact Data Types 3-8
4-1. Registers Affected by Modifiers on Specific Verbs 4-8
7-1. Circumstances that Determine Whether ERROR= Branch Is Taken during Database and File Operations 7-4
7-2. Contents of Status Register After Data Entry Verbs 7-5
7-3. STATUS Option with Database and File Operation Verbs 7-6
7-4. Contents of Status Register Following Operations of Data Management Verbs when STATUS Option Is NOT USED 7-7

C-1. Example of Data Stack Requirements C-32

1

Introduction to Transact

Transact is a high-level programming language used to develop transaction processing applications. Designed as a procedural language, Transact combines the functionality of a third generation language, such as COBOL or Pascal, with a comprehensive set of powerful, high-level constructs that can perform several functions within a single statement.

Applications written in Transact require far fewer lines of code than those written in traditional third generation languages. They are not only easier to write they are also easier to understand and maintain. The result is significantly lower development and maintenance costs.

Transact/V is a compiler and intermediate code interpreter that runs on MPE V-based HP 3000 systems. Transact/iX runs on MPE/iX systems. Included in Transact/iX are the Transact/iX compiler that generates native mode object code as well as the Transact/V compiler and interpreter.

Programmers can quickly port their applications written in Transact/V to MPE/iX systems. Transact/V applications can be run on MPE/iX systems in compatibility mode. Improved performance can be achieved by compiling your application in native mode. Unlike Transact/V interpreted statements, native mode compiled Transact programs provide the user with a performance level comparable to that of traditional third generation languages such as COBOL and Pascal.

Transact Features and Benefits

Consistent Interface to HP File Structures

A major strength of Transact is its integration with file management facilities. Access to TurboIMAGE databases, MPE files, KSAM files, and FORMSPEC forms files support the creation of a wide variety of transaction-oriented applications. The language syntax provides a single interface for easy retrieval and update of all these files.

Integration with System Dictionary and Dictionary/V

Data definitions and structures can be maintained in either System Dictionary or Dictionary/V. Transact automatically resolves data and file definitions during compile through either of the dictionaries; Transact/V will dynamically resolve data definitions in Dictionary/V.

Using a dictionary eliminates the need for data definitions within the program and provides consistent data definitions across Transact applications. Dictionary entries can also be used to set default prompt text for data elements, define edit masks, and define heading labels to identify data when it is displayed.

Command Structure Reduces Coding Time

The command structure built into Transact relieves you from much of the coding usually required to design menu-driven applications. You assign a label to a block of Transact statements that accomplish a given task. The label is then identified to the end user as a command name.

Several command modifiers can be used at execution time to enhance or modify the program procedures that you set up. For example, you can direct a display to the printer, rather than to the terminal. You can also request that information be sorted before it is displayed, or you can request that a command be repeated.

Development of Block-Mode or Character-Mode Applications

The Transact integration with the VPLUS interface facilitates the use of block-mode applications. The interface dramatically reduces the amount of work necessary to communicate with VPLUS. A single statement handles the complete VPLUS interface, from opening the terminal and forms file through edit checking and data conversion. There is no need to specify any low level intrinsics. Transact also contains statements that accept data from a character-mode terminal with equal ease.

The following two sample Transact programs demonstrates the few lines of Transact code necessary to retrieve data from a user and update a database.

Examples

The first example shows the VPLUS form, PRODUCT-INFO, then waits for the user to enter data in the fields and press ENTER. The PRODUCT-INFO data set is then updated with the data entered on the screen.

```
SYSTEM ENTRY, BASE=PRODCT, VPLS=PRODFORM;
LIST(AUTO) PRODUCT-INFO;
GET(FORM) PRODUCT-SCREEN;
PUT PRODUCT-INFO;
END;
```

The second example requests input from the user one field at a time: PROD-NUM, PROD-NAME, and PROD-PRICE. The values entered for each of these prompts are then written to the PRODUCT-INFO data set.

```
SYSTEM ENTRY2, BASE=PRODCT;
LIST(AUTO) PRODUCT-INFO;
PROMPT PROD-NUM:
       PROD-NAME:
       PROD-PRICE;
PUT PRODUCT-INFO;
END;
```

Built-in Debugging Facilities

An extensive, built-in test facility aids the programmer in debugging Transact/V programs. The programmer can choose from a number of options that display different information as the program executes. TRANDEBUG is available in Transact/iX. TRANDEBUG is a full-featured symbolic debugger which provides access to program data and source code as it executes. This debugger allows for the display and modification of data, as well as complete breakpoint manipulation.

Options for Producing Reports

Transact provides two methods for generating reports. Transact can call predefined Business Report Writer (BRW), Inform/V, and Report/V procedures, or it can define reports within the Transact source code.

When defining reports within the source code, the heading and edit masks can be retrieved from either Dictionary/V or System Dictionary. Using a dictionary to define the report reduces the amount of work that you need to do. However, Transact provides the flexibility to define the layout of the report within the source code.

Additional Features

Transact has other features as well:

- Automatic file and data entry locking.
- Automatic and programmatic error handling and recovery. These techniques simplify Transact programming and help to ensure effective processing. When the processor discovers an error, it automatically returns control to the program instruction where the error most probably occurred, thus saving you from having to code error routines. You can, however, override this automatic error handling.
- Support for Dynamic Roll-back with TurboIMAGE/XL.
- Ability to call procedures written in Transact and other languages such as COBOL, FORTRAN, Pascal, and SPL as well as other Transact programs.
- Transact programs can be called from COBOL and Pascal programs.
- Ability to call system intrinsics.
- Native language support.
- The Transact/iX run-time library is included with every release of the MPE/iX operating system, allowing native mode Transact programs to be run on any MPE/iX system.

2

Program Structure

To program in the Transact language you must understand the Transact program structure, which includes the following basic elements:

- SYSTEM statement
- DEFINE(ITEM) statement
- LIST statement
- END and EXIT statements
- Statements
- Comments
- Delimiters
- Reserved words

This chapter discusses each of these elements. Figure 2-1 shows a sample Transact program with comments identifying the components of the program structure.

```
SYSTEM ORDINF, KSAM=ORDER(READ(OLD,ASCII)),       <<SYSTEM Statement>>
       SIGNON="Customer Order Information";

DEFINE(ITEM) CUST-NAME    X(20):                  <<DEFINE(ITEM) Statement>>
             ORDER-DATE   X(8):
             ORDER-NUMBER X(10):
             AMOUNT-DUE   I(5,,2);

LIST   CUST-NAME:                                 <<LIST Statement>>
       ORDER-DATE:
       ORDER-NUMBER:
       AMOUNT-DUE;

<< Request the user enter the customer name  >>   <<Comment>>
<< for the order information needed.  The    >>
<< information is printed from each record   >>
<< that matches the name entered by the user.>>

REPEAT                                            <<verb>>
  DO
<<verb modifier>>
         ↓
  DATA(MATCH) CUST-NAME ("ENTER CUSTOMER NAME ");
  FIND(SERIAL) ORDER, LIST=(CUST-NAME:AMOUNT-DUE), PERFORM=PRINT-IT;
  RESET(OPTION) MATCH LIST(CUST-NAME);
  INPUT "MORE NAMES? (Y OR N)";
  DOEND
UNTIL INPUT = "N";

EXIT;                                             <<EXIT statement>>
PRINT-IT:
  DISPLAY CUST-NAME:          <<delimiters>>
          ORDER-NUMBER:              ↓
          ORDER-DATE, EDIT="^^/^^/^^":
          AMOUNT-DUE, EDIT="$$$$$!^^";
RETURN;                                           <<end of PERFORM>>

END ORDINF;                                       <<END statement>>
```

Figure 2-1. Sample Transact Program

SYSTEM Statement

The SYSTEM statement names the Transact program and any databases, MPE and KSAM files, and VPLUS forms that are used by the program. It can also override default space allocations. The SYSTEM statement must be the first executable statement in any Transact program, but it can be preceded by comments. The example in Figure 2-1 shows a SYSTEM statement that names the program ORDINF, specifies the name and access mode for the ORDER KSAM file, and even specifies a message to identify the program when it is run. See Chapter 8 for a detailed description of the SYSTEM statement.

DEFINE(ITEM) Statement

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

The DEFINE(ITEM) statement in Figure 2-1 defines the fields in the ORDER KSAM file used in the ORDINF program.

Although DEFINE(ITEM) statements may appear anywhere in a program, it is a good practice to place any needed statements immediately after the SYSTEM statement. DEFINE(ITEM) statements that follow the SYSTEM statement define data globally to the Transact program. To define data that is local to a program segment, include the DEFINE(ITEM) statements in that segment. Program segmentation is discussed in Chapter 9. See Chapter 8 for detailed specifications for DEFINE(ITEM) statements.

LIST Statement

The list register is an integral part of any Transact program. It is manipulated by the LIST verb and functions as a map for the data storage used by the Transact application. A complete discussion of the list register can be found under “List and Data Registers” in Chapter 4.

In Figure 2-1, the items used in the program are placed in the LIST register by the LIST statement. See Chapter 8 for detailed specifications for LIST statements.

END and EXIT Statements

The END and EXIT statements are used to transfer control in a Transact program or to terminate the program. The END statement returns control to the next higher level. The EXIT statement terminates the Transact program. Both of these statements are used in Figure 2-1. For further discussion of the END and EXIT statements, see Chapter 8.

Statements

Statements perform a Transact program's data processing functions. The general format for a Transact statement is:

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

These statement parts are described below, followed by a discussion of compound statements and statement formatting. Other statement parts, including relational and arithmetic operators, are listed with the verbs to which they apply. A statement is always terminated by a semicolon. (Rules for punctuating statements are discussed in Table 2-1.)

Labels

Statement labels help to control program flow. They identify the point to which a conditional or unconditional statement should branch. A statement label can be up to 32 characters long, and it must begin with an alphabetic character. It is followed by a colon and one or more Transact statements. The program shown in Figure 2-1 illustrates the statement label PRINT-IT.

Verbs

Transact verbs are the heart of Transact statements. They are the action words for any procedure. Verbs in Figure 2-1 include LIST, DATA, FIND, REPEAT, DISPLAY, RESET, INPUT, EXIT, SYSTEM, DEFINE, RETURN, and END. Transact verbs are described in detail in Chapter 8.

Modifiers

Modifiers that change or enhance a verb's action are an integral part of the verb. Some modifiers specify how values entered by the user are used. Other modifiers describe a file access method. Modifiers are always enclosed within parentheses and must NOT be separated from the preceding verb by a space. For example:

`FIND(CHAIN) DET;` *is correct*
`FIND (CHAIN) DET;` *is NOT correct*

In Figure 2-1, the verb FIND(SERIAL) has a different function with the modifier SERIAL than it has as FIND without a modifier. See Chapter 8 for further information on the modifiers for each verb.

Target

The target identifies the program variable upon which the verb action is performed. It can also identify the file or database for a file operation. Targets used in Figure 2-1 include the KSAM file ORDER and the data item CUST-NAME.

Option-List

A list of one or more options, separated by commas, can be specified with certain verbs to enhance their action. Some options tell how information should be formatted, while others suppress regular processor operations. Examples of option-list options in Figure 2-1 are PERFORM and LIST=(CUST-NAME:AMOUNT-DUE).

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

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

The verbs that allow such multiple target/option lists include DEFINE, DISPLAY, DATA, LIST, and PROMPT; multiple target/option lists are not allowed with database or file access verbs. The DISPLAY statement in Figure 2-1 has multiple target/option lists.

Compound Statements

You can combine several Transact statements to form a compound statement. These can be either unconditional or conditional. All compound statements are bracketed between a pair of DO ... DOEND statements. Compound statements also can be nested. The following example shows an unconditional compound statement:

```
DO

   PROMPT(MATCH) CUST-NO;

   LIST NAME:
        ADDRESS:
        CITY:
        ZIP;

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

DOEND;
```

The next example shows a conditional compound statement.

```
IF (A) = (B) THEN

  DO
     LET (A) = (A) * (D);
     LET (B) = (B) * (X);
  DOEND

ELSE

  DO
     LET (A) = (A) * (C);
     LET (B) = (B) * (Z);
  DOEND;
```

The example in Figure 2-1 has a compound statement. Other examples of compound statements can be found in the IF verb description in Chapter 8 and in the "Compiler Listings" section in Chapter 9.

In the example immediately above, note that the first DOEND does not have the semicolon (;) delimiter; the delimiter is used to end the entire compound statement. Individual statements between the DO ... DOEND pairs are terminated with a semicolon.

DOEND always requires a semicolon except under two circumstances: (1) just before ELSE, or (2) just before UNTIL in a REPEAT statement.

Statement Formatting

A Transact source program contains program text in 72-column records (not including line numbers). Program text can be entered in free format, but good programming practice suggests that you use a paragraph and an indented structure. In general, you can read and modify code more easily if you break the code into separate lines for labels, verbs, and options, and use indentation freely. You can break lines of code in any place except in the middle of a word. Thus, the following two statements would have the same effect:

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

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

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

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

If a string within quotes is split between lines, it must become two quoted strings.

```
DISPLAY "THIS IS A"
" TEST";
```

Comments

Comments document a program but do not affect program execution. Comments appear in the source code listing but do not generate any code. They can appear anywhere in the statement line and are enclosed between pairs of angled brackets (`<<` and `>>`), as follows:

`<<`*comment*`>>`

Figure 2-1 includes several comments. The following example shows how a comment is used in Transact code.

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

Delimiters

Transact programs can contain five explicit delimiters, plus a blank. The Transact delimiters and their functions are listed in Table 2-1. Figure 2-1 contains examples of how Transact delimiters are used.

Table 2-1. Transact Delimiters and Their Functions

Delimiter	Function
;	Semicolon - terminates a statement.
:	Colon - separates target/option phrases within statements, or serves as a terminator for a label, a command label, or a subcommand label. Also specifies a range in LIST= options.
,	Comma - separates options within a statement.
=	Equal sign - when used in an *option-list*, denotes the value an item should take, or serves as an assignment operator in MOVE and LET statements. Serves as a relational operator in condition clauses or specifies the label to which a program should branch.
( )	Parentheses - enclose a modifier or enclose an item name to reference its value. Enclose certain PROC statement parameters. Other uses are noted in verb specifications in Chapter 8.
`blank`	Blank space - required as a delimiter between a verb or verb (modifier) and its target but must never appear between a verb and its modifier. Can be used where blanks are significant place holders such as `DISPLAY "A   B   C"`, otherwise, blanks are ignored.

Reserved Words

The following list of words are used internally by Transact and may cause unexpected results if used as item names or labels. These words should be avoided when writing Transact programs.

AVERAGE
COUNT
MAXIMUM
MINIMUM
TOTAL

We do not recommend the use of Transact verbs or options as item names or labels.

3

Data Items

Data items are a significant part of the Transact Language. It is important to understand the different data types when you are developing applications in Transact. This chapter describes the following:

- Data item definitions
- Data item names
- Data item types
- Data item sizes
- Data type compatibility
- Parent items and child items
- Compound items
- Array subscripting
- Defining and handling arrays
- Alias items
- Using dictionary definitions with data item relations

Data Item Definitions

The previous chapter described the use of the DEFINE(ITEM) statement to define items for use in the program. The DEFINE(ITEM) statement must specify the name and type of the data.

A data dictionary can also be used to define items. Data items defined in a Dictionary/V data dictionary can be obtained at compile time by both the Transact/V and Transact/Ix compilers or at execution time by Transact/V. Those defined in a System Dictionary data dictionary can only be obtained at compile time for both Transact/V and Transact/iX.

Besides user-defined data item names, Transact also has a number of reserved system variables. These variables are available to the Transact programmer and are not defined in a DEFINE(ITEM) statement. The reserved system variables are described as follows:

LINE — the value of the terminal or printer line counter.

PAGE — the output page counter value.

PLINE — the value of the printer line counter.

STATUS — the current value of the status register.

TLINE — the value of the terminal line counter.

$CPU — the cumulative number of CPU seconds used by a Transact program.

$DATELINE — the current date and time.

$HOME — the name of the home or first database defined in the SYSTEM statement.

$PAGE — the current page number.

$TIME — the current time.

$TODAY — today's date.

Data Item Names

The first character of a data item name must be alphanumeric. Subsequent characters can be either alphabetic (A through Z), numeric (0 through 9), or any ASCII character other than the following characters: , ; : = < > ( ) ” or blank. The name can be from 1 to 16 characters long. For example, data items in Figure 2-1 include CUST-NAME, ORDER-DATE, ORDER-NUMBER, and AMOUNT-DUE.

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

```
LIST CUST-NO;
```

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

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

which manipulates the value of CUST-NO.

Data Item Types

Data items defined in a DEFINE(ITEM) statement or through Dictionary/V or the System Dictionary can be one of ten types. Table 3-1 lists the ten types of data items and their corresponding DEFINE(ITEM) code. You can specify that values must be positive by following the type with a plus sign(+). Positive-only values never require an extra character to display the sign.

Table 3-1. Transact Data Item Types

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

Data Item Sizes

The size of a data item is specified by indicating the number of characters or digits you want in each data item. Transact determines how much storage space is required for that number of characters or digits based on the data item type. You may override the default storage space either by specifying an exact storage size or the number of decimal digits—the number of digits you want to the right of the decimal point in a numeric data item. Be sure that space is allocated for the decimal point when you are computing data item sizes.

When data items are displayed, Transact generally requires the same number of display characters as the size specified in the DEFINE(ITEM) statement. Numeric data items must allow an additional character for the sign unless the item is positive only. When determining the space needed for displaying, add a space for the sign, even if the value to be displayed happens to be positive.

The following example identifies each of the values used in an item definition:

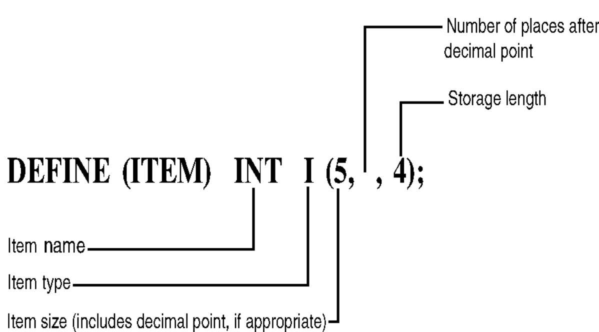


Table 3-2. Data Item Size

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
X or U	ASCII character string; 1 storage byte per specified character.	Same as storage.	DISPLAY PIC X or DISPLAY PIC A
9	ASCII numeric string, positive only; 1 storage byte per specified digit.	Same as storage.	DISPLAY PIC X
I*	Binary integer; default storage length depends on item size:	1 character per digit or decimal point, plus one character for a sign, unless item is positive only:	
	I(1) to I(4) = 2 bytes		COMP S9 to S9(4)
	I(5) to I(9) = 4 bytes	I+(5) = 5 chars I(5) = 6 chars I(5,2) = 6 chars	COMP S9(5) to S9(9)
	I(10) to I(18)=8 bytes		COMP S9(10) to S9(18)
	I(19) to I(27)=12 bytes		(none)

Table 3-2. Data Item Size (continued)

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
J*	COBOL binary type; (use for consistency with TurboIMAGE type J):		
	J(1) to J(4) = 2 bytes		COMP S9 to S9(4)
	J(5) to J(9) = 4 bytes	J(5) = 5 chars J(5) = 6 chars J(5,2) = 6 chars	COMP S9(5) to S9(9)
	J(10) to J(18) = 8 bytes		COMP S9(10) to S9(18)
	J(19) to J(27) = 12 bytes		(none)

*For I or J, any 16-bit integers can range between -32768 and 32767—defined as I(5,,2) or J(5,,2). Any 32-bit integers can range between -2147483648 and +2147483647—defined as I(10,,4) or J(10,,4). Definitions of I or J types with storage lengths other than 2, 4, 8, or 12 bytes get undefined results and can result in arithmetic traps.

Table 3-2. Data Item Size (continued)

Transact Type	Transact Default Storage Allocation	Transact Display Requirements	COBOL Type
Z*	Zoned decimal number; 1 storage byte per digit, including sign, if any, which is combined with the last digit:	1 character per digit or decimal point, plus 1 character for a sign, unless item is positive only:	
	Z+(10) = 10 bytes Z(10) = 10 bytes	Z+(10) = 10 chars Z(10) = 11 chars	DISPLAY PIC 9 DISPLAY PIC S9
P	Packed decimal digit; 1 nibble(1/2 byte) per digit, plus 1 nibble for the sign:	1 character per digit or decimal point, plus 1 character for a sign, unless item is positive only:	COMP-3 PIC 9
	P+(10) = 6 bytes P(10) = 6 bytes P(11) = 6 bytes	P+(10) = 10 chars P(10) = 11 chars P(10,2) = 11 chars	
	(Sign is stored even if item is positive only.		

*Zoned decimal data items are stored as a string of ASCII digits. For data items defined as Z, the right-most digit is always overpunched with a sign indicator, a character that represents both the sign and the right-most digit. The following table shows the characters representing the overpunch.

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

For data items defined as Z+ (implying positive only), no overpunch occurs and the right-most digit is unchanged. Z+ is stored like type 9.

The maximum size for 9, Z, P, I, J, and K data items is 27 characters unless the decimal point is included. In that case, the maximum size is 28 characters. The maximum size for R and E data items is 22 characters.

Table 3-2. Data Item Size (continued)

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

*For K, any 16-bit logical value can range between 0 and 65535 defined as K(5,,2). Any 32-bit logical value can range between 0 and 4,294,967,295 defined as K(10,,4).

Data Type Compatibility

It is important to know exactly how Transact allocates storage for data items used with a database or VPLUS form. Table 3-2 shows the storage allocated by Transact and the number of characters required for display, based on the type of data item and its size. It also gives the corresponding COBOL specification as an aid to understanding the Transact data types.

When Transact programs use VPLUS, databases, and data dictionaries, the data type compatibilities must be considered carefully. Data type compatibilities between Transact and these systems are discussed in the following sections.

Data Types and VPLUS

Data items are displayed on and entered from VPLUS forms in the external display format. Transact automatically converts between the display format and the Transact data type before a value is displayed and after a value is entered. (See Table 3-2 for the display and storage requirements.) It is important to remember that VPLUS does not add a character for the sign to its numeric data types, whereas Transact does. For example, if you want to display a 5-digit numeric data item in a VPLUS form defined with a maximum size of 5 characters, you must define it in Transact as positive only. A VPLUS data item with a size of 5 digits allows a maximum of 5 characters but a Transact data item defined as I(5) requires 6 display characters.

Table 3-3. Compatibility of VPLUS and Transact Data Types

VPLUS Field of Length n	Transact/V Compatible Types
CHAR	All types with display length n.
DIG	All positive only types of length n (no decimal points permitted).
NUM(m) and IMP(m), where m<n	9(n,m) I(n-1,m)* J(n-1,m)* P(n-1,m)* R(n-1,m)* Z(n-1,m)* E(n-1,m)* X(n-1)

* The VPLUS field must be one character larger than the Transact field to allow space for a plus or minus sign.

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 10.

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.

Compound Items

Compound items are data items that are divided into smaller entities that repeat more than once. They all have the same attributes (size, type, and number of decimal places). Compound items can be thought of as arrays. They, too, are defined in the DEFINE(ITEM) statement, if they are not already defined in the data dictionary. A specific occurrence of a compound item is referenced by an offset into the compound item, not by a data item name. Only the compound item name can be added to the list register or referenced in a LIST= option.

Array Subscripting

Occurrences of compound items and child items of compound items can be manipulated using subscripting. Array subscripting is allowed within the context of most verbs *except* when the Transact registers are being manipulated or updated. The only other case where subscripting is not allowed is with data items included in the LIST= option for any of the input/output verbs (such as DELETE, FILE, FIND, GET, PUT, REPLACE, and UPDATE). The error message, "`Subscript not permitted in this context,`" is displayed when such an attempt is made. The discussions of verbs in Chapter 8 specify exactly when array subscripting is allowed.

In contexts where subscripting is allowed, the following format is used:

```
(array-item[(subscript1[,subscript2 ...[,subscriptn]])])...
```

Parameters

array-item The array item must be either a compound item or a child item of a compound item. This parameter is required.

(subscript) A subscript can consist of a *data item*, *constant*, or an *arithmetic expression*. Subscripting is most efficient if the subscript is either a constant, a 16-bit integer item, or a simple expression (less than three operands) consisting of 16-bit integers, constants, or both. When a data item is used as a subscript, it must be enclosed in parentheses within the subscript parentheses. This parameter is optional. If no subscript is specified, the array item is treated as a single or compound data item.

The first subscript applies to the highest level of the compound item of which the array item is the child. Each succeeding subscript refers to the next level in the array. Once the first subscript value is specified, any omitted values will default automatically to 1. Since each level adds to the complexity of the program, we recommend that the number of subscripts be kept at a manageable level. The maximum number of allowable subscripts is 16.

The subscript value is evaluated at run time to determine which occurrence of the array item is to be referenced. A subscript value of 1 references the first occurrence of the array; subscript values less than 1 or greater than 32,767 are invalid.

Defining and Handling Arrays

Simple Arrays

The compound item is the simplest form of a Transact array. For example:

```
DEFINE(ITEM) INVOICE-NO 100 X(10);
```

Defines 100 occurrences of an X(10) type item called INVOICE-NO. Graphically, this array can be portrayed as follows:

```
-------------------------Occurrences of INVOICE-NO------------------------

   > <   54   > <   55   > <   56   > <   57   > <   58   > <   59   > <
----|----------|----------|----------|----------|----------|----------|---
789N|ABC123456D|STX432849D|URE849328D|NVM215425N|WAS950789N|YUR956789N|BB
A
----|----------|----------|----------|----------|----------|----------|---
```

The value in INVOICE-NO(56) is URE849328D, the value in INVOICE-NO(59) is YUR956789N, etc.

This array can be manipulated in the following manner:

```
MOVE(INVOICE-NO(4)) = (INVOICE-NO(80));
```

Assigns the eightieth occurrence of the array to the fourth occurrence of the array.

```
DISPLAY INVOICE-NO((INDEX));
```

Displays the occurrence of the array equal to the current value of the data item INDEX.

```
DISPLAY INVOICE-NO;
```

Displays the entire compound item. Transact treats references to unsubscripted arrays as ordinary compound items. There is one exception to this rule. When the IF statement is used with non-subscripted compound items, only the first element of the compound item is used for the comparison.

The following example shows how the IF statement only compares the first element in an array:

```
SYSTEM T320;

DEFINE(ITEM) INVOICE-NO-A 100 X(10):
             INVOICE-NO-B 100 X(10);

LIST         INVOICE-NO-A:
             INVOICE-NO-B;
MOVE (INVOICE-NO-A(1)) = "G200500001";
MOVE (INVOICE-NO-A(2)) = "9999999999";
MOVE (INVOICE-NO-B(1)) = "G200500001";
DISPLAY INVOICE-NO-A;
DISPLAY INVOICE-NO-B;

 IF (INVOICE-NO-A) = (INVOICE-NO-B) THEN
   DISPLAY "THE FIRST ELEMENTS ARE EQUAL"
ELSE
   DISPLAY "THE FIRST ELEMENTS ARE NOT EQUAL";

MOVE (INVOICE-NO-B) = (INVOICE-NO-A);

DISPLAY INVOICE-NO-B;

EXIT;
```

The following output is generated by the above example program:

```
INVOICE-NO-A:
 G200500001 9999999999




INVOICE-NO-B:
 G200500001




THE FIRST ELEMENTS ARE EQUAL

INVOICE-NO-B
 G200500001 9999999999
```

The example above shows the difference between the IF and MOVE statements with non-subscripted compound items. The IF statement results in the invoice data items being equal since Transact only compares the value of the first element of the array when this statement is used.

The last MOVE statement results in all elements of `INVOICE-NO-A` being copied to the corresponding elements of `INVOICE-NO-B`.

Transact does not allow you to manipulate arrays with an occurrence count of 1. The following example causes a run-time error message, "`Cannot subscript a non-array item,`" to appear.

```
Move (one(1)) = "A";
```

An array with an occurrence count of 1 is considered a non-array item by Transact and therefore cannot be used.

Complex Arrays

Complex arrays have one or more child items associated with each occurrence of the compound parent item.

The child items may be defined as simple items or compound items. The child items can be parent items with child items defined as well. Each child item definition that is a compound item adds a level to the array definition. Each level is a redefinition of its parent, thus providing more detailed definition of the array.

Child Identified as Simple Items

The following is an example of a complex array definition whose array child items are defined as simple items.

```
DEFINE(ITEM) INVOICE-NO 100  X(10):
                INVOICE-PFX  X(3) = INVOICE-NO(1):
                INVOICE-SFX  9(6) = INVOICE-NO(4);
```

Defines 100 occurrences of an X(10) type item called INVOICE-NO. In addition, it defines two child items, INVOICE-PFX and INVOICE-SFX, for each element of the INVOICE-NO array.

Each of the child items can be subscripted even though they are not themselves compound items. When subscripted, the value of the subscript applies to the compound parent item of the child item referenced. This structure is very similar to a Pascal array of records structure.

Graphically, this structure can be portrayed in the following manner:

```
-----------------------Occurrences of INVOICE-NO -------------------------
   > <   54   > <   55   > <   56   > <   57   > <   58   > <   59   > <
----|----------|----------|---------|----------|----------|----------|---
789N|ABC123456D|STX432849D|URE849328|NVM215425N|WAS950789N|YUR956789N|BBA
----|----------|----------|---------|----------|----------|----------|---
```

The value in INVOICE-NO(55) is STX432849D and the values in INVOICE-PFX(55) and INVOICE-SFX(55) are STX and 432849, respectively.

For subscripting to work correctly, the total storage length of all child items defined in such a structure must be less than or equal to the storage length for each occurrence of the parent. Child elements must not span across occurrences of the parent. The following structures, for example, would be incorrect:

```
DEFINE(ITEM) INVOICE-NO 1000  X(1):
                INVOICE-PFX   X(3) = INVOICE-NO(1):
                INVOICE-SFX   9(7) = INVOICE-NO(4);
```

This structure defines 1000 occurrences of an X(1) type item called INVOICE-NO. The child items, totaling 10 bytes in storage length, cannot fit within a parent that is only 1 byte long.

```
DEFINE(ITEM) INVOICE-NO 100  X(10):
                INVOICE-PFX  X(4) = INVOICE-NO(1):
                INVOICE-SFX  9(7) = INVOICE-NO(5);
```

This structure is incorrect because the child items total 11 bytes and thus cannot be defined within a parent item that is 10 bytes long. However, this structure

```
DEFINE(ITEM) INVOICE-NO 100  X(10):
                INVOICE-PFX  X(4) = INVOICE-NO(1):
                INVOICE-SFX  9(7) = INVOICE-NO(4);
```

is correct because, although the total storage lengths of child items is greater than the storage length of the parent, the last byte of INVOICE-PFX and the first byte of INVOICE-SFX overlap, thereby requiring only 10 bytes of total storage length.

Child Items Defined as Compound Items

The following example illustrates an array definition whose child items are defined as compound items. For simplicity, this example assumes that all months are 28 days long. The item YEAR will hold one character for each day of the year. YEAR is not defined as an array. However, the definition of its child items MONTH, WEEK, and DAY redefine it into arrays for easier data manipulation. Each of the child items are compound items; MONTH and WEEK are also parent items. This definition contains three levels.

```
DEFINE(ITEM) YEAR X(336):
                 MONTH 12 X(28) = YEAR(1):
                      WEEK 4 X(7) = MONTH(1):
                          DAY 7 X(1) = WEEK(1);
```

The following example displays a specific day of the year:

```
DISPLAY DAY(2,1,5);
```

This verb statement displays the fifth day of the first week in the second month. Since YEAR is not a compound item, the first subscript (2) refers to the highest level compound item, MONTH. The next subscript (1) refers to the WEEK which is the next highest level compound item. Finally, the last subscript (5), refers to the fifth occurrence of the compound item, DAY.

The next example shows what happens when succeeding subscripts are omitted:

```
DISPLAY DAY(2);
```

Since the omitted subscripts default to 1, this is equivalent to (2,1,1) that displays the first day of the first week in the second month.

When the number of subscripts exceeds the number of dimensions specified by the DEFINE statement, an error occurs. For example:

```
DISPLAY DAY(2,1,5,3);
```

Obviously, there are not enough parent levels to justify four subscripts. This example will result in an error and the message, "Too many subscripts for item," is displayed.

If the subscripts are larger than the specified range an error occurs. For example:

```
DISPLAY DAY(999);
```

This example results in an error and the message, "Array subscript is out of range", because 999 is beyond the 12 element definition.

Under some conditions, you may want to reference globally all of the elements in the array. For example:

```
MOVE (YEAR) = " ";
```

Since there are no subscripts specified, this will set all 336 elements to blanks by moving a space to the first element of the year and filling the remaining 335 elements with blanks. This differs from the following example:

```
MOVE (DAY) = "1";
```

which sets the first character of DAY to a 1 as requested, then pads the remaining six characters with blanks. In the next example, the subscript is specified:

```
MOVE (DAY(1)) = "X";
```

which results in the element DAY (1,1,1) being set to an X. Since the length of the source and destination are the same, no filling is done.

Note

Since Transact originally supported only single-level array subscripting, the LET OFFSET construct was used to reference arrays with multiple levels. This is no longer necessary. We strongly recommend that you address array items by using subscripts. However, there may still be programs that manipulate arrays in this manner. If this is the case, you should be careful not to combine the use of the LET OFFSET verb with a subscripted item. Doing so may cause the program to update the data register in areas outside the limits of the item referenced and could lead to unpredictable results. Since this was previously the only way to simulate arrays with multiple levels, no error message is generated.

Special Considerations

If any level definition in an array structure has an occurrence of 1, it is ignored by Transact as part of the array definition. For example,

```
DEFINE(ITEM) INVOICE-DATA X(1000):
                INVOICE-NO 100 X(10) = INVOICE-DATA(1);
```

is a single level array of 100 invoice numbers. Any subscript applied to INVOICE-NO is treated as a subscript to the INVOICE-NO array. Likewise,

```
DEFINE(ITEM) INVOICE-DATA 100 X(10):
                INVOICE-NO X(10) = INVOICE-DATA(1);
```

is also a single-level array of 100 invoice numbers.

By the same token,

```
DEFINE(ITEM) INVOICE-DATA  X(1000):
                   INVOICE-NO 100  X(10) = INVOICE-DATA(1) :
                     INVOICE-PFX  X(3)  = INVOICE-NO(1) :
                     INVOICE-SFX  9(6)  = INVOICE-NO(4) ;
```

is also a single-level array because INVOICE-DATA is not a compound item. The child items of INVOICE-NO are simple items and do not add a level to the array structure.

Alias Items

Any data item can be assigned an *alias-name*—the alias is another name for the defined data item. You would use an alias in a Transact program where the data dictionary definition of a data item has the same definition but a different name in a data set. The primary definition in a data dictionary can be associated with one or more alias names to identify data items in data sets that have different names. The primary name is always used in the Transact program. You must define all alias relations with a DEFINE(ITEM) statement in your program; Transact ignores alias definitions in a data dictionary.

Using Dictionary Definitions with Data Item Relations

You must use caution when using data dictionary definitions of parent/child relations, compound/sub-item relations, and aliases. You must specifically define an alias relation in the DEFINE(ITEM) statement of your Transact program; any alias relations defined in a data dictionary are ignored by Transact. The Transact processor recognizes parent/child and compound/sub-item relations defined in a data dictionary, but you can only reserve space in the list register for the parent or compound item. (For details, see the DEFINE(ITEM) discussion in Chapter 8, and the sections "Parent Items and Child Items," "Compound Items," and "Alias Items" in this chapter.)

4

Transact Registers

The Transact language differs from many conventional languages (such as COBOL and Pascal) in its use of dynamic, run-time "registers" to allocate and store data, control run-time processing, and communicate run-time status. Because these registers are active when the program is executing, you should always be aware of their values. Skillful manipulation of these registers can greatly improve both program and programmer efficiency.

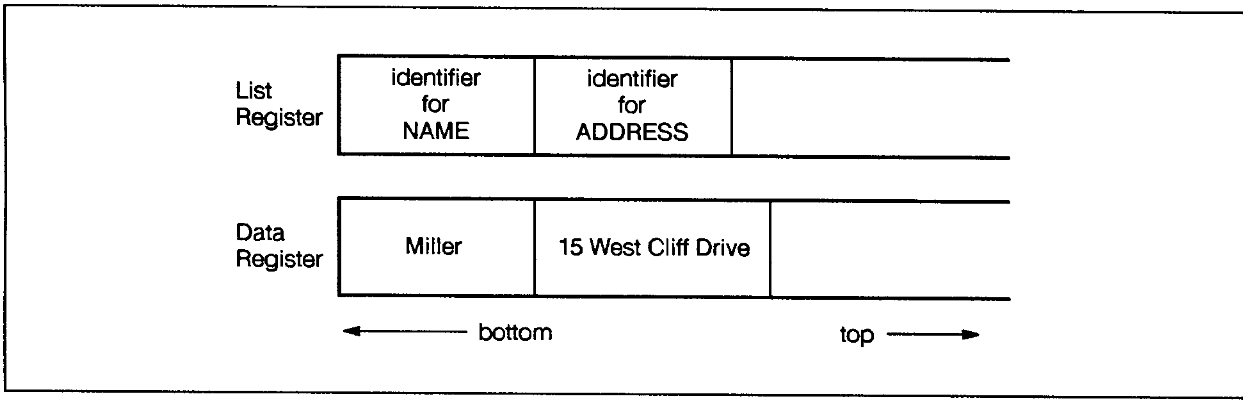
This chapter describes the Transact registers and how they work, including:

- List and data registers
- Key and argument registers
- Match register
- Update register
- Input register
- Status register

List and Data Registers

The list and data registers are used to allocate data storage space for data items manipulated by the program. The data register has a default size of 1024 words, and it is the actual storage area for the values of data items. By itself, however, this register does not differentiate between the end of one item and the beginning of another. In order for the Transact program to do this, the values in the data register must be mapped by the identifiers in the list register. Before a program can manipulate a data item, it must be "listed," or placed in the list register by including a LIST statement in the program. For every data item that you name in the LIST statement, Transact includes an identifier in the list register. Transact uses this identifier to locate the correct value in the data register.

The order in which the data item identifiers are listed in the list register determines the order of the corresponding data item values in the data register. Data item identifiers in the list register and the corresponding space in the data register, are allocated in the order that the data item names are specified in the LIST statement. For example, if the first data item identifier added to the list register is for a data item called NAME, a 6-byte character string, the first 6 bytes of the data register are allocated to hold the value of NAME. If the second item identifier in the list register is for a data item called ADDRESS that identifies a 20-byte character string, the next 20 bytes of storage space in the data register are reserved for the value of ADDRESS.



Data item identifiers and item values are added to the list register and data register starting at the bottom and extending toward the top. The same data item can be added to the list register and data register more than once in a single program. When this is done, only the most recent addition of the data item is accessible by the program.

In many ways these registers behave like stacks, and it is helpful to think of them as such.

Managing the List and Data Registers

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

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

To minimize your data storage requirements, you should release the data register space occupied by your data items when you are finished using them. Transact can do this for you. If you are using a command structure, Transact resets the list register whenever a command sequence executes. If you are not using a command structure, you should manage your data storage directly with Transact statements.

When data items are removed from the list register, they are removed from top to bottom; that is, the last data item added is the first data item removed. The values, however, corresponding to the data items removed from the list register, still exist in the data register. You can access these values again by listing the data item again in the list register. It is possible also to relist different data items in the list register and redefine the values in the data register.

When a data item is listed multiple times, the last occurrence of the data item in the data register is the one that is used. To access a previous occurrence of the data item, you must remove the current occurrence from the list register via the SET(STACK) verb.

Key and Argument Registers

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

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

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

Key Register

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

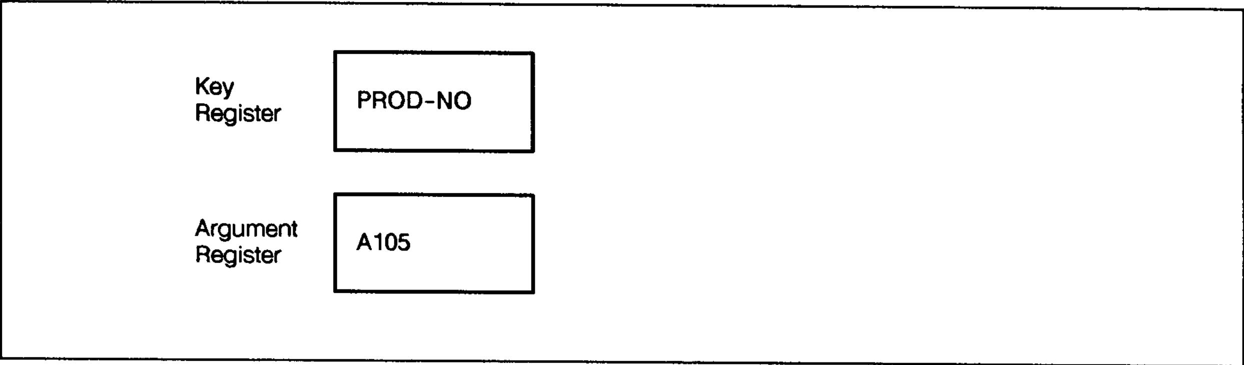
The key register is needed only when the key name must be specified. It is needed to locate a particular key in a KSAM file and to locate the chain head in a detail data set. The key register is not needed to access key items in manual or automatic master sets. There is only one key (search) item in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

Argument Register

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

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

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



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

Match Register

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

To use the match criteria, the match items must be in the list register, and they also must be retrieved by the data management statement that uses the match criteria. Therefore, you must add match items to the match register, add each data item to the list register, *and* include each data item in a LIST= option of the data management statement. If a match item is not specified in the LIST= option, or the LIST= option is omitted and the match item is not in the list register, the data management statement ignores the match criteria associated with that data item.

You can specify as many match criteria as you want. Also, you can assign different criteria to the same data item or to different data items, or specify the same criteria for different data items. By default, a Boolean AND connects selection criteria gathered from different PROMPT(MATCH) or DATA(MATCH) statements. A Boolean AND also connects selection criteria from multiple SET(MATCH) statements unless the statements use the same data item name and specify equality as the connector; such statements are joined by a Boolean OR.

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

Match Register

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

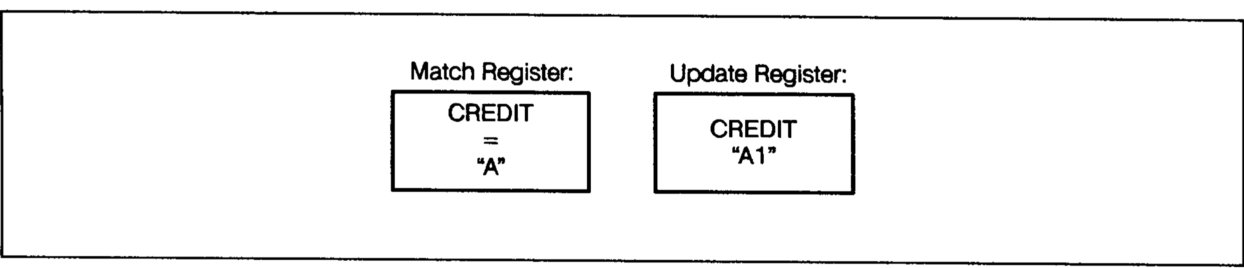
Entries in the match register can be cleared or selectively deleted by using the RESET verb with the MATCH option. Users can also override the specific defaults with their responses to a PROMPT(MATCH) or a DATA(MATCH) prompt. See "Responding to a MATCH Prompt" in Chapter 5.

Update Register

The update register holds *pairs* of update specifications. Each pair consists of a data item name and a new value for that data item. These name/value pairs can be used to update records in an MPE or KSAM file or in a data set. The update register is only used with the REPLACE verb to update one or more records. Entries in the update register can be cleared or deleted selectively by using the RESET verb with the UPDATE option.

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

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



Note

You do not use the update register with the UPDATE verb nor would you normally use it with REPLACE to update multiple entries with different values. The update register is particularly useful for making the same change to multiple entries.

Input Register

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

Status Register

The status register is a double word register that holds status information about the last operation performed. The contents of the status register differ depending on whether Transact uses automatic error handling, or you control error handling programmatically. See "Automatic Error Handling" and "Using the STATUS Option" in Chapter 7 for explanations of the status register contents.

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

How Registers Work

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

Verbs and Registers

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

- LIST causes a data item name to be placed in the list register. Appropriate space is allocated in the data register.
- DATA places values into space already allocated in the data register. These values come from user input, because DATA causes a prompt.
- PROMPT places the data item name in the list register. The value (supplied by the user) is placed in the data register.
- INPUT places a character string (supplied by the user) into the input register.

How LIST, DATA, and PROMPT behave is specified by their modifier. In addition to the verbs listed above, all the database- and file-access verbs (except PUT) and the assignment verbs LET, SET, and MOVE, update the data register. The data access verbs get the data from files or databases. With LET, SET, and MOVE, the data is assigned in the program.

For example, PROMPT affects only the list and data registers, whereas PROMPT(PATH) affects the list, data, key, and argument registers. If you only want to add data items to the list register, use LIST with no modifier; if you only want to add a data item to the key register, use LIST(KEY). Table 4-1 shows how verbs and modifiers work together to affect registers.

Table 4-1. Registers Affected by Modifiers on Specific Verbs

	Register Affected by the Verb:			
Modifier	Prompt	List	Data	Input
none	List Data	List	Data	Input (1)
PATH	List Data Key Argument	List Key	Data Argument Key (2)	-
KEY	Key Argument	Key	Argument Key (2)	-
MATCH	List Data Match	List Match	Data Match	-
UPDATE	List Data Update	List Update	Data Update	-
SET	List (1) Data (1)	-	Data (1)	-
ITEM	-	-	Data (3)	-

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

Sample of Transact Coding

The following code extracted from a Transact program explains how registers work. It shows:

- Data set operations and
- Register activity.

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

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

PROMPT(PATH) CUST-NO;

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

PROMPT(PATH) PART-NO;
PROMPT QTY-ORDERED;

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

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

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

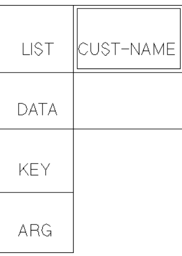
The database that is referenced contains the three data sets shown below (PARTS, CUSTOMERS, and ORDERS). The data items in each data set are also listed.

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

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

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

The following figures show how specific statements affect specific registers.

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

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



LIST	CUST-NAME	CUST-ADDRESS
DATA		
KEY		
ARG		

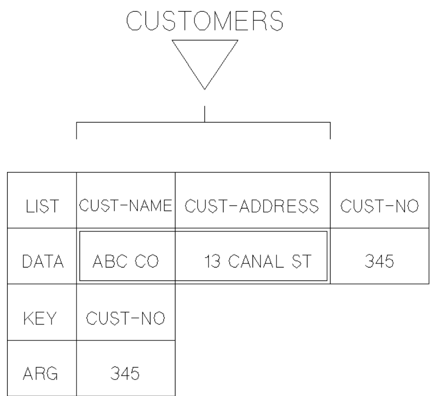
```
PROMPT(PATH) CUST-NO;
```

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

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

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

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



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

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

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

```
PROMPT QTY-ORDERED;
```

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

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

```
LIST COST;
```

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

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

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

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

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

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

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

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

PARTS

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

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

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

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

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

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

PARTS

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

Computes the cost and places it in the data register. Updates the ORDERS data set with the values from CUST-NO through COST.

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

ORDERS

5

User Interface

Transact supports three modes of user interface which may be used singularly or jointly in any Transact program. The three modes are:

- Command sequences
- Character mode using DATA, INPUT, and PROMPT verbs
- Block mode data entry using VPLUS

This chapter discusses each of these modes as well as the use of special characters and keys that affect the execution of the program.

Command Sequences

You can structure the body of a Transact program around command sequences specifically designed for a particular interactive interface to the program. A command sequence consists of the statements placed between a command or a subcommand and the next command, subcommand, or END statement, whichever comes first. One command sequence in Figure 5-1 begins with the statement following the subcommand $C and ends with the statement preceding $PAYMENT. The statements after $P and before $$UPDATE are also considered a command sequence.

Command sequences divide the Transact program into functional parts that make logical sense to you and that are meaningful to the user.

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

```
SYSTEM CINFO,
       BASE = CUST(,3,1);

  $$ADD:
  $$A:
     $CUSTOMER:       <<Add a new customer to the database>>
     $C:

        PROMPT CUST-NO("Enter customer number "):
         CUST-NAME("Enter customer name "):
         CUST-ADDR("Enter customer address ");
        PUT CUST-MAST;

     $PAYMENT:        <<Add payment to A/R data set>>
     $P:

        PROMPT CUST-NO("Enter customer number "),
                   CHECK = CUST-MAST:
               PDATE("Enter payment date "):
               INV-NO("Enter invoice number "):
               AMOUNT("Enter amount of payment ");
        PUT AR-DETAIL;

  $$UPDATE:
  $$U:
     $ADDRESS:        <<Change customer's address>>
     $A:
        PROMPT CUST-ADDR("Enter customer address ");
        UPDATE CUST-MAST, LIST=(CUST-ADDR);

  $$DELETE:
  $$D:
     $CUSTOMER:       <<Delete old customer from database>>
     $C:
        PROMPT(KEY) CUST-NO("Enter customer number ");
        DELETE CUST-MAST;

END CINFO;
```

Figure 5-1. Program Using Command Sequences

Processing Command Sequences

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

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

When the sample program in Figure 5-1 is executed, the user might enter any of the following in response to the prompting character:

```
> ADD CUSTOMER

> ADD PAYMENT

> UPDATE ADDRESS

> DELETE CUSTOMER
```

Command and Subcommand Labels

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

$$*command:*
    $*subcommand:*

Either label can contain from 1 to 16 characters in addition to the leading $$ or $. The label must begin with an alphanumeric character. The remaining characters can be any characters except the delimiters ($, ; : = < > ( ) [ ] " or a blank). All command and subcommand labels are global to the program and can be referenced from any program segment. (See "Program Segmentation" in Chapter 9.)

You can provide the user with short forms for commands and subcommands. These short forms are illustrated for each command and subcommand in Figure 5-1.

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

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

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

```
> CHANGE ADDRESS
```

or merely

```
> CHANGE
```

User-Entered Passwords for Commands and Subcommands

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

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

Consider the following code:

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

When the user enters the command

```
> ADD CUSTOMER
```

Transact requests the password:

```
COMMAND PASSWORD>
```

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

```
COMMAND PASSWORD> PQX2
```

Note that the password is in uppercase not `pqx2` (lowercase). Passwords must be exact.

Subcommands as well as commands can require passwords:

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

When the user enters the command:

```
> ADD CUSTOMER
```

Transact requests both passwords:

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

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

```
INVALID COMMAND PASSWORD
```

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

```
INVALID SUB-COMMAND PASSWORD
```

In either case, Transact issues a prompt for another command.

Built-in Commands

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

COMMAND [*command-name*]	Lists all the commands, or lists all the subcommands associated with the specified command name, in the currently loaded program. An "*" will be displayed for null subcommands.
EXIT	Generates an exit from Transact.
INITIALIZE	(Transact/V only) Generates an exit from the current program and initiates the loading of a new program. When you enter INITIALIZE, you are prompted with SYSTEM NAME>.
RESUME	Causes the resumption of a process that was interrupted by a Ctrl Y. (Ctrl Y is explained later in this chapter under "Special Characters that Control Program Execution".
TEST[,*mode* [,*range*]]	(Transact/V only) Causes Transact to execute in test mode for the specified range; if no mode is specified, test mode is turned off.

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

Command Qualifiers

Transact program commands, such as ADD CUSTOMER or UPDATE ADDRESS, can be qualified by using command qualifiers. The qualifiers that are recognized by Transact include:

FIELD	Indicates the prompted-for field length on HP 264X series terminals.
PRINT	Directs output to the line printer instead of to the user's terminal.
REPEAT	Repeats a command sequence until a termination character ("]") is entered for a prompt other than the first field of a command sequence. Entering a "]" for the first field prompt or "]]" for any field prompt returns control to the command mode regardless of the level of command or subcommand. However, if there is an active END sequence block specified by the "SET(OPTION) END=" statement, it is executed before passing control to the first statement of a command sequence or returning to command mode. The list register is reset before repeating a command sequence.
SORT	Sorts any data generated by an output verb within the command sequence. Entering a "]" from other than the first prompt returns the user to the first prompt of the current command or subcommand.

TPRINT Directs a line-printer-formatted display to the user's terminal.

The command string "DISPLAY COMPANY" causes a number of companies to be listed on the terminal. The user can enter the command:

```
PRINT SORT DISPLAY COMPANY
```

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

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

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

This command produces on the line printer a sorted list of all company names beginning with the letters "DE". (See "MATCH Specification Characters" later in this chapter, for an explanation of the character "^".)

The REPEAT option could be added to this command string:

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

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

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

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

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

DATA, INPUT, and PROMPT

Transact can perform a number of different data entry functions in character mode with the use of the DATA, INPUT, and PROMPT verbs.

The DATA and PROMPT verbs will prompt the user for a response to a prompt string or item name. Transact will validate the data and prompt the user again if any errors are detected. The major difference between the DATA and PROMPT verbs is the registers that are updated. The PROMPT verb will add the item to the list register, whereas the DATA verb will only update an item already on the list register. Depending on the modifier used for each verb, values would also be placed in the data, argument, match, and/or update registers. The input value for the DATA and PROMPT verbs is usually used for subsequent file and data set operations.

The INPUT verb will prompt the user for a response to a prompt string. The value will be placed in the INPUT register. The INPUT verb is used to test a user response.

Special characters that can be entered by the user in response to the prompt are discussed later in this chapter. To get additional information and examples of the DATA, INPUT, and PROMPT verbs, see Chapter 8. A discussion of the use of the MATCH modifier with the DATA and PROMPT verbs follows.

Responding to a MATCH Prompt

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

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

With native language support, the usage of the specified language is reflected in the relational operators, the connectors, and the range indicator (such as, "TO"). See "Native Language Support" in Appendix E for more information.

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

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

Where:

relation — Is a relational operator for a condition that is other than equal; use one of the following:

NE - not equal to (< >)
LT - less than (<)
LE - less than or equal to (<=)
GT - greater than (>)
GE - greater than or equal to (>=)

value — A numeric value or a partial string specification. A string with embedded blanks must be enclosed in quotation marks.

TO — Specifies a range of values bounded by the value preceding and the value following TO.

connector A logical connector that is one of the following:

AND — Specifies that the record accessed must contain both the value before and the value after this operator.

OR — Specifies that the record accessed must contain either the value before or the value after this operator.

The precedence of these connectors is AND then OR.

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

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

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

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

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

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

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

> 20 TO 30

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

> GE 20 AND LE 30

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

> LAX OR CGY

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

> REG^ OR SAS^ AND CITY^^

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

> "San Diego, California"

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

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

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

VPLUS Interface

Transact uses a subset of the data management verbs (GET, PUT, SET, UPDATE) to access and control VPLUS forms. Without making calls directly to VPLUS intrinsics, you can retrieve data from forms, move data to forms, control the sequence of forms, manage function keys, and send messages to the forms window. The VPLUS interface supports function key labels on HP terminals and PCs using terminal emulators when such labels are defined in FORMSPEC. It does not support the split screen feature of HP 262X terminals, nor does it support data capture terminals.

Often, many different functions are performed with a single Transact statement. For instance, GET(FORM) does the following:

- Gets and displays a form.
- Reads data entered by the user.
- Performs any edits specified through FORMSPEC.
- Highlights any field that contains errors and sends an error message to the window.
- Transfers the data to the program.
- Checks the data against the data definitions in the program, and again performs error processing, if necessary.
- Performs any finish phase operations specified by FORMSPEC.

Normally, Transact programs operate with the terminal in character mode. Using any VPLUS interface verb places the terminal in block mode. Transact automatically switches back to character mode for any operation, such as a DISPLAY statement, that requires character mode.

VPLUS forms files and form layouts can be defined in the data dictionary or in the SYSTEM statement of a Transact program. When definitions are taken from the data dictionary, a reference to a forms file in the SYSTEM statement causes Transact to retrieve all the forms defined in the forms file for use at run time. You can also specify the form definitions completely in the SYSTEM statement. This gives you the flexibility of redefining field types and field order to suit your own purposes. Also, you can specify within a forms file only the forms that are needed.

Local Form Storage

Transact supports VPLUS local form storage with all forms caching terminals. Local form storage reduces datacomm overhead with frequently used forms and causes the form to be displayed all at once instead of being painted on the screen line by line. This is accomplished by placing the form into the terminal memory. Only form images are loaded—not associated data.

The HP 2626A and HP 2626W terminals can store as many as four forms locally. The HP 2394A and HP 2624B terminals can store a maximum of 255 form names in their forms directory. However, the number of actual form images they can store might be lower than this, depending on the size of the forms and the size of the terminal memory. When the terminal is not one of these types, the p-code generated for local form storage is ignored.

The local form storage feature is enabled by the FSTORESIZE parameter of the SYSTEM verb. This parameter specifies the maximum number of forms to be loaded into the terminal memory. The SET(OPTION) FORMSTORE statement causes the specified forms to be loaded into the terminal memory. The number of forms successfully loaded is returned to the STATUS register. Transact suppresses any error messages resulting from overloading the terminal's local form storage memory at run time. Refer to the *VPLUS Reference Manual* for more information.

Note

Transact VPLUS forms caching is implemented to operate within a single Transact system level. Forms caching should not be used within both the main system and a subsystem or between different levels in a Transact program.

Look-Ahead Loading

After local form storage is enabled, look-ahead loading is performed by default. With this option, VPLUS loads the next form (as defined by the forms file) before or after reading data from the current form, depending on the type of datacomm being used. If the terminal memory is full, VPLUS unloads the least recently used form (or forms) to make room. To disable look-ahead loading, you use the SET(OPTION) NOLOOKAHEAD statement. The RESET(OPTION) NOLOOKAHEAD statement can be used to enable look-ahead loading, but it is needed only if the SET(OPTION) NOLOOKAHEAD statement was previously used in the program.

You can still control which forms are loaded during look-ahead loading, but the forms you specify can be unloaded to make room for the next form. If you are unfamiliar with VPLUS local form storage, you should not disable look-ahead loading, since Transact and VPLUS do most of the work for you. If you include the FSTORESIZE parameter in the SYSTEM statement but do not use the SET(OPTION) FORMSTORE statement, local form storage is still performed using look-ahead loading, even though forms are not explicitly loaded in the program.

If look-ahead loading is disabled, VPLUS does not unload forms from local storage to make room for new ones. To unload forms, you must use the RESET(OPTION) FORMSTORE statement. This statement is used 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 look-ahead loading to choose the best form to unload. The RESET(OPTION) FORMSTORE statement is not required in any other situation, since the local form storage memory is purged at the end of the process.

Automatic Form Loading

Automatic form loading causes VPLUS to load each new form into local storage before displaying it, if the form is not already in local storage. When the local form storage memory is full, VPLUS unloads the least recently used form. You can enable automatic form loading by using the SET(OPTION) AUTOLOAD statement.

The two types of form loading differ in that automatic form loading loads the *current* form, while look-ahead loading loads the *next* form (as defined in FORMSPEC). When the form sequence in FORMSPEC is not the one used in the application, look-ahead loading is usually inappropriate. (For more information, see the discussions of the SYSTEM and SET verbs in Chapter 8.)

Local Form Storage Example

The example shown below causes VPLUS to use automatic form loading with a maximum of four forms. Local form storage is initialized to contain four forms. Any new form not already in local form storage will be added after the least recently used form is deleted.

```
SYSTEM ORDDL, BASE=ORDMGT, VPLS=ORDMGTFF, FSTORESIZE = 4;

 DEFINE(ITEM) FORMSLOADED I(4);
     |
     |

SET(OPTION) FORMSTORE =
            (ORDMGTMENU,ADDCUST,CHGCUST,DELCUST); << Load forms >>

IF (FORMSLOADED) < 4 THEN
   DO
   << Report form loading error >>
   DOEND;
     |
     |

SET(OPTION) NOLOOKAHEAD;                 << Look-ahead off >>
SET(OPTION) AUTOLOAD;                    << Auto load enabled >>
```

Special Notes

If you enable local form storage, VPLUS automatically configures the HP 2626A and HP 2626W terminals to use datacomm port 1 and removes the HPWORD configuration from the HP 2626W. If local form storage is not enabled (that is, if the FSTORESIZE parameter is omitted from the SYSTEM statement), then VPLUS does not disturb the configurations of the terminals. The local form storage memory is wiped out if the user presses RESET or REFRESH at run time, if the DISPLAY verb is used between forms, if the terminal goes out of block mode, or if appending forms are used on the HP 2626A and HP 2626W terminals.

Special Characters and Keys That Control Execution

Several categories of special characters and keys lend programmers and users powerful control over Transact's program execution. These characters and keys include:

- Ctrl Y
- Data entry control characters
- Match specification characters
- Field delimiters
- Special keys for use with VPLUS forms

Control Y

Transact recognizes Ctrl Y entered from the user terminal as an operation break that returns control to Transact.

You can use the Ctrl Y feature to halt program execution temporarily in order to enter a TEST or COMMAND command. After using either of these commands, you can continue execution by entering the command RESUME. This feature is especially useful during Transact/V program debugging. For example, you can enter the command TEST followed by a test-mode parameter when the program is temporarily halted. When you resume execution, the program executes in the specified test mode. (See Chapter 10 for a description of the test facility.)

Data Entry Control Characters

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

]	Terminates the current operation. Control passes to the next higher processing level, which can be the command level.
]]	Terminates the current operation. Control passes to command level.
!	Generates null responses for all subsequent prompts when entered as a response to a data item prompt. It generates null responses for all subsequent sub-item prompts within a compound item when entered as a response to a compound item prompt.

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

MATCH Specification Characters

Several special characters help to set up match specifications and are used in response to prompts issued by PROMPT(MATCH) and DATA(MATCH) statements. Because of their special meaning, these characters should only be used for these purposes in character strings. They include the following:

Single Caret(^) Indicates a partial-word selection criterion for alphanumeric string data items.

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

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

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

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

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

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

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

Field Delimiters

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

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

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

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

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

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

Special Keys for Use with VPLUS Forms

Certain special keys can be used while processing VPLUS forms sequences:

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

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

f1 to **f7**
: Control passes to the next statement in the sequence.

f8
: Control returns to command level unless there are no commands to execute, in which case the EXIT/RESTART> prompt is issued.

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

6

Accessing Databases and Files

Transact's data management facilities allow you to use databases, KSAM files, and MPE files without making intrinsic calls. The data management interface is built into a common set of verbs that use a common set of special purpose registers. This chapter covers:

- Using databases with Transact.
- Using KSAM and MPE files with Transact.

When using databases, KSAM files, and MPE files through Transact, the verbs and modifiers specify particular functions. For example, FIND(CHAIN) retrieves all entries that have a particular key value from a database or KSAM file. The key value is specified in the KEY and ARGUMENT registers. Similarly, you can use FIND(SERIAL) to sequentially scan a database, KSAM file, or MPE file for all entries that meet the selection criteria set up in the MATCH register.

The flexibility provided by each verb and modifier is enhanced by the special registers. Please see Chapter 4 for a description of the register functions. The following are examples of when these special registers are used:

- The key register contains the key for keyed selection.
- The match register contains criteria for selecting particular records or entries.
- The update register specifies the data item to update and its new value.
- The status register contains values used in error handling. With automatic error handling, the status register is set to the number of selected records for the file or database being accessed. When automatic error handling is suppressed, the status register is set to the subsystem error number if an error occurs.

Although you can take all data definitions from a data dictionary, in the SYSTEM statement of the program you must name each file or database used by your program. You need not name the individual data sets. For a database, you can also specify a password, a locking scheme, and an open mode in the SYSTEM statement. For a file, you can specify the access options (*aoptions*) and file options (*foptions*) to be used when opening the file.

Using Databases

Transact opens the database at the start of your program. The SYSTEM statement's BASE option allows you to define the name of the database, the password, the type of access mode to use in opening the database, and the locking scheme to use. For example:

```
SYSTEM MYSYS,BASE=DBASE("READER",1,1);
```

In this example, the database name is `DBASE`, the password is `READER`, the access mode is one and the lock option (*optlock*) is one.

Access Mode

The access mode you choose on the SYSTEM statement determines the type of operation that you can perform on the database as well as the types of operations other users can perform concurrently. To simplify the definition of the various access modes, the following terminology is used:

- *Read* access allows the user to locate and read data entries. The FIND and GET statements are used with read access.
- *Update* access allows the user to replace values in all data items except search and sort data items. Update access also provides read access. UPDATE, FIND, and GET statements are used with update access.
- *Modify* access allows the user to add and delete entries. Modify access also provides update and read access. REPLACE, PUT, DELETE, FIND, GET, and UPDATE statements are used with modify access.

For additional information on access modes, see the *TurboIMAGE/V* or *TurboIMAGE/XL Database Management Reference Manual*.

Database Close

A database can be closed in Transact by:

- Using the CLOSE verb
- Calling DBCLOSE via the PROC statement
- Ending the program

You can use the CLOSE verb to perform a database (DBCLOSE mode 1) or data set (DBCLOSE mode 2) close. Rewinding a data set (DBCLOSE mode 3) is not done by the CLOSE verb. However, rewinding the data set occurs automatically when serially (or reverse-serially) reading data sets for the FIND, GET, and OUTPUT statements. If the CLOSE verb is used in the middle of a dynamic transaction, a TurboIMAGE error message is issued and the transaction is aborted.

You can use the PROC statement to call DBCLOSE at anytime in a Transact program. A call to DBCLOSE mode 3 (rewind) is allowed within a dynamic transaction. However, if a call is made to DBCLOSE mode 1 or mode 2, a TurboIMAGE error message is issued and the dynamic transaction is aborted.

A database is usually not closed until the program ends, at which time Transact automatically closes the database. No transactions should be in process at this time. However, if a transaction is in process, the transaction will be aborted and an error message will be issued.

Database and File Locking

It is important, especially for online interactive applications, to establish a locking strategy at the time of system design. In general, locking is related to the *transaction*, the basic unit of work performed against a database. Typically, a transaction consists of several Transact statements to locate and modify data. Devising a locking scheme requires an understanding of locking levels, unconditional versus conditional locking, how Transact handles locking, and how access mode affects locking.

Locking Options Available with Transact

The Transact programmer has several options regarding database locking:

- Not locking at all by specifying NOLOCK on a SET(OPTION) statement.
- Allowing Transact to handle the locking automatically by using either:
  - Default locking—Transact locks at the database level. Choose this method of locking by setting *optlock* to 0 on the SYSTEM statement. This is the default setting and unconditional locking is used.

    OR

  - Optimized locking—locks at the database, data set, or entry level, depending on the operation. Choose this method of locking by setting *optlock* to 1 on the SYSTEM statement. Conditional locking is used.
- Using the LOCK option on the LOGTRAN statement. This allows you to specify locking for a database, a data set, or for several data sets across a logical transaction. It also allows you to specify conditional or unconditional locking. You must specify SET(OPTION) NOLOCK prior to the LOGTRAN statement to ensure that automatic locking does not operate for the verbs within the transaction.
- Using the PROC statement to call the database-locking intrinsics (DBLOCK and DBUNLOCK). See the *TurboIMAGE/V* or *TurboIMAGE/XL Database Management System Reference Manual* for more information.

With no locking or with automatic locking, you have the additional capability of specifying the LOCK option on individual database access verbs, as explained in "Using the LOCK Option with the Database Access Verbs" later in this chapter.

Avoiding Deadlocks in Transact Programs

A deadlock can also occur when there are conflicting locks within a single program. This is uncommon due to the error checking provided by Transact and the database, but can occur if you mix locking methods injudiciously. For example, suppose that within a Transact program you use DBLOCK through a PROC statement to lock a data set unconditionally. Then you issue a data access verb, that invokes automatic default locking, to request an unconditional lock on the database itself. The request for the database lock cannot complete because the data set lock must be released first. But the data set lock cannot be released because the program is waiting for the database lock to be granted. The result: deadlock due to conflicting locks within one program.

Caution

The special MPE capability, Multiple RIN (MR), is required to complete multiple, simultaneous locks on the same database. Transact provides this capability. Use extreme caution when employing a multiple-lock strategy. Hewlett-Packard does not accept responsibility for possible deadlocks or system lockouts that could result from the improper use of the MR capability.

Recommendations:

- Use multiple simultaneous locks *only* with conditional locking, *not* with unconditional locking.
- Use multiple simultaneous locks *only* if absolutely required, such as when locking more than one database.
- Use a consistent locking strategy. All programs using multiple, simultaneous locks and concurrent access should lock *at the same level* and *in the same order*.

In addition, if you are using Transact with unconditional locking and multiple, simultaneous locks, there are two situations you should avoid or, at least, handle very carefully. They are:

- Using a file equation so as to have two names for the same database, data set, or data entry.
- Combining Transact Default Locking with a PROC statement to call DBLOCK and lock a data set or entry unconditionally.

With Default Locking, Transact uses unconditional locking at the database level. If you implement multiple simultaneous locks and allow concurrent database access, either you must develop a locking strategy that does not deadlock, or you must use one of the other methods of locking available with Transact.

With Optimized Locking, Transact combines conditional locking at the appropriate level with a retry loop, so that your program seems to wait until the lock is granted. However, Transact retains control. If you press (Ctrl) Y, the program is interrupted and Transact returns to command mode with a > prompt. There is no possibility of a deadlock.

You can use the LOCK option on the LOGTRAN statement to specify whether to use conditional or unconditional locking and what locking level to use. Transact keeps track of any locks applied for transaction locking and returns an error message if you attempt to issue conflicting locks. To take full advantage of this protection, we recommend that you do not combine other methods of locking with transaction locking within the same program.

Of course, if you choose not to use locking, creating a deadlock is impossible. This is only recommended if combined with read-only or exclusive database access. Otherwise, you cannot ensure data integrity.

Understanding the Optimized Locking Scheme

All locks applied by optimized locking are conditional. Table 6-1 shows the rules that Transact uses for optimized locking when determining the level at which to lock. However, the Transact programmer need not know these rules, since the rules are applied automatically. The rules are presented for the programmer who needs to know the locking level to assure the logical integrity of the database and to conform to a pre-established locking strategy.

Table 6-1 summarizes the conditions affecting the locking level. The first four columns present the conditions as boxes, alternate possibilities appearing in the same column. For example, if you are using the DELETE verb, you would pick the second box in the first column. To the right of this box, you would find two boxes representing two groups of modifiers. If you are using the CHAIN modifier, you would choose the top box. There are two boxes to the right of this box. If a PERFORM appears in the DELETE statement, you would choose the Using PERFORM? box. If the DELETE statement includes a LOCK, then you would choose the Using LOCK? box in the next column. In the final column, the Locking Level box, the entry "Database" indicates that the combination "DELETE(CHAIN) ... , LOCK,PERFORM= ... " results in a database level lock.

Table 6-1 Automatic Locking Using the Optimized Locking Scheme

Verb	Modifier	Using PERFORM?	Using LOCK?	Locking Level
FIND OUTPUT	CHAIN RCHAIN	Yes	Yes	Data Base
			No	No Lock
		No	Yes	Entry
			No	No Lock
	None DIRECT CURRENT SERIAL RSERIAL	Yes	Yes	Data Base
			No	No Lock
		No	Yes	Data Set
			No	No Lock
DELETE REPLACE	CHAIN RCHAIN	Yes	Yes	Data Base
			No	Data Base
		No	Yes	Entry/Set*
			No	Entry/Set*
	None DIRECT CURRENT SERIAL RSERIAL	Yes	Yes	Data Base
			No	Data Base
		No	Yes	Data Set
			No	Data Set
PUT UPDATE	None	Not Applicable	Yes	Data Set
			No	Data Set
GET	CHAIN RCHAIN	Not Applicable	Yes	Entry
			No	No Lock
	None DIRECT CURRENT SERIAL RSERIAL	Not Applicable	Yes	Data Set
			No	No Lock

* The REPLACE verb locks at the entry level with the UPDATE option and at the set level without the UPDATE option.

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.

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;
```

Dynamic Roll-Back

The TurboIMAGE/iX dynamic roll-back feature is supported by Transact/iX. This section briefly explains how dynamic roll-back works.

TurboIMAGE/iX, through the use of Transaction Manager (XM), allows uncommitted logical transactions to be rolled back dynamically (online) while other database activity is occurring. This feature is accomplished through the use of three TurboIMAGE intrinsics: DBXBEGIN, DBXEND, and DBXUNDO. These intrinsics are supported in Transact/iX by using the LOGTRAN verb with modifiers XBEGIN, XEND, and XUNDO.

Transact can compile dynamic roll-back code correctly on MPE V and in compatibility mode on MPE/iX, but will issue a warning indicating that this is not supported. Transact/V and Transact/CM will issue an error message if they encounter the dynamic roll-back instruction p-code while processing the p-code file.

For more detailed descriptions of dynamic roll-back, see the *TurboIMAGE/XL Database Management System Reference Manual.*

Locking

To use dynamic roll-back, TurboIMAGE requires that applications have strong locking (second level consistency). Strong locking means that DBLOCK is the first call for the transaction and DBUNLOCK is the last call in the dynamic transaction. (DBUNLOCK should be called immediately after DBXEND.) Because of this requirement, a dynamic transaction does not allow the database to be opened in mode 2, which does not enforce locking.

Transact requires two events for dynamic roll-back transactions:

- Automatic locking must be disabled by issuing a SET(OPTION) NOLOCK statement prior to the LOGTRAN statement. If this is not done, strong locking will not be possible and an error message will be issued.
- Assuming that SET(OPTION) NOLOCK has been issued, Transact automatically performs strong locking when the LOCK option is used with LOGTRAN(XBEGIN).

If the LOCK option is not included on the LOGTRAN(XBEGIN) verb, the user takes all responsibilities for locking.

If the user performs redundant locks or unlocks in the transaction before ending it (if opened in mode 1), a TurboIMAGE error message is issued and the transaction is aborted.

If the necessary locking is not done prior to the LOGTRAN(XBEGIN), the transaction cannot begin and a TurboIMAGE error message is issued.

If an error occurs inside a database transaction, any database access verbs used after that error will return a status of -222. (See the *TurboIMAGE/XL Database Management System Reference Manual* for an explanation of return status.) Only a LOGTRAN(XUNDO) is allowed when a database transaction encounters an error.

Examples of Locking Strategy With LOGTRAN

The first example is a simplified roster program for an airline reservation system. Each flight has a master data set record that contains the number of seats available on that flight. This record is linked to a detail data set chain containing the list of passengers. The data dictionary is assumed to contain all the necessary data about the database.

The ADD FLIGHT command sequence uses a PUT statement to add a flight record to the master data set. NO-OF-SEATS contains the total number of seats on the flight when the PUT statement is executed. Since optimized locking is in effect, FLIGHT-MAST is locked at the data set level. If optimized locking were not specified, the entire database would be locked.

The ADD PASSENGER command sequence uses a GET statement to retrieve the number of seats available on that flight from FLIGHT-MAST. Automatic locking would lock at the data set level if the LOCK option were specified, but in this example no locking is done at all. If the number of seats is not zero, then the program subtracts 1 from NO-OF-SEATS and uses an UPDATE statement to update the flight record and a PUT statement to add a passenger record to ROSTER-DETL. Otherwise, a message is issued and a record is not added.

```
SYSTEM FLIGHT,BASE=FLIGHT("SEATS",1,1)
$$ADD:
  $FLIGHT:
    PROMPT(PATH) FLIGHT;
    PROMPT NO-OF-SEATS("Total number of seats available");
    PUT FLIGHT-MAST;

  $PASSENGER:
    PROMPT PASSENGER;
    PROMPT(PATH) FLIGHT;
    LIST NO-OF-SEATS;
    GET FLIGHT-MAST,LIST=(@);
    IF STATUS=0 THEN
    IF (NO-OF-SEATS) <> 0 THEN
      DO
        LET (NO-OF-SEATS) = (NO-OF-SEATS) - 1;
        UPDATE FLIGHT-MAST,LIST=(@),STATUS;
        IF STATUS=0 THEN
          PUT ROSTER-DETL,LIST=(PASSENGER:FLIGHT);
      DOEND
    ELSE
      DO
        DISPLAY "Sorry, no more seats available."
      DOEND;

$$ROSTERS:
    LIST(AUTO) FLIGHT-MAST;          << Flight,no-of-seats,passenger >>

    FIND(SERIAL) FLIGHT-MAST,PERFORM= OUTPUT-ROSTER,LIST=(@);
    END(SEQUENCE);

  OUTPUT-ROSTER:
    DISPLAY FLIGHT:NO-OF-SEATS;
    SET(KEY) LIST(FLIGHT);
    FORMAT PASSENGER;
    OUTPUT(CHAIN) ROSTER-DETL,LIST=(PASSENGER);
    RETURN;
```

If the program relies only on automatic locking, a problem can arise with this transaction in a multi-user environment. If another process alters the same flight record in FLIGHT-MAST between the GET statement and the UPDATE statement, then the quantity in NO-OF-SEATS will be incorrect. Also, other processes accessing both FLIGHT-MAST and ROSTER-DETL could get erroneous results between the UPDATE statement and the PUT statement in the ADD PASSENGER transaction because the database is in an inconsistent state. To insure data integrity both the FLIGHT-MAST and ROSTER-DETL data sets should be locked just before the GET statement and unlocked at the end of the command sequence.

The following code does that:

```
SET(OPTION) NOLOCK;

LOGTRAN(BEGIN) $HOME, "Add passenger transaction",
  LOCK(FLIGHT-MAST,ROSTER-DETL);

GET FLIGHT-MAST,LIST=(@);

IF (NO-OF-SEATS) <> 0 THEN
  DO
    LET (NO-OF-SEATS) = (NO-OF-SEATS) - 1;
    UPDATE FLIGHT-MAST,LIST=(@),STATUS;
    IF STATUS=0 THEN
     PUT ROSTER-DETL,LIST=(PASSENGER:FLIGHT);
  DOEND
ELSE
  DO
    DISPLAY "Sorry, no more seats available."
  DOEND;

LOGTRAN(END) $HOME, "End of add passenger";
RESET(OPTION) LOCK;
```

Note that the LOGTRAN(END) statement is placed so that Transact encounters the statement no matter which way the IF statement branches.

Finally, the ROSTERS command sequence (see preceding example) uses FIND and OUTPUT statements to produce a report. FLIGHT-MAST supplies the flight identification and number of available seats; ROSTER-DETL supplies the names of the passengers. Automatic locking does no locking because these are reporting statements and the LOCK option was not included. If the FIND statement had LOCK on it, Transact would lock the entire database for the entire FIND(SERIAL) because of the PERFORM= modifier. A LOGTRAN(BEGIN) statement with a LOCK option including the two data sets can be added just before the FIND(SERIAL). This would lock just the two data sets for the entire transaction. Then a LOGTRAN(END) statement immediately after the FIND(SERIAL) would unlock the data sets.

The next example is the same as the example above, except that it includes dynamic transaction logging instead of user logging.

To insure data integrity, both FLIGHT-MAST and ROSTER-DETL data sets are locked prior to the GET statement (by the LOGTRAN verb). If something goes wrong while adding a passenger to the ROSTER-DETL or updating the NO-OF-SEATS in the FLIGHT-MAST data set, the transaction will be rolled back to the state it was in prior to the LOGTRAN(XBEGIN).

The following example shows how dynamic transactions are used:

```
SET(OPTION) NOLOCK;

LOGTRAN(XBEGIN) $HOME, "Add passenger transaction",
  LOCK(FLIGHT-MAST,ROSTER-DETL);

GET FLIGHT-MAST,LIST=(@);

IF (NO-OF-SEATS) <> 0 THEN
  DO
    LET (NO-OF-SEATS) = (NO-OF-SEATS) - 1;
    UPDATE FLIGHT-MAST,LIST=(@),STATUS;
    IF STATUS = 0 THEN
      PUT ROSTER-DETL,LIST=(PASSENGER:FLIGHT),STATUS;
  DOEND
ELSE
  DO
    DISPLAY "Sorry, no more seats available."
  DOEND;

IF STATUS <> 0 THEN
  LOGTRAN(XUNDO) $HOME, "Roll-back passenger - cannot add"
ELSE
  LOGTRAN(XEND) $HOME, "End of add passenger";

RESET(OPTION) LOCK;
```

Limitations

Transaction Manager (XM) allows a transaction to be up to 1 MB in length. This poses a limitation for transactions. Some Transact verbs perform many iterations within a statement, such as DELETE(CHAIN) and REPLACE(SERIAL), in which transactions can reach the 1 MB limit. If that limit is reached, the dynamic transaction is aborted and the partial transaction is backed out.

The Transact verbs affected by dynamic transactions are PUT, UPDATE, DELETE, and REPLACE. The PUT and UPDATE verbs are single iteration verbs so the limitation should not affect them. However, DELETE and REPLACE are affected by this limitation so you may want to use the following workarounds:

DELETE Verb

The DELETE verb can have many iterations when using the SERIAL, RSERIAL, CHAIN, or RCHAIN modifier that can cause the Transaction Manager limitation to be reached. If there is a strong possibility of reaching the limit, you should use the FIND verb with the PERFORM option as a workaround. The PERFORM routine should call dynamic transaction logging for every *n* number of DELETE(CURRENT) records (where *n* is the number or records deleted for each dynamic transaction that is shorter in length than the limitation), or select a smaller set of records to delete by using the match register. For more information, see the *TurboIMAGE/XL Database Management System Reference Manual*.

REPLACE Verb

The REPLACE verb can have many iterations when using the SERIAL, RSERIAL, CHAIN, or RCHAIN modifier that can cause the Transaction Manager limitation to be reached. If there is a strong possibility of reaching the limit, you should use the FIND verb with the PERFORM option as a workaround. The PERFORM routine should call dynamic transaction logging for every *n* number of REPLACE (CURRENT) records (where *n* is the number or records deleted for each dynamic transaction that is shorter in length than the limitation), or select a smaller set of records to replace by using the match register.

Database Unlocking

The first call to DBUNLOCK unlocks *all* previously-set locks for that database. The first call to DBUNLOCK can be issued in several ways:

- A PROC statement.
- A LOGTRAN(XEND) statement, provided that the LOCK option was specified on the LOGTRAN(XBEGIN) statement.
- A LOGTRAN(XUNDO) statement, provided that the LOCK option was specified in the LOGTRAN(XBEGIN) statement.
- The end of the Transact/iX program.

If you are calling DBLOCK and DBUNLOCK with the PROC statement, we recommend that you do all your locking/unlocking with the PROC statement. Combining LOGTRAN locking/unlocking with calls to DBLOCK/DBUNLOCK can produce unexpected results or errors.

Using KSAM and MPE Files

When using KSAM and MPE files, it is best to define a buffer record whose child items are the pieces of the record. Then you can read, write, or list the record by the buffer name and also refer to the items individually.

Defining a Buffer Record

The following example shows how to define a buffer record that contains all fields.

```
SYSTEMS FREC, FILE= WORK(ACCESS(R/W),40,3,100);

  DEFINE(ITEM) BUFFER X(80):
               ITEM1  X(25) = BUFFER(1):
               ITEM2  X(30) = BUFFER(26):
               ITEM3  X(15) = BUFFER(56):
               ITEM4  X(10) = BUFFER(71);

  LIST BUFFER;

  GET(SERIAL) WORK, LIST=(BUFFER);

  DISPLAY ITEM1:
          ITEM2:
          ITEM3:
          ITEM4;

  DATA(SET) ITEM1:
            ITEM2:
            ITEM3:
            ITEM4;

END FREC;
```

The LIST= option must contain either the entire KSAM record buffer or some beginning portion of the record. In other words, Transact must know where the beginning of the buffer is to calculate key offsets, since the first item in the LIST= option is used to determine where the record starts. Transact needs this information to call FFINDBYKEY, which requires the key position as one of its parameters.

Table 6-4 Understanding the KSAM Interface

KSAM File Access			
Transact Verb	**Key Type**	**File Defined As:**	
		MPE File	**KSAM File**
FIND(SERIAL)	N/A	Primary key sequence	Chronological sequence
FIND(CHAIN)	PRIMARY	Reads chronological record zero and then primary key sequence until EOF	Primary key sequence
	SECONDARY	Same as primary	Secondary key sequence
FIND	None	One record primary key sequence	Error
	PRIMARY	Same as no key	One record of primary key value
	SECONDARY	Same as no key	One record of secondary key value

General Format for Key-Driven Access

With the construct shown below, you can set up a primary or secondary key (even a generic search value) and read subsequent values in the key sequence. The only thing you cannot do is set up a key value that doesn't exist. The first GET(CHAIN) determines the starting position in the file.

```
SYSTEM NAME, KSAM=FILENAME;              << OPEN AS A KSAM FILE         >>
DEFINE(ITEM) RECORD 80 X(1):
             KEYN      X(5) = RECORD(1);
LIST RECORD;

MOVE (KEYN) = "VALUE";                   << GIVE KEY A VALUE             >>

SET(KEY) LIST(KEYN);                     << SET UP KEY/ARGUMENT REGISTERS >>

GET(CHAIN) FILENAME, LIST=(RECORD);      << USE CHAINED ACCESS ON PRIMARY >>
                                         << KEY                          >>
REPEAT
  DO
  GET(CHAIN) FILENAME, LIST=(RECORD), STATUS;
  DOEND
UNTIL STATUS <>0;

END NAME;
```

Traversing a KSAM File by Primary Key

In this example, a KSAM file is read in primary key sequence. Compare it with a later example of how to read an MPE file by primary key.

```
SYSTEM KPLAY4, KSAM=KTRAN3;                << OPEN AS A KSAM FILE          >>

DEFINE(ITEM)  KARRAY 80 X(1):
              KEY1      X(5)= KARRAY(1);   << DEFINE PRIMARY KEY           >>

LIST KARRAY,INIT;

MOVE (KEY1)= "$$$$$";                      << GIVE KEY A VALUE             >>

SET(KEY) LIST(KEY1);                       << SETUP KEY/ARGUMENT REGISTERS >>

FIND(CHAIN)   KTRAN3,LIST=(KARRAY),        << USE CHAINED ACCESS ON        >>
              PERFORM=DISP;                << PRIMARY KEY                  >>

EXIT;

DISP:
  DISPLAY KARRAY;
  RETURN;

END KPLAY4;
```

Traversal by Alternate Key

Here the file is accessed by alternate key, where the key value is AAAAE.

```
SYSTEM KPLAY4, KSAM= KTRAN3;                << OPEN AS A KSAM FILE         >>

DEFINE(ITEM) KARRAY 80 X(1):
             KEY2      X(5) = KARRAY(2);    << DEFINE ALTERNATE KEY        >>

LIST KARRAY, INIT;

MOVE (KEY2)= "AAAAE";                       << GIVE KEY A VALUE            >>

SET(KEY) LIST (KEY2);                       << PUT KEY AND VALUE IN        >>
                                            << KEY/ARGUMENT REGISTERS      >>

FIND(CHAIN) KTRAN3,LIST=(KARRAY),           << USE CHAINED ACCESS ON       >>
            PERFORM=DISP;                   << ALTERNATE KEY               >>

EXIT;
```

```
DISP:
  DISPLAY KARRAY;
  RETURN;

END KPLAY4;
```

General Format for Generic Keys

The way to set up the key register for a generic search is to define a child item that is the size of the search string and set up the Key/Argument registers with its name or value. For example:

```
SYSTEM NAME, KSAM=FILENAME;                 << OPEN AS A KSAM FILE        >>


DEFINE(ITEM) RECORD X(80):                  << 80 BYTE RECORD             >>
             KEY1   X(5) = RECORD(1):       << KEY IS FIRST FIVE CHAR     >>
             GEN    X(2) = RECORD(1);       << GENERIC SEARCH ITEM        >>

LIST RECORD,INIT;

MOVE (GEN) = "AB";                          << DEFINE GENERIC SEARCH VALUE >>

SET(KEY) LIST(GEN);                         << SETUP KEY/ARGUMENT REGISTER >>

FIND(CHAIN) FILENAME,LIST=(RECORD);         << USE CHAINED ACCESS ON      >>
                                            << GENERIC KEY                >>

END NAME;
```

Search with Generic Key

Generic keys must be equal to or less than the length of the search value.

```
SYSTEM KPLAY4, KSAM=KTRAN3;                 << OPEN AS A KSAM FILE        >>

DEFINE(ITEM) KARRAY 80 X(1):
             KEY       X(5) = KARRAY(2):    << ALTERNATE KEY              >>
             KGEN      X(2) = KARRAY(2);    << GENERIC SEARCH VALUE       >>

LIST KARRAY, INIT;

DATA KGEN;                                  << RETRIEVE GENERIC SEARCH    >>
                                            << VALUE                      >>
```

```
SET(KEY) LIST(KGEN);                             << SETUP KEY/ARGUMENT REGISTER >>
                                                 << WITH SEARCH VALUE          >>

FIND(CHAIN) KTRAN3,LIST=(KARRAY),                << USE CHAINED ACCESS ON      >>
            PERFORM=DISP;                        << KSAM FILE                  >>
EXIT;

DISP:
  DISPLAY KARRAY;
  RETURN;

END KPLAY4;
```

Simulating an Approximate Key Search

Transact does not support true approximate key searches. This example shows a method of using Transact to achieve the same results as with an approximate key search.

The first GET(CHAIN) determines the starting position in the file. Subsequent reads follow the key sequence and read all values greater than or equal to the key value defined. The initial key value must exist.

```
SYSTEM APROX,KSAM=KTRAN3;                        << OPEN AS KSAM FILE          >>

DEFINE(ITEM) KARRAY 80 X(1):
             KEY1      X(5)=KARRAY(1);           << DEFINE KEY                 >>

LIST KARRAY, INIT;

MOVE (KEY1)= "SSSSS";                            << GIVE KEY INITIAL VALUE     >>

SET(KEY) LIST(KEY1);                             << SETUP KEY/ARGUMENT REGISTER >>

GET(CHAIN) KTRAN3, LIST=(KARRAY);                << USE GET(CHAIN) TO READ     >>

DISPLAY;                                         << FIRST RECORD               >>

REPEAT          << USE GET(CHAIN) WITH STATUS TO CONTINUE READING IN KEY   >>
                << SEQUENCE AND TO PREVENT STOPPING WHEN KEY VALUE CHANGES >>
  DO
    GET(CHAIN) KTRAN3, LIST=(KARRAY),STATUS;
    DISPLAY;
    INPUT "DO YOU WANT TO CONTINUE (YES/NO)?";
  DOEND

UNTIL INPUT = "NO";

END APROX;
```

Chronological Traversal of a KSAM File

The file is read in chronological sequence using FREADC.

```
SYSTEM KPLAY4, KSAM= KTRAN3;              << OPEN AS A KSAM FILE      >>

DEFINE(ITEM) KARRAY 80 X(1):

LIST KARRAY, INIT;

FIND(SERIAL) KTRAN3, LIST=(KARRAY),       << ACCESS THE FILE SERIALLY   >>
             PERFORM=DISP;

EXIT;
DISP:
    DISPLAY KARRAY;
    RETURN;
END KPLAY4;
```

IPC Files

If your Transact program is using IPC (message) files, it is important to remember that one program can open a message file only for either reading or writing, not for both. Therefore, if two processes are reading and writing to each other, two message files are needed.

The following example shows two programs, both accessing the same IPC file called MSG1. The first program uses the file for read only, and the second for write only. This message file was built outside of Transact with the MPE BUILD command, for example:

```
BUILD MSG1;MSG
```

The read program uses the GET statement instead of the FILE(READ) statement, so that STATUS can be used to control error handling. If automatic error handling is used (such as no STATUS option), the program aborts when it gets to the end of file.

```
READ PROGRAM

     SYSTEM MSGR,FILE=MSG1(READ(OLD));

     DEFINE(ITEM) E X(256);

     LIST E,INIT;

     LOOP1:
     GET(SERIAL) MSG1,LIST=(E),STATUS;
     IF STATUS=-1 THEN
       DO
       END;       << END PROGRAM >>
       DOEND;
     DISPLAY;
     GO TO LOOP1;

WRITE PROGRAM

     SYSTEM MSGW,FILE=MSG1(WRITE(OLD));

     DEFINE(ITEM) E X(256);

     LIST E,INIT;

     LOOP1:
     DATA E;
     FILE(WRITE) MSG1,LIST=(E);
     DISPLAY;
     GO TO LOOP1;
```

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

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);

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

(4)  FIND (DATA-ITEM);

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

Consider the following:

1. The user enters the data item in response to PROMPT in (2).
2. The program increments the counter from 0 to 1 in (3).
3. The data item entered in (2) causes an error in the FIND operation in (4).
4. Automatic error handling causes the program to repeat (2).
5. The user receives an error message from the database and is prompted again for DATA-ITEM.
6. The user enters the corrected DATA-ITEM.
7. The program increments the counter from 1 to 2 in (3).
8. The FIND operation in (4) is now successful.
9. The PUT operation in (5) stores the count of 2—which may not be the desired result.

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

There are some conditions under which the ERROR= option is not taken when no entry is found. This information is summarized in Table 7-1.

Table 7-4 shows the contents of the status register following a database or file access statement when the STATUS option is not used.

You can test the contents of the STATUS register with an IF statement within your own error routines at a label specified by the ERROR= option. You can display the contents of the register by first assigning it to a 32-bit data item. The data item should be type `I(10,,4)` to hold the maximum STATUS value. For example:

```
DEFINE(ITEM) STAT I(10,,4);
LIST STAT;
LET (STAT) = STATUS;
DISPLAY STAT;
```

In addition to branching, the ERROR= option sets a value to the status register to identify the type of error.

**Table 7-1.
Circumstances that Determine Whether ERROR= Branch Is Taken during Database and File Operations**

Modifier	Chain RChain		Serial RSerial	Direct	Current	Primary	None	
Database or File Error	**No entry**	**No master**	**No record**	**Invalid record number**	**No current record defined**	**No master**	**No entry**	**No master**
Delete	N	Y	N	Y	Y	Y	N/A	Y
Delete (S)	Y	Y	Y	Y	Y	Y	N/A	Y
Find	N	N	N	Y	N	N	N/A	N
Find (S)	Y	Y	Y	Y	N	N	N/A	N
Get	Y	Y	Y	Y	Y	Y	N/A	Y
Get (S)	Y	Y	Y	Y	Y	Y	N/A	Y
Output	N	Y	N	Y	Y	Y	N/A	Y
Output (S)	Y	Y	Y	Y	Y	Y	N/A	Y
Path	N/A	N/A	N/A	N/A	N/A	N/A	N	N
Path (S)	N/A	N/A	N/A	N/A	N/A	N/A	N	Y
Put	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Y
Put (S)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Y
Replace	N	Y	N	Y	Y	Y	N/A	Y
Replace (S)	Y	Y	Y	Y	Y	Y	N/A	Y

Y = Taken
N = Not taken
(S) = Status option used

Arithmetic Calculations

Arithmetic errors can be handled within a Transact program by including the ERROR= option on the LET verb. (See a complete discussion under LET in Chapter 8.)

Using the STATUS Option

You can disable several aspects of Transact's automatic processing by using the STATUS option. Using the STATUS option causes the status register to be set differently then when the STATUS option is not used. You can then test the contents of the status register (by using an IF statement) before deciding what further processing should be done. The STATUS option has a different effect depending on whether the statement in which it appears performs data entry or accesses a database or file.

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

```
LET STATUS = 0;
```

Data Entry Errors

Under automatic error handling, the status register contains the number of characters entered in response to the data entry verbs DATA, INPUT, and PROMPT.

When the user enters "]" or "]]" and the verb does not have a STATUS option, an escape to the next processing level is generated as discussed in the "Data Entry Control Characters" section in Chapter 5, "User Interface." The STATUS option suppresses the escape and allows you to test the contents of the register before continuing processing.

Table 7-2 shows the contents of the status register when a data entry verb is used with and without the STATUS option.

Table 7-2. Contents of Status Register After Data Entry Verbs

User Entry	Status Register with no STATUS Option	Status Register with the STATUS Option
<CR>	0	0
ABC	3	3
blanks	-3	-3
timeout	-4	-4
]	escape	-1
]]	escape	-2

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

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

Database or File Operation Errors

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

Status Register Value	Meaning
0	The operation was successful.
-1	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 the database condition word or KSAM file system error documentation corresponding to the value.
1	If NOFIND option or the GET verb is used and the record is found.

In addition, STATUS has the following effects:

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

Table 7-3 summarizes the effect of STATUS with database and file operation verbs.

Table 7-3. STATUS Option with Database and File Operation Verbs

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

Table 7-4 shows the contents of the status register when a data management verb is used without the STATUS option.

Table 7-4.
Contents of Status Register Following Operations of Data Management Verbs when STATUS Option Is NOT USED

Verb	Status Register Value	
	Operation Successful	**Operation Not Successful**
DELETE FIND OUTPUT REPLACE	Number of entries or records selected (not necessarily number retrieved)	0 = No entries or records found* -1 = No master entry (FIND(CHAIN) and FIND(RCHAIN) only*) Otherwise undefined
GET PUT UPDATE	0 = One entry or record found	-1 = Entry not found Otherwise undefined
FILE(READ)	Number of bytes read Otherwise undefined	-1 = End of file
PATH	Number of records in detail data set chain	0 = No detail set chain* -1 = No master entry Otherwise undefined
FILE(CLOSE) FILE(CONTROL) FILE(SORT) FILE(UPDATE) FILE(WRITE)	0 = Successful operation	Undefined

* Entry not found does not always activate the ERROR= option; see Table 7-1.

For additional information about the STATUS option with these verbs, see Chapter 8, "Transact Verbs."

Compiler Error Messages

The compilers for Transact/V and Transact/iX generate error messages with different formats.

Transact/V Error Message Formats

- Errors that resulted in the generation of no p-code or erroneous p-code. Unless you have specified the XERR compiler control option, no p-code file is produced.

```
*** ERROR# n        ^(error-info) error-message
```

- Conditions detected by the compiler that do not completely follow Transact syntax rules. However, they are correctable by the compiler in generating the p-code file.

```
** WARNING# n       ^(error-info) error-message
```

where "^" is positioned under the location in the statement line where the compiler detected the error condition, and where *n* is the total number of errors reported at this point of the compile of the program.

Transact/iX Error Message Formats

The Transact/iX compiler generates three types of error messages in addition to those generated by the Transact/V compiler:

```
{ *ERROR:   }
{ *INFO:    } error-message (error-info)
{ *WARNING: }
```

Error-Info

Error-Info takes the following form:

(*type number*) [*code-location*]

where:

type — One of the following:

Transact/V	TVA
Transact/iX	TXF, TXG

number — The error number for types TVA, TXF and TXG.

code-location — (For Transact/iX only) The internal location (in the program) at which the error occurred. (See the second column of numbers in the program-compilation listing.)

Run-time Error Messages

Transact generates two types of error message: one type indicates actual errors, the other type provides information to the user but does not indicate an actual error. Both types of message are described below and in the appropriate reference manual for the indicated subsystem.

Error Message Format

Error messages are displayed in the following format:

```
{ *ERROR: }
{ *INFO:  } error-message (error-info) [program-name]
```

Information messages are not errors but are conditions that the processor will tell the user about. Also, messages that occur only in test modes are type INFO.

Error-Info

Whether appearing in an error or an information message, *error-info* can contain up to five of the following fields:

(*type* *number* [,*code-location*[,PARM(*n*)][,*file-name*]]

where

type is one the following:

Transact/V	TVB
Transact/iX	TXA, TXB, TXC, TXD, TXE, TXH, TXI, TDEBUG
	The following error types are derived from the indicated subsystem. Consult the appropriate reference manual for an explanation of the error condition.
IMAGE	IMAGE/V or TurboIMAGE/V database error.
KSAM	KSAM utility error or file system error while operating on a KSAM file.
MPEF	MPE file system error.
VPLUS	VPLUS data entry utility error.

number The error number for types TVB, TXA, TXB, TXC, TXD, TXE, TXH, TXI, TDEBUG. For types IMAGE, KSAM, MPE, or VPLUS, it is a number meaningful to the indicated subsystem.

code-location The internal location in the program at which the error occurred. (See the second column of numbers on the program compilation listing.)

PARM(*n*) *n* is the field number on multiple data entry fields at which the error was detected. All of the following data entry fields are ignored.

file-name The name of the data set or file that was involved in the error condition.

Program-Name

program-name can be one of the following:

Transact/V	The system name from the SYSTEM statement. The last one to six characters of the file-name if the IP <name> file is renamed using an MPE command.
Transact/iX	The system name from the SYSTEM statement. The file-name containing the Transact/iX program if an error occurs before the system is established. The procedure name of a called subprogram in an MPE executable library (XL).

Using EXPLAIN

Use the EXPLAIN subsystem to obtain information about Transact error messages. EXPLAIN tells you what causes the errors and what actions you can take to fix them.

When using `EXPLAIN`, you can provide parameters interactively or from the MPE command line.

Command-line execution is convenient for looking up one message quickly. `EXPLAIN` performs the command specified on the command line and returns to the system prompt (:). For example, to display information for the error message `TVB 1030`, enter:

```
:EXPLAIN TVB 1030
```

If you have several messages to look up, you can run the program interactively by entering the `EXPLAIN` command without parameters. Look up one message after another by entering a message type and number each time you see the `EXPLAIN>` prompt. If several messages have the same type, you do not have to enter the type each time.

By default `EXPLAIN` uses the type of the last message explained or printed. You can enter commands to print messages, search the message catalog, get online help, or exit the program. For a list of your options at any prompt, type `?`. If you don't understand a prompt, enter `HELP` for an explanation.

The `PRINT` and `FIND` commands require additional information to execute. You are prompted for this information if you do not supply it when entering the command. `PRINT>` or `FIND>` precedes these prompts to remind you that you are not at the main `EXPLAIN>` prompt. To break out of these commands without completing them, enter CTRL `Y` or `EXIT` at any prompt.

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.

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.
prompt-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.
option-list	A field specifying how the data should be formatted and/or other checks to be performed on the entered value. Include one or more of the following options (separated by commas) unless you use the ITEM modifier (Syntax Option 2):

BLANKS	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)]}[("prompt-string")][,option-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-name*, 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) 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       X(10)= RECORD(3);

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

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

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

The key search is similar to that shown above; use 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.

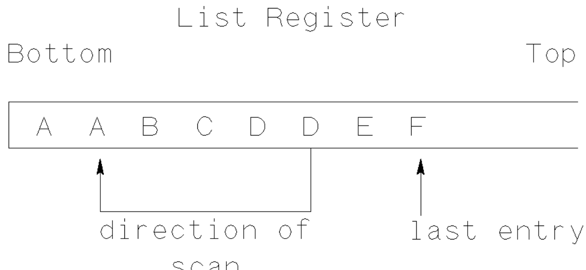
All item names specified must be parent items.

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.

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.
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. 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.
nM	Displays the first n letters of month name in uppercase; if n > number of letters in month name, trailing blanks are not inserted.
nm	Displays the first n letters of month name in lowercase except for the first letter, which appears in uppercase.
YY	Displays the last two digits in current year.
YYYY	Displays the current year.
nW	Displays the first n letters of day of week in uppercase; if n > length of the week name, no trailing blanks are inserted.
nw	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.

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, specify LINE=0. The default is 1. LINE=0 and LINE= (no number specified) cause a carriage return but no line feed. To accumulate output from several display statements on one line, use the parameter NOCRLF.

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.

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.
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.
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.
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=1.
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.

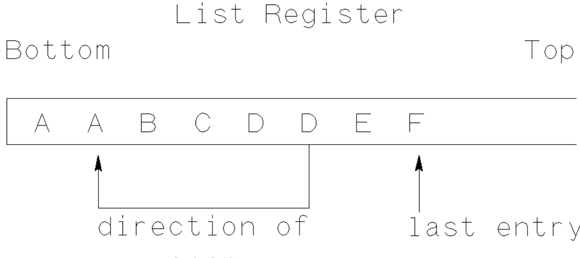
All item names specified must be parent items.

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.

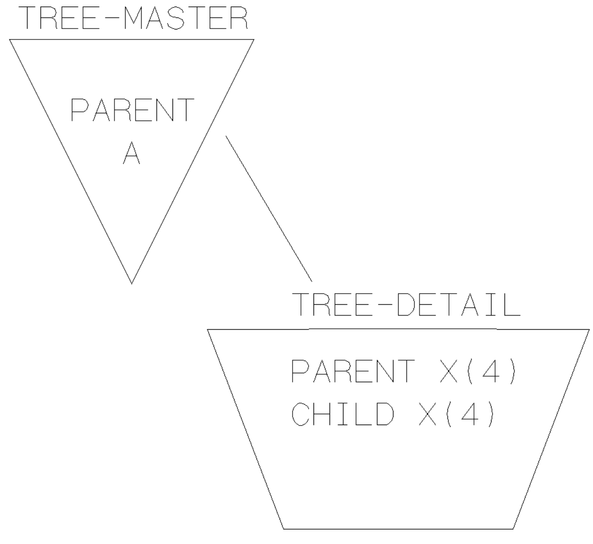
The following example uses a PERFORM= option to test data values in each retrieved entry. The routine TEST1 is performed on every record retrieved by FIND(CHAIN).

```
   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.

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.

  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*:

  ^ 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. idx|S and SS edit characters|

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.

nM Displays the first n letters of month name in uppercase; if n > number of letters in month name, trailing blanks are not inserted.

nm Displays the first n letters of month name in lowercase except for the first letter, which appears in uppercase.

YY Displays the last two digits in current year.

YYYY Displays the current year.

nW Displays the first n letters of day of week in uppercase; if n > length of the week name, no trailing blanks are inserted.

nw 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.

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 over print and you want it to do so, you should specify LINE=0. Line= gives a carriage return but no line feed. The default is 1.
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.
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.
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.
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.
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=1.
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.

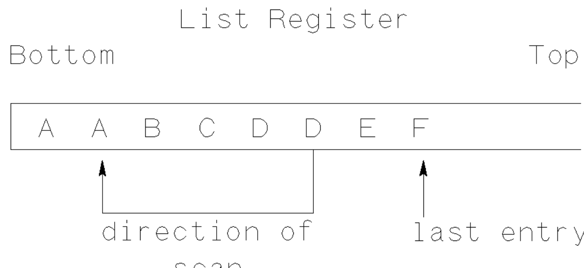
All item names specified must be parent items.

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 One 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. verb|

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 in this chapter 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.

---

source1 — 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

source# — The same as *source1*.

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 Handling" 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);
```

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 cause such a branch.

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

```
ASCII( { (item-name[(subscript)]) } )
       { "character-string"       }
```

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

```
LENGTH( { (item-name[ (subscript) ]) } )
        { "character-string"         }
```

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

```
LN( { (item-name[ (subscript) ]) } )
    { numeric-constant           }
```

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

LOG({ (*item-name*[(*subscript*)]) | *numeric-constant* })

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

```
POSITION( { (item-name1[ (subscript) ]) } , { (item-name2[ (subscript) ]) } )
          { "character-string1"         }   { "character-string2"         }
```

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

```
SQRT( { (item-name[(subscript)]) } )
      { numeric-constant         }
```

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

```
VALUE( { (item-name[ (subscript) ])  } )
       { "character-string"          }
```

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.

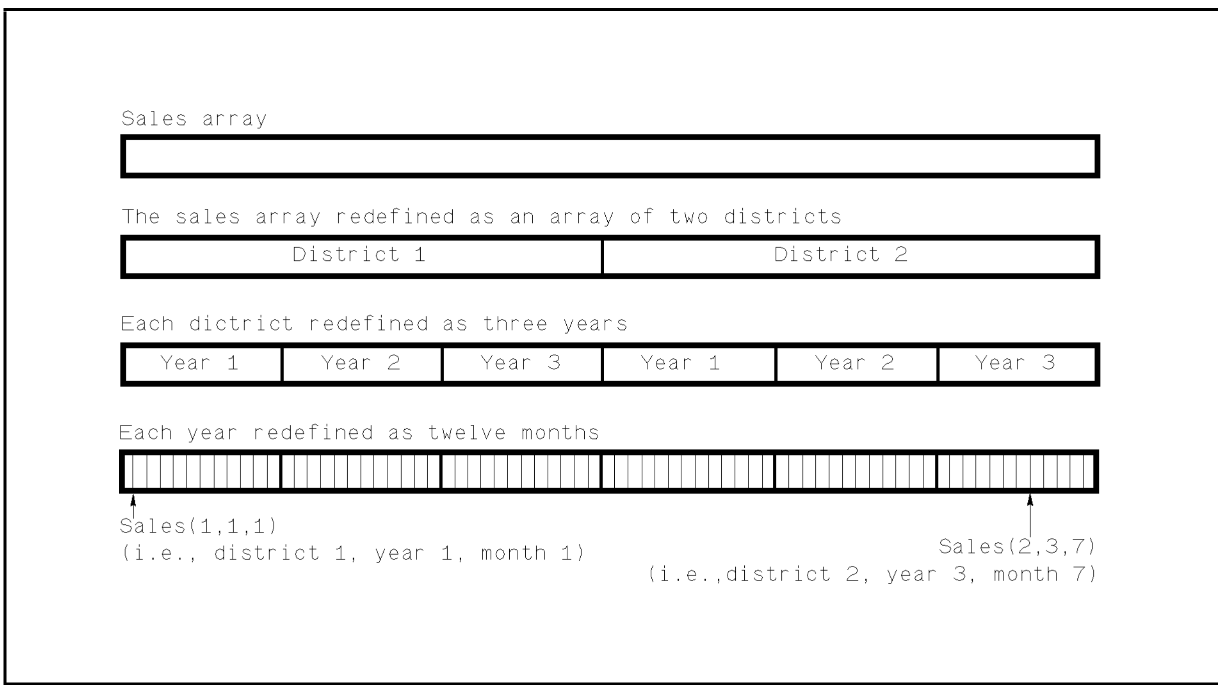


Figure 8-1. Complex Array of Sales Figures.

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 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.

Note DEFINE(ITEM) defines these items as 2-byte or 4-byte integers (I,J).

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.

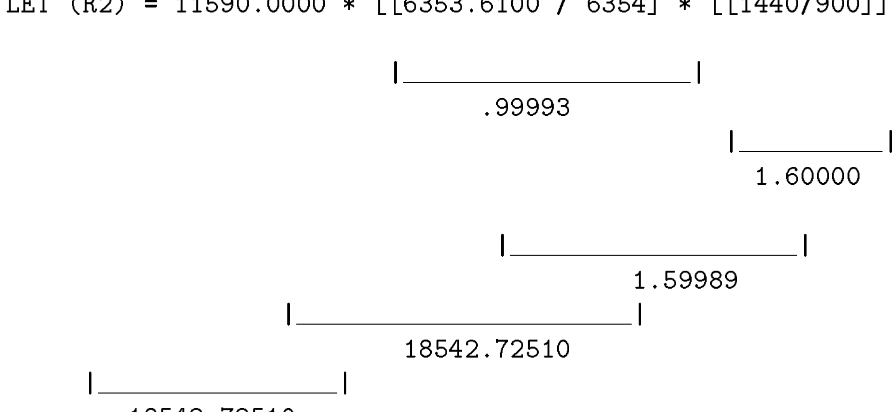
```
SYSTEM PAKDEC;
   DEFINE(ITEM) R1 R(6):         << No decimal place                 >>
                R2 R(11,5):      << More decimal places              >>
                I3 I(9,2);       << Fewer decimal places             >>
```

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]];

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

                              |____________________|
                                     .9999
                                                    |__________|
                                                            1

                                        |____________________|
                                               .9999
                        |_______________________|
                                 11588.8410
            |_________________|
                   11589   (ROUNDED)
```

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

```
LET (R2) = 11590.0000 * [[6353.6100 / 6354] * [[1440/900]];
```



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)
```

Performance Considerations

The following guidelines will help you optimize arithmetic operations in your Transact programs.

- It is most efficient to use 16-bit integer types (I or J) for single +, -, =, or negation operations.
- Use 64-bit reals (E or R) for single operations that include *, /, //, LN, LOG, SQRT, or exponentiation, as well as +, -, =, or negation operations.

- Use packed decimal types (P) for all other operations.
- Avoid mixing types within an operation.

Error Branching

The ERROR= option gives you the ability to handle arithmetic errors on calculation from within Transact. In addition to branching to the specified *label* and resetting the list register to (*item-name*), ERROR= causes the status register to be set to one of the following values that identifies the type of error.

Value	Type of Error
1	Attempt to assign a negative value to a positive item.
2	Invalid arithmetic field for item/invalid decimal digit.
3	Divide by zero.
4	Overflow.
5	Underflow.
6	LOG, LN, or SQRT function attempted on a negative number.

The following Transact errors cause the ERROR= branch to be taken:

User Errors:

16 - Attempt to assign negative value to an item.
17 - Invalid arithmetic field for an item.

Programmer Errors:

46 - Decimal divide by zero.
47 - Decimal overflow.
48 - Extended precision divide by zero.
49 - Extended precision underflow.
50 - Extended precision overflow.
51 - Integer overflow.
52 - Floating point overflow.
53 - Floating point underflow.
54 - Integer divide by zero.
55 - Floating point divide by zero.
76 - Attempted LN, LOG, or SQRT function on a negative number.
81 - Invalid decimal digit.
84 - Attempt SQRT function on a number that is less than zero.

The ERROR= option is very useful in trapping the above errors. Each of these errors will generate an error message and the program will continue. The ERROR= option allows the programmer to evaluate the error and determine how the program should proceed.

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

CHAR({ (*item-name*[(*subscript*)]) | *numeric-constant* })

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

```
COL( { (item-name[ (subscript) ]) } ,position)
     { "character-string"         }
```

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

```
LOWER( { (item-name[ (subscript) ]) } )
       { "character-string"         }
```

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

SPACE({ (*item-name*[(*subscript*)]) | "*character-string*" } , *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

```
        { (item-name[(subscript)]) }
STRING( {                          } ,position,length)
        { "character-string"       }
```

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

```
UPPER( { (item-name[ (subscript) ]) } )
       { "character-string"         }
```

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.

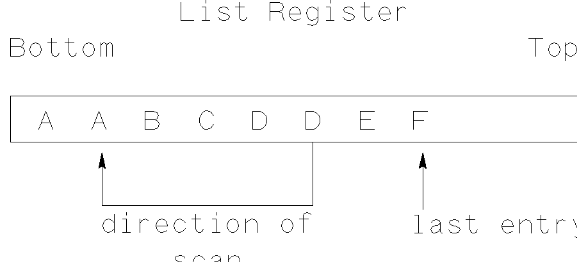
The options for *range-list* and the data items retrieved by OUTPUT include the following:

(*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 description.

```
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 to establish a KSAM key in a KSAM file or to establish a detail set for chained access. 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.

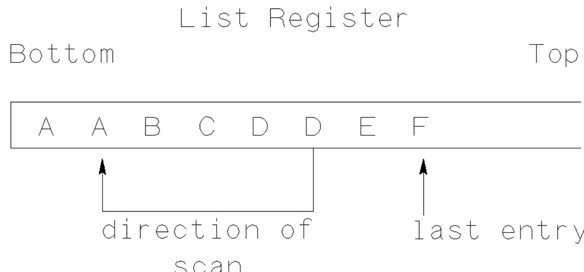
The LIST= option has a limit of 64 individually listed item names and a limit of 255 items specified by a range.

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 8-2 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 real and HP real 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 SL or XL.

parameter-list The items in the *parameter-list* specify one or more variables that are passed between the Transact program and the external procedure. The list can contain any number of variables, separated by commas. The order in which you place the variables is determined by the order in which the called procedure expects them. The only exception is that a function return variable can be placed anywhere in the list; a function return variable is indicated by a preceding "&".

The following special characters can precede any 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 a parameter by word address.

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.

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.

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 the 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 compiled 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 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.

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/XL converts hyphens to underscores. Therefore, unless precautions are taken, COBOL II subroutines that are recognized by Transact/V in compatibility mode will not be recognized when the subroutine is recompiled using COBOL II/XL and linked or loaded with a Transact/iX program. To make the names consistent, you can specify the COBOL II/XL compiler option that removes hyphens from COBOL subroutine names or you can use underscores instead of hyphens when naming any COBOL subroutines that will be used under Transact.

Option Variable Procedures

Since 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 in the compile. The bit map, included as part of the parameter list, is ignored and the remaining parameters are checked in accordance with their specification in SYSINTR.PUB.SYS. No implicit type conversions are performed.

User-defined option variable procedures must either be accessed via a switch to a compatibility mode routine or be converted to fixed parameter procedures and recompiled with a native mode compiler. The former 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, all null parameters for option-extensible intrinsics must be designated by single commas. 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.

One method of getting around this incompatibility is to modify the source so that 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. However, 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 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);
DEFINE(ITEM) COUNT I(4);
DEFINE(ITEM) CONTROL I(4);

LIST MSG : COUNT : CONTROL;

MOVE (MSG) = "HELLO THERE WORLD!!";
LET (COUNT) = -19;
LET (CONTROL) = 0;

PROC FWRITE (#FILEID(TRANOUT), (MSG), #(COUNT), #(CONTROL));
```

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;
```

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*];

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*];

PROMPT(KEY) places the *item-name* in the key register and the input value in the argument register. The data item and its value are used as a retrieval key for a subsequent data set or file operation.

(3) PROMPT(MATCH)*item-name*[("*prompt-string*")][,*option-list*];

PROMPT(MATCH) adds the *item-name* to the list and match registers. In addition, it adds the input value to the data register and also sets up this value as a selection criterion in the match register for a subsequent 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 CHECK NOT=, 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 "ˆ" on input
SCAN	Matched item must contain the input string; equivalent to the use of trailing "ˆˆ" on input
TRAILER	Matched item must end with the input string; equivalent to the use of a leading "ˆ" 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*];

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.

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*];

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, CHECKNOTCUST-MASTER;
      ⋮
      PUT CUST-MASTER;
   $TRANS:
      PROMPT(PATH) CUST-NUMBER, CHECKCUST-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.

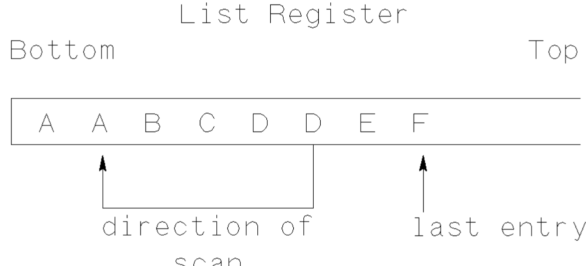
The options for *range-list* and the records upon which they operate include the following:

(*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.

PUT with the STATUS option could be used as shown in the following example. When a data set is full, you may want to write to an overflow file. To trap and display the full error condition, you could use the following code:

```
PUT DATA-SET,
    LIST=(A:N),
    STATUS;
IF STATUS <> 0 THEN             << Error, check it out             >>
    IF STATUS <> 16 THEN        << Unexpected error                >>
       GO TO ERROR-CLEANUP
    ELSE                        << Write to overflow               >>
       DO                       << Set full                        >>
          PUT OVERFLOW,
             LIST=(A:N),
             STATUS;
          IF STATUS <> 0 THEN
             GO TO ERROR-CLEANUP;
          DISPLAY "OVERFLOW FILE USED";
       DOEND;
```

Options Available 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 the single word integer *item-name*. The function key number is a digit from 1 through 8 for function keys f1 through f8, or zero for the ENTER key. Transact determines which function key was pressed from the value of the field LAST-KEY in the VPLUS comarea. 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),
     PERFORMLIST-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;
```

The last 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);
```

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.

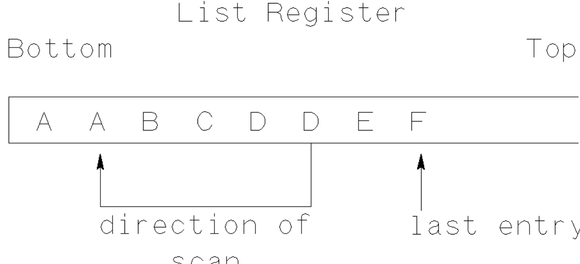
All item names specified must be parent items.

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.
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 PERFORME;     >>
IF(VALUE)="ALLAN" THEN

   RETURN(1);                    <<Transfers control to first        >>
                                 <<statement following PERFORMD;     >>
IF(VALUE)="BROWN" THEN

   RETURN(@);                    <<Transfers control to first        >>
                                 <<statement following PERFORMA;     >>
```

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. 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

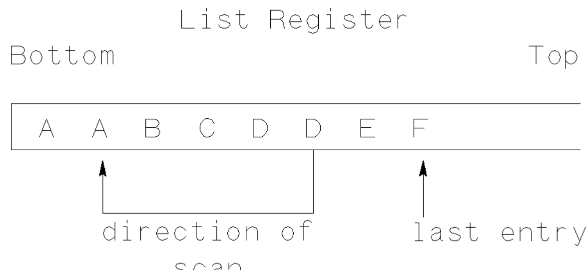
To ensure that the cursor will be positioned on the correct field, you must have a one to one correspondence between the fields defined in VPLUS. Transact determines where to position the cursor by counting the fields.

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.”

(6) SET(MATCH) LIST ({*item-name*})[,*option-list*];
{ * }

SET(MATCH) sets up a match criterion in the match register using the specified item name from the list register and its current value in the data register. If the item name is an unsubscripted array, only the value in the data register for the first element of the array will be set up as match criterion in the match register.

The resulting match criterion is used for subsequent data set and file operations. By default, the relation between the item name and its value is equality. You can choose another relational operator from *option-list*. If an * is specified, the last item added to the list register is used.

You can set up as many match criteria as you desire using separate SET(MATCH) statements for each. Match criteria set up with the same item name and no option, or the same item name and one of the options LEADER, TRAILER, or SCAN, are joined by a logical OR; those set up with different item names or with one of the options NE, LT, LE, GT, or GE are joined by a logical AND. (See the PROMPT(MATCH) and DATA(MATCH) descriptions in this chapter for other ways to set up match criteria.)

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

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

Examples of SET(MATCH)

This example sets up the match register with the selection criterion shown below:

```
LET (QTY-ON-HAND)  10;
SET(MATCH) LIST (QTY-ON-HAND), LT;
```

```
+-------------+
| QTY-ON-HAND |
|  less than  |
|     10      |
+-------------+
```

These statements set up the match register with the selection criteria shown below. Note that criteria with the same item name are joined by a logical OR, those with a different name by a logical AND. These criteria select entries whose value for STATE is either CA or NM and whose value for DATE is 010192.

```
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:

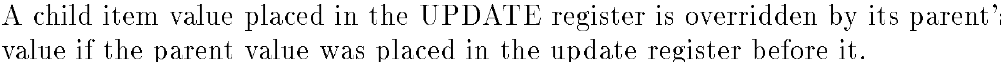
```
SET(STACK) LIST(*);
```

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 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 file requires HP floating point format. HP3000_32 specifies that the file requires IEEE floating point format. If no type is specified, 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: Any of the following fields provided that they do not conflict in meaning: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 neither HP3000_16 or HP3000_32 is specified, 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.)

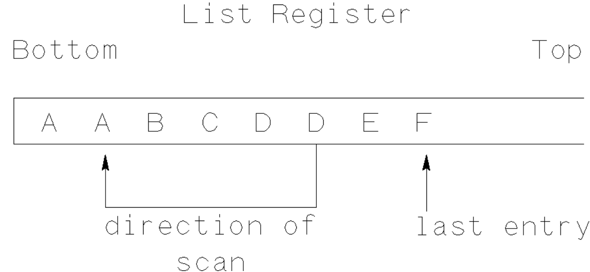
The LIST= option has a limit of 64 individually listed item names. A range limitation of 255 items for TurboIMAGE data sets and 128 items for VPLUS forms also exists.

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-name*) 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),
   PERFORMUPDATE-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;          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 ([ ]).

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 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.
EXCLA-MATION 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.
`!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
```

In addition to the switches X0-X9, there is an eleventh switch, XL, which is set automatically to OFF when code is compiled with Transact/V and to ON when code is compiled with Transact/iX. This switch can be tested 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
```

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.

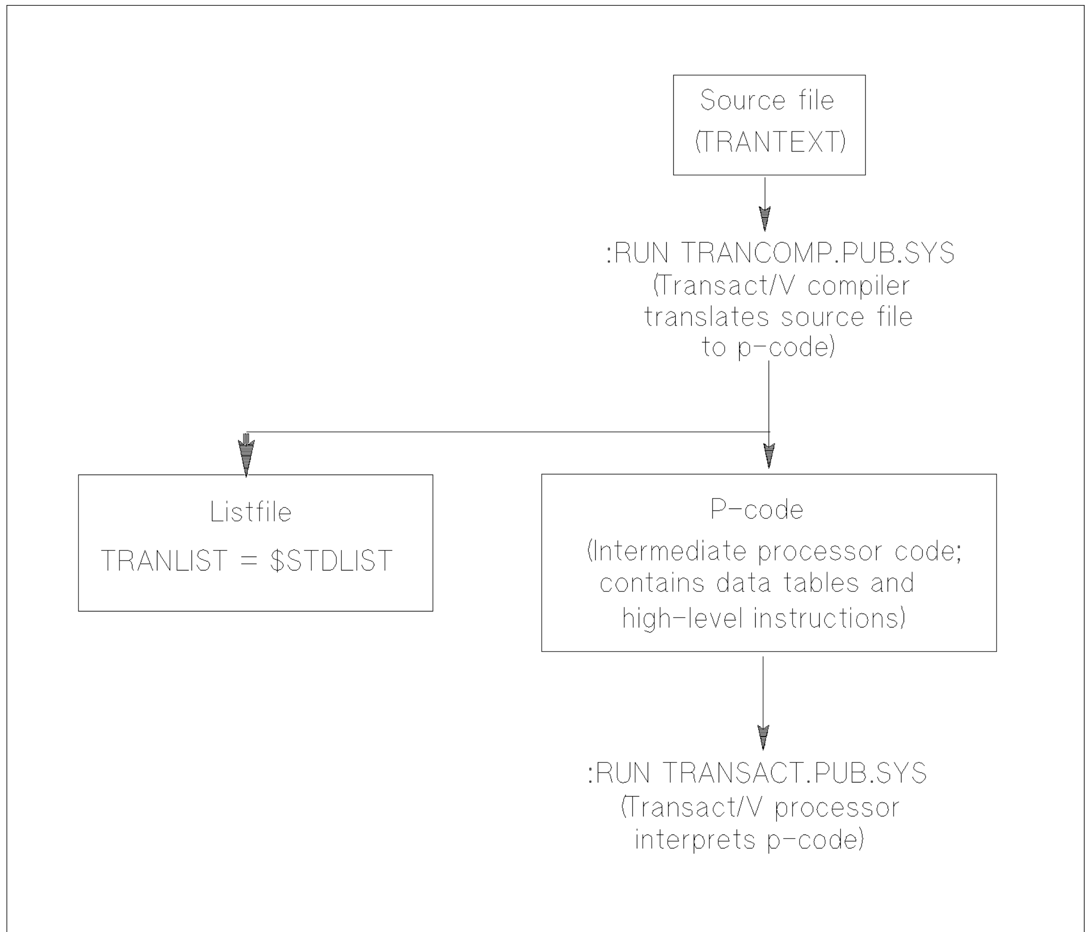


Figure 9-1. Compiling and Executing a Transact Program under MPE V

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 later 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 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.

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
```

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). If you want test mode output to be sent to another device, you can specify TRANDUMP in a file equation. This is particularly useful when you are using test mode with a program that uses VPLUS,and you do not have another terminal handy for the VPLUS forms.

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

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

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 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:

Command	Description
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.

10

Transact/V Test Facility

The Transact test facility, which is available in the MPE V and MPE/iX compatibility mode environments, lets you trace a program through execution for program debugging. To use the test facility, issue the TEST command with the following format:

```
TEST [numeric-parameter [,[segment1.]starting-instruction-address]
          [,[segment2.]ending-instruction-address]]
```

Statement Parts

numeric-parameter	Integer number that specifies the particular test mode. The specific test modes are described in Table 10-1.
segment1	Segment number where the test should begin. If none is given, the root segment (segment 0) is assumed.
starting-instruction-address	Instruction address where the trace should begin. This address is the same as the *internal-location* shown in the compiler listing produced when a Transact program is compiled with the LIST option.
segment2	Segment number where the test should stop. If none is given, segment 0 is assumed.
ending-instruction-address	Instruction address where the trace should end. As with the *starting-instruction-address*, this is the *internal-location* shown on a compiler listing.

To use the test facility, issue the TEST command with a numeric parameter at anytime when you are in command mode. The test facility stays in effect until you reissue the TEST command without a numeric parameter.

For example, if you are in command mode and want to execute all subsequent code in test mode 25, issue the following command:

```
>TEST 25
```

If you want to use test mode only between instructions 0 and 8 of the root segment, issue the following command:

```
>TEST 25,0,8
```

You terminate test mode as follows:

```
>TEST
```

If you want to use the test facility during the execution of a program, when you are NOT in command mode, you must do the following:

1. Press Ctrl Y to enter command mode.
2. Issue the appropriate TEST command.
3. Issue the RESUME command.

Execution of the program with the specified test facility continues from the point where it stopped. In response to the SYSTEM NAME> prompt, you enter the test parameters (without the key word TEST) following the program name and mode parameter.

For example, to execute in test mode 25 between instructions 0 and 8 of the root segment of program MYPROG, enter the following response to SYSTEM NAME>:

```
                                   mode parameter omitted
                                   ↓
     SYSTEM NAME> MYPROG,,25,0,8
```

TEST Output

Normally, the output from TEST is sent to the file TRANOUT (your terminal in a session or the line printer in a batch job). If you want to change the test destination, you can precede the test number parameter with a minus sign. Then the output goes to TRANDUMP (the formal file-designator for the destination of the test mode output). By default, TRANDUMP is assigned to the line printer. You can change the assignment of TRANDUMP to a different device by using a file equation.

For example, if you are executing in a command sequence, you can direct the test mode to the line printer by preceding the mode numeral designation with a minus sign:

```
>TEST -25                    <---to request test mode 25 with output to TRANDUMP
```

If you are not in a command sequence, you can accomplish the same results with the following procedure:

```
Ctrl Y                       <---to stop execution
  *CONTROL (Y) BREAK

>TEST -25                    <---to request test mode 25 with output to TRANDUMP

>RESUME                      <---enter command to resume execution
```

You could also direct the test mode output to a disk file you create specifically for that purpose. For example, to send the test output to the file TEST:

```
:BUILD TEST; REC=-80,,F,ASCII     <---create a file for test output
:FILE TRANDUMP=TEST               <---equate file TRANDUMP with file TEST
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> MYPROG,,-25          <---send test output to TRANDUMP(=TEST)
```

Another method to accomplish this is to defer test mode by setting the output priority for `TRANDUMP` to 1. For example:

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

SYSTEM NAME> MYPROG,,-25
```

After executing VTEST, you can run SPOOK5.PUB.SYS to examine the test mode information saved in a spool file.

If you use test mode for statements that access a VPLUS forms file, you should either direct the test output to a terminal other than the one where the VPLUS forms are displayed, or direct the forms to a different terminal. Otherwise, the test output will appear on the terminal screen with the forms. You could also defer test output as shown above.

For example, you can direct the test output to another terminal whose logical device number is 19, as shown:

```
:FILE TRANDUMP; DEV=19      <---direct test output to ldev 19
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> VTEST,,-34     <---run VTEST in mode 34; output to TRANDUMP
```

An alternative procedure is to direct the VPLUS forms to another terminal, while the test results are sent to your terminal. To redirect the VPLUS forms, use the TRANVPLS formal file designator:

```
:FILE TRANVPLS; DEV=19      <---direct VPLUS forms to ldev 19
:RUN TRANSACT.PUB.SYS

SYSTEM NAME> VTEST,,34      <---run your program with test mode 34
```

Now, the test mode output and character mode output appear at your terminal, but the VPLUS forms output appears on another terminal identified by its logical device number.

Table 10-1 lists the allowed test parameters and their functions.

Table 10-1. Numeric Parameters for the Test Facility

Parameter	Function
(none)	Switches off existing test mode.
1	Displays data block with information about the file or database operations only if an error occurs.
2	Displays each instruction address, the level for that instruction, and the compiler code at that address.
3	Displays each instruction address, the level for that instruction, the compiler code at that address, the space used by the list and data registers, and the amount of remaining processor work space.
4	Displays each instruction address, the level for that instruction, the compiler code at that address, the instruction timings, and the HP3000 data stack pointers Z, S, Q, and DL.
22	Displays each instruction address, the level for that instruction, the compiler code at that address, and the data block for any instructions that perform database and file operations. This parameter does not operate for the FILE verb. The data block includes the values and offsets of items in the key and argument registers used by the database or file operation.
23	Displays the instruction address, the level for that instruction, the compiler code at that address, and the data block for any instructions that perform database or file operations. The display follows a multiple record operation. This parameter does not operate for the FILE verb. The data block includes the values and offsets of items in the list, data, key, argument, match, and update registers specifically used by the database or file operation.

Table 10-1. Numeric Parameters for Test Facility (continued)

Parameter	Function
24	Displays the instruction address, the level for that instruction, the compiler code at that address, and the data block for any instructions that perform database and file operations. Does not operate for the FILE verb. The data block includes the values and offsets of items in the list, data, key, argument, match, and update registers specifically used by the database or file operation. This display is issued only when an accessed record meets the selection criteria in the match register. If there are no selection criteria for this operation or if the NOMATCH option is in effect, the display is issued for every record retrieved by the database or file operation.
25	Displays the instruction address, the level for that instruction, the compiler code at that address, and the data block for any instructions that perform database and file operations. This parameter does not operate for the FILE verb. Displays the values and offsets of items in the list, data, key, argument, match, and update registers for items specifically used by the database or file operation. This display is issued for every record accessed by the database or file operation .
34	Displays the instruction address, the level for that instruction, the compiler code for that address, and the contents of the VPLUS buffer following an instruction generated by a statement that references a VPLUS form.
42	Displays instruction address and compiler code for that address only if the instruction is not listed in the compiler listing. Displays contents of the list and data registers whenever the content of the list register (not the data register) changes.
43	Displays the instruction address, the level for that instruction, the compiler code, and the contents of the list and data register for every instruction. This parameter does not operate for the OUTPUT verb.

Table 10-1. Numeric Parameters for Test Facility (continued)

Parameter	Function
44	Displays the instruction address, the level for that instruction, the compiler code, and the contents of the list, data, key, argument, match, and update registers for every instruction.
101	Lists the data and work space recovery statistics for every command sequence.
102	Lists the data and work space recovery statistics for the entire program.
121	Issues an overlay trace whenever a program switches segments.
122	Issues a trace whenever a file is locked or unlocked.
123	Issues a work space recovery message whenever recovery is needed.

Examples

The following annotated examples show various test modes. The compiler listing shown below is for the ADD PROGRAMMER command sequence used in the examples of test modes 1, 3, and 4:

```
              starting-location
              ↓
176.000  0077    $$ADD:      <<Begin the ADD commands>>
177.000  0077    $$A:
178.000  0077      $:        <<Help for the ADD command>>
179.000  0078
180.000  0078          DISPLAY "The sub commands for ADD are: ",line=2;
181.000  0080          SET(COMMAND) COMMAND(ADD);
183.000  0082
184.000  0084      END;      <<End of help for ADD>>
185.000  0085
186.000  0085
187.000  0085      $PROGRAMMER:
188.000  0086      $PR:
190.000  0086
191.000  0086          LIST PROGRAMMER:
192.000  0087               PHONE;
193.000  0088          DATA LNAME:
194.000  0089               FNAME:
195.000  0090               PHONE;
196.000  0091          PUT PROGRAMMERS, list=(PROGRAMMER:PHONE);
197.000  0095
198.000  0095      END;      <<End of ADD PROGRAMMER>>
              ↑
              ending-location
```

In these examples, the tests are requested by the TEST command just before executing the ADD PROGRAMMER command sequence.

Test Mode 1

This test mode displays the error message only when an error occurs. In this example, a duplicate key item error occurs.

```
> TEST 1,77,95             <---Execute instructions 77 thru 95 in test mode 1

> ADD PROGRAMMER

Enter programmer's last name: MARTIN      <---duplicate name

Enter programmer's first name: JOAN

Enter phone extension number: 3803

*ERROR: DUPLICATE KEY VALUE IN MASTER  (IMAGE 43,95,PROGRAMMERS)

+-D-A-T-A---F-I-L-E---D-U-M-P----------+ <---data block for unsuccessful PUT
!
! PUT             COND: 43  STATUS: 43  RECNO: -1
! BASE: PROGB   SET: PROGRAMMERS
!
! POSN: LIST:                       DATA:
!  0     PROGRAMMER                  MARTIN          JOAN
!  30    PHONE                       3803
+--------------------------------------+

Enter programmer's last name: JONES       <---unique name; no test output

Enter programmer's first name: JAMES

Enter phone extension number: 3067
```

Test Mode 3

This test mode shows the same information as test mode 2 (the instruction address and the compiler code for every instruction). It also shows the space used by the list and data registers and the remaining processor work space.

```
> TEST 3,77,95                # of entries in list register
                              ↓      # of words in data register
> ADD PROGRAMMER              ↓      ↓
        00000  000:000   LIST  DATA  CELL  WORK
        00087  032:007      1    15    64   256
        00088  032:005      2    17    64   256

Enter programmer's last name: FRANCIS
        00089  024:008      2    17    64   256

Enter programmer's first name: JAMES
        00090  024:009      2    17    64   256

Enter phone extension number: 4835
        00091  024:005      2    17    64   256
        00092  048:129      2    17    64   256
           ↑     ↑   ↑                 ↑     ↑
           ↑     compiler code         ↑     words left in work space
           instruction location        entries left in work space
```

Test Mode 4

In addition to the instruction location and compiler code issued by test mode 2, this mode displays instruction timings and the location of the stack pointers, Z, S, Q, and DL. (See Appendix C for more information about stack pointers.)

```
> TEST 4,77,95
                                                 -----stack pointers------
> |ADD PROGRAMMER|                               ↓                  ↓
        00000   000:000   000000 000000 : Z      S      Q      D
        00087   032:007   000001 000001 : 07362 05612 05335 00092
        00088   032:005   000001 000002 : 07362 05612 05335 00092
                             ↑      ↑
                           instruction times
Enter programmer's last name: MAYOTTE
        00089   024:008   000016 000018 : 07362 05612 05335 00092

Enter programmer's first name: MARK
        00090   024:009   000015 000033 : 07362 05612 05335 00092

Enter phone extension number: 3303
        00091   024:005   000014 000047 : 07362 05612 05335 00092
        00092   048:129   000016 000063 : 09922 07735 05335 00092

> EXIT

END OF PROGRAM
```

Direct Test Output to File

The following example directs the test mode output to a file TEST. Test mode 1 is selected when the program is executed.

```
:BUILD TEST; REC=-80,,F,ASCII        <---build file for test output
:FILE TRANDUMP=TEST                  <---equate TRANDUMP to that file
:RUN TRANSACT.PUB.SYS

TRANSACT    HP32247A.00.01 - (C) Hewlett-Packard Co. 1982

SYSTEM NAME> MYPROG,,-1,77,95        <---send test mode 1 output to TRANDUMP

MYPROG  A00.00

 PASSWORD FOR PROGB>

*INFO: OPENED PROGB,3  (USER 23,-1)

> ADD PROGRAMMER                     <---command sequence at locations 77-95

Enter programmer's last name: MARTIN

Enter programmer's first name: JOAN

Enter phone extension number: 3803

*ERROR: DUPLICATE KEY VALUE IN MASTER  (IMAGE 43,95,PROGRAMMERS)

Enter programmer's last name: * CONTROL(Y) BREAK

> EXIT

END OF PROGRAM
```

To see the test mode output, run EDIT/3000 (or another text editor) and display or list the contents of TEST.

Test Modes 22 through 25

Test modes 22 through 25 are very similar. For that reason, only test mode 25 is shown. The compiler code used for the example of test mode 25 is shown below. It uses PUT, REPLACE, and DELETE to access a database. In the case of the REPLACE(CHAIN), two entries in the chain are replaced.

```
             starting location
                  ↓
453.000  0302   $PROGRAMMER:
454.000  0303   $PR:
455.000  0303       <<Replace one programmer with another>>
456.000  0303
457.000  0303       <<Set up and add entry for new name to PROGRAMMERS>>
458.000  0303
459.000  0303       LIST PROGRAMMER:
460.000  0304            PHONE;
461.000  0305       DATA LNAME ("Enter new programmer's last name"):
462.000  0307            FNAME ("Enter new programmer's first name"):
463.000  0309            PHONE;
464.000  0310       PUT PROGRAMMERS, LIST=(PROGRAMMER:PHONE);<<add new>>
465.000  0314       SET(UPDATE) LIST(PROGRAMMER);
466.000  0316       DATA LNAME ("Enter old programmer's last name"):
467.000  0318            FNAME ("Enter old programmer's first name");
468.000  0320       SET(KEY) LIST(PROGRAMMER);
469.000  0321       RESET(STACK) LIST; <<Release space>>
470.000  0322
471.000  0322       <<Update entries in PROG-AUTHOR>>
472.000  0322
473.000  0322       DISPLAY "Updating entries in PROG-AUTHOR", line=2;
474.000  0324       LIST PROG-NAME: <<Temp. storage for update>>
475.000  0325            PROGRAMMER;
476.000  0326       REPLACE(CHAIN) PROG-AUTHOR,
477.000  0326                      LIST=(PROG-NAME:PROGRAMMER);
478.000  0330       RESET(STACK) LIST; <<Release temp. storage>>
479.000  0331
480.000  0331       <<Delete old entry in PROGRAMMERS>>
481.000  0331
482.000  0331       DELETE PROGRAMMERS, LIST=();
483.000  0334
484.000  0334   END; <<End of REPLACE PROGRAMMER>>
                  ↑
             ending location
```

Test Mode 25

This test mode, like test modes 22 through 24, displays the data block (DATA FILE DUMP) for instructions that access files or data sets. As part of the data block display, test mode 25 shows the contents of all the registers used by each database or file operation. Note in the example below that the data block for REPLACE(CHAIN) is issued every time an entry is selected in the chain of the detail set PROG-AUTHOR.

```
> TEST 25,302,334

> REPLACE PROGRAMMER
        00303  032:007
        00304  032:005
        00305  040:008

Enter new programmer's last name: KING
        00307  040:009

Enter new programmer's first name: WENDY
        00309  024:005

Enter phone extension number: 3818
        00310  048:129
+-D-A-T-A---F-I-L-E---D-U-M-P----------+         <---data block for PUT
!
! PUT               COND: 0  STATUS: 0  RECNO: 21
! BASE: PROGB   SET: PROGRAMMERS
!                                                 <---contents of list and
! POSN: LIST:                      DATA:               data registers
!  0     PROGRAMMER                 KING            WENDY
!  30    PHONE                      3818
+--------------------------------------+
        00314  031:000
        00316  040:008

Enter old programmer's last name: CINTZ
        00318  040:009

Enter old programmer's first name: SIMON
        00320  198:007
        00321  208:254
        00322  081:028
        00323  080:000

Updating entries in PROG-AUTHOR
        00324  032:006
        00325  032:007
        00326  067:132
```

```
+-D-A-T-A---F-I-L-E---D-U-M-P----------+         <---data block for 1st REPLACE
!
!
! REPLACE(CHAIN)    COND: 0  STATUS: 0  RECNO: 1
! BASE: PROGB   SET: PROG-AUTHOR
!   KEY: PROGRAMMER   ARGUMENT: CINTZ          SIMON  <--key/argument regs
!
!       UPDATE:                     VALUE:
!        PROGRAMMER                   KING            WENDY <--update value
!
! POSN: LIST:                        DATA:        <--list/argument regs
!  0     PROG-NAME                    PROG1A
!  8     PROGRAMMER                   CINTZ           SIMON
+---------------------------------------+

+-D-A-T-A---F-I-L-E---D-U-M-P----------+         <---data block for 2nd REPLACE
!
! REPLACE(CHAIN)    COND: 0  STATUS: 0  RECNO: 2
! BASE: PROGB   SET: PROG-AUTHOR
!   KEY: PROGRAMMER   ARGUMENT: CINTZ          SIMON
!
!       UPDATE:                     VALUE:
!        PROGRAMMER                   KING            WENDY
!
! POSN: LIST:                        DATA:
!  0     PROG-NAME                    PROG2B
!  8     PROGRAMMER                   CINTZ           SIMON
+---------------------------------------+

2 RECORDS REPLACED
        00330  208:254
        00331  068:129

+-D-A-T-A---F-I-L-E---D-U-M-P----------+         <---data block for DELETE
!
! DELETE            COND: 0 STATUS: 0 RECNO: 14
! BASE: PROGB   SET: PROGRAMMERS
!   KEY: PROGRAMMER   ARGUMENT: CINTZ          SIMON
!
! POSN: LIST:                        DATA:
+---------------------------------------+
        00334  000:000

> EXIT
```

Note that this test mode displays only that part of the list and data registers included in a LIST= option of the data management statement.

Test Mode 34

This test mode is used to trace instructions that access VPLUS forms. The output from test mode 34 should always be sent to an alternate device rather than your terminal. Otherwise, the output interferes with the forms displayed on the screen.

The compiler code used for this example is shown below.

```
                starting location
                ↓
   98.000  0035     ADD-CUSTOMER:
   99.000  0035
  100.000  0035         GET(FORM) ADDFORM,
  101.000  0035            INIT,
  102.000  0035            LIST=(ACCOUNT:DATE),
  103.000  0035            WINDOW=("Please enter a new customer"),
  104.000  0035            F7=START-OF-PROGRAM,
  105.000  0035            F8=END-OF-PROGRAM,
  106.000  0035            AITPREAD;
  107.000  0051
  108.000  0051     PUT-CUSTOMER:
  109.000  0051
  110.000  0051         SET(KEY) LIST(ACCOUNT);    <<Set up key register>>
  111.000  0052         FIND CUSTOMER, LIST=();    <<Check if customer exists>>
  112.000  0055
  113.000  0055         IF STATUS <> 0 THEN        <<Customer already in base>>
  114.000  0055 1         GO TO ADD-CUST-ERROR;
  115.000  0058
  116.000  0058         PUT CUSTOMER,
  117.000  0058            LIST=(ACCOUNT:DATE),
  118.000  0058            ERROR=PUT-ERROR(*);     <<Process PUT verb error>>
  119.000  0064
                ↑
                ending location
```

Before running the program VTEST in test mode -34, build a file named TEST to receive the test data and then equate TRANDUMP to that file:

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

SYSTEM NAME> VTEST,,-34,35,64                <---run VTEST in test mode 34 with
                                                 test output sent to file TEST
```

The test output from file TEST looks like this:

```
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+  \
!                                          \
! PUT(FORM)       CODE: 0     FKEY: 0       \
! FORM: MENU                FILE: CUSTF  TF  \
!                                             \
+--------------------------------------+       <--from previous form access
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+      /      statements
!                                            /
! UPDATE(FORM)    CODE: 0     FKEY: 1       /
! FORM: MENU                FILE: CUSTF    /
!                                         /
+--------------------------------------+
        00035  160:131
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+      <--output from location 35
!
! GET(FORM)       CODE: 0     FKEY: 0         <--last key pressed is ENTER
! FORM: ADDFORM          FILE: CUSTFORM
!
! OFFSET: LIST:                 DATA:
!  0       ACCOUNT               1113434343        \
!  10      FIRST-NAME            MARGARET           \
!  28      INITIAL               S                   \
!  29      LAST-NAME             TRUEMAN              \
!  49      STREET-ADDR           524 East 79th Street  <--entered data
!  71      CITY                  New York             /
!  85      STATE                 NY                  /
!  87      ZIP                   10024              /
!  96      DATE                  07/21/82          /
!
+--------------------------------------+
        00051  198:000
        00052  065:137
        00055  011:001
        00058  048:137
        00064  004:000
        00035  160:131
```

```
+-V-P-L-U-S---B-U-F-F-E-R---D-U-M-P----+        <--back to location 35
!
! GET(FORM)       CODE: 0     FKEY: 8           <--f8 pressed to exit
! FORM: ADDFORM            FILE: CUSTFORM
!
! OFFSET: LIST:                   DATA:
!  0       ACCOUNT                 ...w...;b.              \
!  10      FIRST-NAME              .W...$...,...B,H..       \
!  28      INITIAL                 S                         \
!  29      LAST-NAME               TRU..AN......  ...Z.       \
!  49      STREET-ADDR             Y..>..5"......,.X.?...      <--garbage
!  71      CITY                    .......3......             /
!  85      STATE                   >.                        /
!  87      ZIP                     .........                /
!  96      DATE                    ..2....                 /
!
+--------------------------------------+
```

Test Mode 42

This test mode lists the contents of the list and data registers only when the list register is changed.

The compiler listing shown below is for two subcommands that are part of a LIST command sequence; this code is executed by entering LIST PROGRAMMER and LIST PROGRAM respectively.

```
                starting location
                ↓
 97.000  0019       $PROGRAMMER:
 98.000  0020       $PR:
 99.000  0020           <<list programmers>>
100.000  0020
101.000  0020           LIST PROGRAMMER:
102.000  0021                PHONE;
103.000  0022           OUTPUT(SERIAL) PROGRAMMERS,
104.000  0022                          LIST=(PROGRAMMER:PHONE),
105.000  0022                          SORT=(PROGRAMMER),
106.000  0022                          NO COUNT;
107.000  0028
108.000  0028       END; <<end of LIST PROGRAMMER>>
109.000  0029
110.000  0029
111.000  0029       $PROGRAM:
112.000  0030       $P:
113.000  0030           <<list programs>>
114.000  0030
115.000  0030           LIST PROG-NAME:
116.000  0031                DESCRIPTION;
117.000  0032           OUTPUT(SERIAL) PROGRAMS,
118.000  0032                          LIST=(PROG-NAME:DESCRIPTION),
119.000  0032                          SORT=(PROG-NAME),
120.000  0032                          NO COUNT;
121.000  0038
122.000  0038       END; <<end of LIST PROGRAM>>
                ↑
                ending location
```

Note that in the following test output, the current contents of the data register are never shown. Only the previous contents are shown. Thus, the data register display in the test output from LIST PROGRAMMER contains garbage. Similarly, the data register display in the test output from LIST PROGRAM contains data from the previous command sequence.

```
> TEST 42,19,38

> LIST PROGRAMMER
        00020  032:007
+-L-I-S-T---D-U-M-P--------------------+         <--issued for LIST PROGRAMMER
! POSN: LIST:                        DATA:
!  0     PROGRAMMER                   .a.B.a..b.............B.H....
+--------------------------------------+
        00021  032:005
+-L-I-S-T---D-U-M-P--------------------+         <--issued for LIST PHONE
! POSN: LIST:                        DATA:
!  0     PROGRAMMER                   .a.B.a..b.............B.H....
!  30    PHONE                        ....
+--------------------------------------+
        00022  066:129

Programmer                    Phone Number
 CRESSMAN       PETE           3805
 ERCOLANI       JOE            4343
 KING           WENDY          3818
 LEDERMAN       ABE            3753
 VANN           KEITH          4046
        00028  000:000

> LIST PROGRAM
        00030  032:006
+-L-I-S-T---D-U-M-P--------------------+         <--issued for LIST PROG-NAME
! POSN: LIST:                        DATA:
!  0     PROG-NAME                    VANN
+--------------------------------------+
        00031  032:002
+-L-I-S-T---D-U-M-P--------------------+         <--issued for LIST DESCRIPTION
! POSN: LIST:                        DATA:
!  0     PROG-NAME                    VANN
!  8     DESCRIPTION                          KEITH         4046.........6.B..
!                                     ........"........X.?
+--------------------------------------+
        00032  066:130
```

```
Program Name  Program Description
  CRUNCH        Compacts ASCII files.
  DISCOPY       Copies disc files.
  GTDATA        Generates random test data.
  PROJMAN       Project management using the critical path method.
  SGEN          Generates STREAM job files.
  TLIST         Lists the contents of a "STORE" tape.
  UNCRUNCH      Expands a file compacted by CRUNCH.
         00038  000:000

> EXIT
```

Test Modes 101 and 102

These test modes allow you to keep track of the list and data register size, and whether work-space recovery was needed. Test mode 101 displays test data at the end of every command sequence; test mode 102 displays test data only at the end of the program.

```
> TEST 101                          <---request test mode 101

> ADD PROGRAMMER                    <---start of command sequence

Enter programmer's last name: MARTIN

Enter programmer's first name: JOAN

Enter phone extension number: 3803

+-S-E-Q-U-E-N-C-E---D-U-M-P--+      <---current status of list/data regs
!                                       at end of this command sequence
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 17 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+

> ADD PROGRAM                       <---new command sequence

Enter program name: MYPROG

Program description: Test program for Manual
```

```
+-S-E-Q-U-E-N-C-E---D-U-M-P--+       <---status at end of second
!                                        command sequence
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 34 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+

> TEST 102                           <---request test mode 102

> LIST PROGRAMMER

Programmer                    Phone Number
 CRESSMAN       PETE           3805
 ERCOLANI       JOE            4343
 KING           WENDY          3818
 LEDERMAN       ABE            3753
 MARTIN         JOAN           3803
 VANN           KEITH          4046

> LIST PROGRAM

Program Name     Program Description
 CRUNCH           Compacts ASCII files.
 DISCOPY          Copies disc files.
 GTDATA           Generates random test data.
 MYPROG           Test program for Manual
 PROJMAN          Project management using the critical path method.
 SGEN             Generates STREAM job files.
 TLIST            Lists the contents of a "STORE" tape.
 UNCRUNCH         Expands a file compacted by CRUNCH.

> EXIT

+-R-U-N---D-U-M-P------------+       <---test output only issued
!                                        at end of program
!   MAXIMUM LIST= 2 ITEMS
!   MAXIMUM DATA= 34 WORDS
!   WORKSPACE RECOVERY= 0
!
+----------------------------+
```

11

Transact/iX Symbolic Debugger: TRANDEBUG

Overview

TRANDEBUG is an interactive tool that helps you debug native mode Transact/iX applications. You use TRANDEBUG to isolate run-time errors after your program has compiled and linked successfully.

TRANDEBUG allows you to:

- Stop the execution of a program at specific locations
- Display the contents of:
  - Transact data items
  - Transact registers
  - Run-time options
  - Call stack
  - Perform stack
  - VPLUS communication area
- Set breakpoints at specific locations, data items, and data item values
- Modify the contents of Transact data items, registers and run-time options
- Trace the execution path of a program
- View the source code for the program being debugged
- Record and playback TRANDEBUG commands using an MPE/iX file
- Issue MPE/iX commands
- Invoke the MPE/iX Native Mode Debug Facility, NMDEBUG
- Trace KSAM, MPE/iX file, and TurboIMAGE intrinsic calls.

Features and Benefits

TRANDEBUG offers the following major features and benefits.

Symbolic Debugger

TRANDEBUG is a symbolic debugging interface for Transact/iX. You can interactively monitor your program execution and location, and examine the contents of Transact data items and registers without knowing the memory addresses.

Breakpoints

TRANDEBUG lets you stop your program at specific points of interest called breakpoints. Set breakpoints by using the following four commands:

- BREAK SET command sets a breakpoint at a specified segment number and p-code offset.

- DATA BREAK SET command sets a breakpoint at a specified data item or data-item value.

- DATA BREAK REGISTER command sets a breakpoint at a specified register.

- LABEL BREAK SET command sets a breakpoint at a specified label.

Once breakpoints are set, use the following commands to list and delete breakpoints:

- BREAK LIST
- BREAK DELETE
- DATA BREAK LIST
- DATA BREAK DELETE

You'll find syntax, descriptions, and examples of these commands in the "TRANDEBUG Commands" section of this chapter.

Transact Display Functions

TRANDEBUG lets you display information about your Transact/iX program with the numerous options of the DISPLAY command. You can display the contents of Transact data items, registers, run-time options, database or file information, call and perform stacks, and VPLUS communication areas. The data and list registers are displayed by the DISPLAY ITEM command. Other registers that can be displayed are the match, key, update, status, statusin, argument, and input run-time registers.

Transact Modify Functions

TRANDEBUG lets you modify values of Transact data items, registers and run-time options. The MODIFY ITEM command allows you to update the contents of a data register. You can modify the input, data, key, argument, match, status, and update registers.

Program Execution Control

You can control your program execution when you use TRANDEBUG. The CONTINUE command resumes program execution. When breakpoints are encountered, TRANDEBUG suspends program execution so that you can examine or manipulate data. You can also use the STEP command to execute a program on a statement-by-statement basis.

MPE/iX Subsystem Support

TRANDEBUG lets you take advantage of the MPE/iX native mode environment. Statements that begin with a colon (:) are passed automatically to the MPE/iX Command Interpreter for execution. TRANDEBUG allows you to invoke the native mode debugger (NMDEBUG.PUB.SYS) when you want to monitor your program's execution at the machine-instruction level.

Source Code Window

TRANDEBUG includes a window through which you can view source code for the program being debugged. This window can be turned on and off at any time during a TRANDEBUG session. Use the WINDOW ON command to display the source file. You can change the window size, page forward or backward through the source file, or jump to a specified statement or label.

TRANDEBUG Log and Command Files

With TRANDEBUG, you can log your interactive debug commands to MPE/iX files. You can use this feature to create a procedure for establishing frequently used breakpoints in a program under development. The LOG ON command starts a recording session where your subsequent keystrokes are recorded to a log file. The LOG OFF command ends the recording session, and LOG CLOSE saves the recording to a permanent file. You then invoke the commands in the log file with the USE command. You can also use a text editor to create command files that can be invoked with the USE command.

Arithmetic Trapping

TRANDEBUG provides a mechanism to gain control of your program when arithmetic traps occur. For example, an integer overflow or division by zero sets an arithmetic trap. When this happens, an error message is displayed and control returns to TRANDEBUG so you can determine what caused the error.

Control Y Trapping

When (Ctrl) Y is pressed during a debugging session, program control returns to TRANDEBUG, and at this point, you can enter any valid TRANDEBUG command. This allows you to regain control of TRANDEBUG before the next breakpoint is reached.

Online Help

The HELP subsystem provides a way to obtain information on command syntax, parameter descriptions, and examples. Any time you need help in executing a command, type `HELP` and (optionally) the command name. Commands are also described in the "TRANDEBUG Commands" section later in this chapter.

Compatibility

TRANDEBUG provides debugging support exclusively for Transact/iX applications. Transact/iX programs compiled from TRANCOMP/V generated p-code files are not supported. You must compile your program from a source file in order to use TRANDEBUG.

TRANDEBUG does not support other programming languages, such as COBOL, FORTRAN, or Pascal. The native mode debugger can be invoked from within TRANDEBUG for this purpose.

Using TRANDEBUG

This section provides a tutorial to introduce you to the frequently-used TRANDEBUG commands. It will help you become familiar with using TRANDEBUG. A complete list of TRANDEBUG commands and examples is found in the "TRANDEBUG Commands" section later in this chapter.

After reading this section, you will know how to perform the following tasks:

- Compile a program with the TRANDEBUG option
- Start and end a TRANDEBUG session
- View source code in TRANDEBUG
- Set breakpoints
- Continue program execution from within TRANDEBUG
- Display data items
- Modify data items
- Step through a program.

Compiling with the TRANDEBUG Option

To use TRANDEBUG, you first compile your Transact/iX program(s) with the TRANDEBUG option. Specify this option in addition to any other compiler options that your application requires.

The following example shows how to compile and link a main program using the TRANDEBUG option:

```
:TRANXLLK MYSOURCE,MYPROG,MYLIST;INFO="TRANDEBUG,NOLIST"
```

The compiled output appears as follows:

```
PAGE 1 Transact/iX HP30138A.04.02 © Copyright HEWLETT-PACKARD CO. 1987
MON, OCT 26, 1992, 9:42 AM

COMPILING WITH OPTIONS: CODE,DICT,ERRS,TRANDEBUG


NUMBER OF ERRORS =   0          NUMBER OF WARNINGS =   0
PROCESSOR TIME  0:00:02.1       ELAPSED TIME  0:00:03


END OF COMPILE
HP Link Editor/XL (HP30315A.00.25) Copyright Hewlett-Packard Co. 1986

LinkEd> link from=$oldpass;to=myprog

END OF LINK
:
```

If your Transact application includes Transact subprograms, each subprogram that requires debugging must be compiled with the TRANDEBUG option.

Starting and Ending TRANDEBUG Sessions

After you have successfully compiled and linked your program using the TRANDEBUG option, run your program to start a TRANDEBUG session. The following figure shows how TRANDEBUG identifies itself on your screen:

```
:RUN MYPROG
```

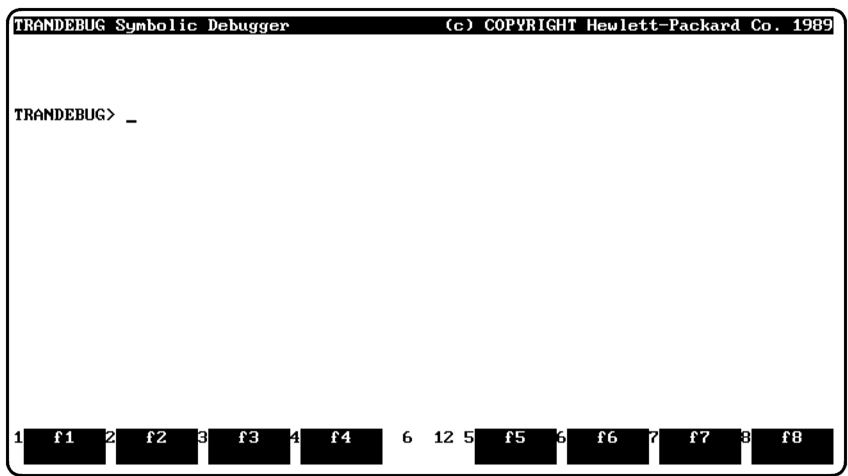


Figure 11-1. The TRANDEBUG Screen

To terminate a TRANDEBUG session, type the following at the TRANDEBUG> prompt:

```
TRANDEBUG> ABORT
```

This command forces your program to abort TRANDEBUG, then exit. After you complete your debugging session, you must recompile your program without the TRANDEBUG compiler option. Be careful not to move program files compiled with TRANDEBUG to production.

Displaying Source Code in TRANDEBUG

TRANDEBUG provides a source code window that lets you view statements in the source code as they execute. This window can be controlled with the WINDOW and PAGE commands. Your program can run with the windows turned on or off. You can change the size of the window and scroll forward and backward through the source code listing independent of the program's execution.

To display the source code for the program being debugged, type WINDOW ON at the TRANDEBUG> prompt as follows:

```
TRANDEBUG> WINDOW ON
```

TRANDEBUG immediately displays the source code for the program being debugged in a source code window like this:

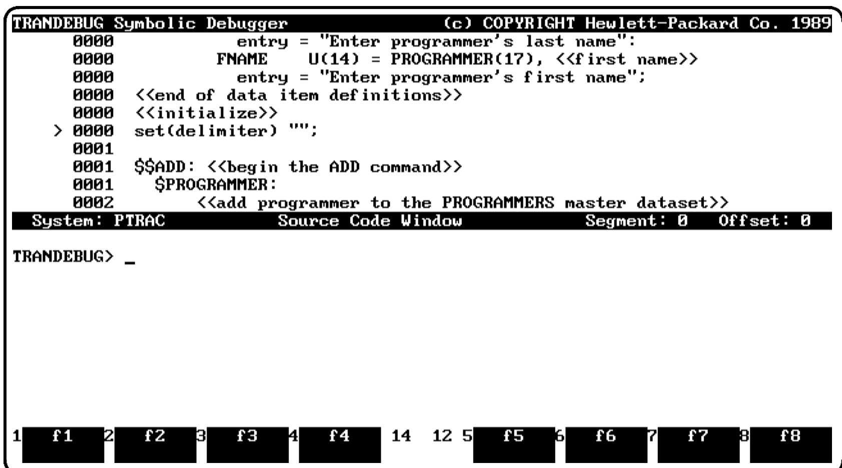


Figure 11-2. TRANDEBUG Source Code Window

In the example shown, the right arrow (">") marks the *current location*. It points to the next statement that will be executed. In this example, the current location marker points to the first executable program statement. The *current system*, *segment*, and *offset* are displayed within the source code window.

Setting a Breakpoint

When you want to stop at a specific location in your program, you can set a breakpoint. With TRANDEBUG, you set a breakpoint by specifying the segment number and p-code offset within a Transact/iX program. If a breakpoint is to be set in the current segment, just specify the p-code offset.

Figure 11-3 shows a compiled listing for a program that adds programmers to a TurboIMAGE database. This program is used in the remaining examples in this section.

```
   line number
   ↓      p-code offset
 1.000   ↓        SYSTEM PTRAC,
 2.000   0000            BASE = PROGB ( ,3),
 3.000   0000            SOGMPM = "PTRAC   A00.00";
 4.000   0000
 5.000   0000     <<PTRAC is a program that adds programmers to    >>
 6.000   0000     <<a TurboIMAGE database called PROGB.            >>
 7.000   0000     <<PROGB is opened with exclusive modify access.  >>
 8.000   0000
 9.000   0000     DEFINE(ITEM)
10.000   0000
11.000   0000         PHONE        U(4), <<programmer's phone extension #>>
12.000   0000           HEAD = "Phone Number",
13.000   0000           ENTRY = "Enter phone extension number":
14.000   0000
15.000   0000         PROGRAMMER   U(30),   <<programmer's name>>
16.000   0000           HEAD = "Programmer":
17.000   0000             LNAME    U(16) = PROGRAMMER(1),  <<last name>>
18.000   0000               ENTRY = "Enter programmer's last name":
19.000   0000             FNAME    U(14) = PROGRAMMER(17), <<first name>>
20.000   0000               ENTRY = "Enter programmer's first name";
21.000   0000     <<end of data item definitions>>
22.000   0000     <<initialize>>
23.000   0000     SET(DELIMITER) "";
24.000   0001
25.000   0001     $$ADD: <<begin the ADD command>>
26.000   0001       $PROGRAMMER:
27.000   0002          <<add programmer to PROGRAMMERS master data set>>
28.000   0002
29.000   0002           LIST PROGRAMMER:
30.000   0003                PHONE;
31.000   0004           DATA LNAME:
32.000   0005                FNAME:
33.000   0006                PHONE;
34.000   0007           PUT PROGRAMMERS, LIST=(PROGRAMMER:PHONE);
35.000   0011       END; <<end of ADD PROGRAMMER>>
36.000   0012
37.000   0012     END PTRAC;
```

Figure 11-3. Sample Transact Program

In Figure 11-3, only the p-code offset is required to set a breakpoint. A segment number is not required since only the current segment is being debugged.

To specify a breakpoint at p-code offset 7, type BREAK SET at the TRANDEBUG> prompt:

```
TRANDEBUG> BREAK SET 7
```

After the breakpoint is set, TRANDEBUG displays the following:

```
          BREAKPOINT SET:
           SYSTEM      SEGMENT      OFFSET     COUNT       COMMAND LIST
           ------------------------------------------------------------
        0. PTRAC             0           7         1
```

When the breakpoint is reached, TRANDEBUG suspends program execution and displays the TRANDEBUG> prompt.

Continuing Program Execution

The CONTINUE command resumes the execution of the program being debugged. Once you have set up initial breakpoints during a debugging session, you can resume program execution by typing CONTINUE at the TRANDEBUG> prompt. TRANDEBUG continues execution of your program until a breakpoint is encountered or the program terminates.

The following example uses the CONTINUE command for the sample program shown in Figure 11-3:

```
TRANDEBUG> CONTINUE
>ADD PROGRAMMER
Enter programmer's last name>  LORENZ
Enter programmer's first name> JAMES
Enter phone extension number> 5000
            BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

               SYSTEM          SEGMENT      OFFSET
               -----------------------------------
               PTRAC                 0           7
TRANDEBUG>
```

Normal program execution resumes after you issue the CONTINUE command. The program continues until a breakpoint is reached when TRANDEBUG will suspend program execution and display the TRANDEBUG> prompt.

When Ctrl Y is invoked during a debugging session, program control returns to TRANDEBUG and the TRANDEBUG> prompt. You can then set additional breakpoints and type CONTINUE to resume execution of your program, or issue any other valid TRANDEBUG commands.

The following is an example of how Ctrl Y is used:

```
TRANDEBUG> CONTINUE
>ADD PROGRAMMER
Enter programmer's last name: Ctrl Y
TRANDEBUG> BREAK SET 7
TRANDEBUG> CONTINUE
```

Displaying the Values of Data Items

To display the value of a data item, you issue the DISPLAY ITEM command at the TRANDEBUG> prompt. When you use this command, you can display the contents of all items in the list and data registers, or specific items.

The following example displays all items in the list and data registers:

```
TRANDEBUG> DISPLAY ITEM

 LIST REGISTER:
PROGRAMMER          :   LORENZ     JAMES
PHONE               :   5000
```

The following example displays a specific data item:

```
TRANDEBUG> DISPLAY ITEM PROGRAMMER

PROGRAMMER          :   LORENZ     JAMES
```

Modifying the Values of Data Items

To modify the value of a data item, type MODIFY ITEM followed by the data item name at the TRANDEBUG> prompt. You can modify the contents of the data register, but not the list register. After you issue the MODIFY ITEM command, TRANDEBUG displays the specified data item value and prompts you for the new value. Pressing (Return) without entering a new value leaves the data item unchanged.

The following example modifies the FNAME child data item:

```
TRANDEBUG> MODIFY ITEM FNAME
FNAME             :  < JAMES         >      := < JIM          >

TRANDEBUG> DISPLAY ITEM PROGRAMMER

PROGRAMMER:         :   LORENZ     JIM
TRANDEBUG>
```

Stepping Through a Program

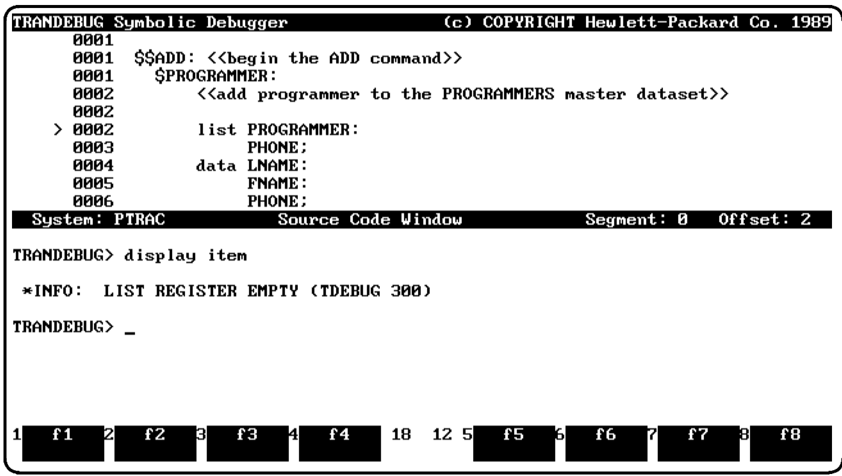
The STEP command causes one or more groups of instructions to be executed. It can be used when you want to examine the contents of variables both before and after the groups of instructions are executed.

The Transact/iX compiler determines the level of granularity at which you can single-step through a program. TRANDEBUG allows you to single-step through programs at a level that is meaningful to the Transact/iX language.

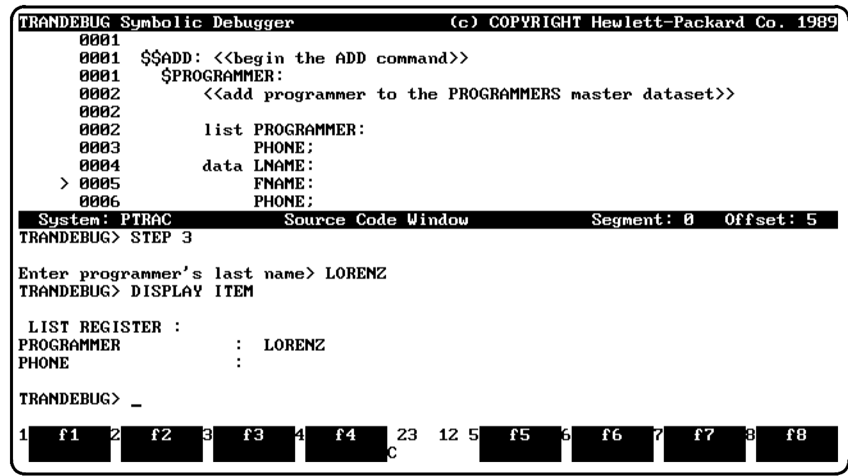
When single-stepping through the LIST and DATA statements, TRANDEBUG returns control to you *after each data item* is loaded into the list and data registers.

When single-stepping through other Transact/iX verbs, TRANDEBUG returns control to you *after each statement* is executed.

The following example uses the STEP command to control program execution. The current location marker points to the LIST PROGRAMMER statement (p-code offset = 2). The user types **DISPLAY ITEM** at the TRANDEBUG> prompt as shown below:



Program execution is resumed briefly with the **STEP 3** command. The name “LORENZ” is entered in response to a program prompt. Control returns to TRANDEBUG after the 3 program instructions are executed. The user then types the **DISPLAY ITEM** command at the TRANDEBUG> prompt:



TRANDEBUG Startup Initialization File

When TRANDEBUG begins to execute a program, it searches for a startup file before returning control to you. This startup file must be named "TDBGINIT" and reside in the same group as the program file. The file must be in ASCII format with each line representing a command line for TRANDEBUG.

When all the commands in the startup TDBGINIT file have executed, control returns to you. This startup file can be used if a series of commands must be executed each time the debugger is started. The file can be created using the LOG command, which is discussed in "Command Descriptions."

Redirecting VPLUS Input and Output

Before you use TRANDEBUG to debug a VPLUS application, you should redirect your VPLUS input and output to an alternative terminal. To do this, follow these steps:

1. Find an alternative terminal that is available and is hard-wired to your computer.
2. Log onto the alternative terminal.
3. Type SHOWME to determine the logical device of the alternative terminal.
4. Write down the logical device number of the alternative terminal.
5. Log off the alternative terminal and go back to your terminal.
6. Type the following file statement to redirect your application forms to the alternative terminal and then run your program:

```
:FILE TRANVPLS; DEV=##   (Use the Ldev # from step 4.)
:RUN MYPROG
```

If the redirection fails, do the following steps:

1. Execute the following command to determine if the alternative terminal is available **(AVAIL)**:

```
:SHOWDEV ##  (## is the LDEV of the alternative terminal)
```

2. Determine the system baud rate by asking your system manager. Then, check the alternative terminal baud rate to ensure that it is the same as the system baud rate.

Disabling the Debugger

Under some circumstances, it might be desirable to turn off TRANDEBUG and not have to recompile the program without the TRANDEBUG compiler option. The TRANDEBUG debugger examines the TRANDEBUG MPE/iX system variable and disables debugging if this variable is set.

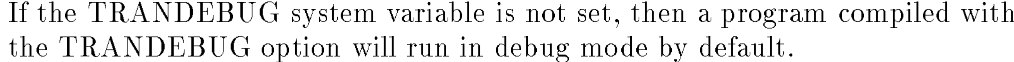
To disable TRANDEBUG for your session, type the following command before running your program:

```
:SETVAR TRANDEBUG,"OFF"
```

You can then turn on the debugger by typing the following command:

```
:SETVAR TRANDEBUG,"ON"
```

Note



If the TRANDEBUG system variable is not set, then a program compiled with the TRANDEBUG option will run in debug mode by default.

Alternative Debug Entry Points

Sometimes, you may have a large complex application composed of many Transact systems that are under development. When performing integration testing, all systems compiled with TRANDEBUG might be enabled at the same time, but you only want to run one of these systems in debug mode. You can do this by using the TRANDEBUG system variable.

To begin your debugging session at a Transact/iX system other than the main program, use the TRANDEBUG system variable as follows:

:SETVAR TRANDEBUG,"*startsystem*"

This allows your program to execute without invoking TRANDEBUG until *startsystem* begin to execute.

TRANDEBUG Run-Time Environment

By default, TRANDEBUG is invoked automatically after you compile your program with the TRANDEBUG compiler option. Unless you use the SETVAR TRANDEBUG system command to specify which system you want to debug (see “Alternative Debug Entry Points”), TRANDEBUG is invoked for the main program.

For TRANDEBUG to be invoked for a particular system, the following two conditions must be met:

- The system must be compiled with the TRANDEBUG compiler option.
- The list file IU*xxxxxx* (where *xxxxxx* is the system name) generated by the Transact/iX compiler, must be accessible.

Note

For versions prior to Transact/iX A.04.02, the list file format is IT*xxxxxx* (where *xxxxxx* is the system name).

The Transact/iX compiler automatically creates the IU*xxxxxx* list file for each system in your current group that is compiled with TRANDEBUG. This file contains the symbol table and source code information needed during debugging. These files are purged when you recompile your program with the debug option turned off.

Arithmetic Traps

TRANDEBUG provides a mechanism to gain control of your program when arithmetic traps occur. When an arithmetic trap occurs, the error message is displayed and control returns to TRANDEBUG so that you can display data-item values and determine what caused the error.

Once you resume execution from TRANDEBUG, the same program flow takes place as before. If you are running a command-driven program, control is transferred to the TRANDEBUG> prompt; if you are not running a command driven program, the EXIT/RESTART? prompt appears.

TRANDEBUG Commands

This section describes, in detail, each TRANDEBUG command and gives the syntax for the command and an example. The commands are presented in alphabetic order.

:

Allows access to the MPE/iX Command Interpreter.

Syntax

```
:[COMMAND]
```

Parameters

command The MPE/iX Command Interpreter command that will be executed.

Discussion

This command allows access to the MPE/iX Command Interpreter. Any commands that can be used with the HPCICOMMAND intrinsic can be used with the : command. You can use this command to run other programs from within TRANDEBUG.

Example

The following example uses the : command:

```
TRANDEBUG> :listf prog@

FILENAME

PROG01P      PROG02P      PROG03P
```

ABORT

Terminates execution of the Transact/iX program and exits TRANDEBUG.

Syntax

```
ABORT
```

Parameters

None.

Discussion

This command exits TRANDEBUG and terminates the execution of the Transact/iX program.

Example

The following example uses the ABORT command to terminate TRANDEBUG:

```
TRANDEBUG> CONTINUE
Customer       Dollars Spent
----------------------------
John Smith     $100.25
Jane Doe       $201.75
Joe Customer   $ 21.45

            BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

                SYSTEM         SEGMENT       OFFSET
                ------------------------------------
            1. PARENT                0           10

TRANDEBUG> ABORT

END OF PROGRAM
:
```

AUTORPT

Allows you to repeat the last command typed by pressing the (Return) key.

Syntax

```
{ AUTORPT } [ ON  ]
{ AR      } [ OFF ]
```

Parameters

ON	Turns on the auto-repeat flag to allow you to repeat a previously-typed command by pressing (Return).
OFF	Turns off the auto-repeat flag.

Discussion

This command allows you to repeat automatically the last command typed. First you must type the AUTORPT command followed by another TRANDEBUG command. When you press (Return), the last command typed will repeat itself.

This is useful if single-stepping is desired through a lengthy program, or if a breakpoint is set and then subsequently reached many times. In such cases, you would first type the STEP or CONTINUE command, then use the (Return) key for repeated executions. If neither ON nor OFF is specified, the current value of the AUTORPT flag is displayed. It can then be changed. The default for the AUTORPT flag is OFF.

Example

The following example uses the AUTORPT command to step automatically through a program.

```
TRANDEBUG> AUTORPT ON
TRANDEBUG> STEP

             EXECUTION STOPPED:

                SYSTEM           SEGMENT        OFFSET
                ------------------------------------
                PARENT                 0            28

TRANDEBUG> (Return)
           REPEATING: STEP

             EXECUTION STOPPED:

                SYSTEM           SEGMENT        OFFSET
                ------------------------------------
                PARENT                 0            34

TRANDEBUG> (Return)
```

```
          REPEATING: STEP

            EXECUTION STOPPED:

               SYSTEM          SEGMENT       OFFSET
               ------------------------------------
               PARENT                0           36

TRANDEBUG> AUTORPT OFF
TRANDEBUG> [Return]
```

BREAK DELETE

Deletes a specified breakpoint.

Syntax

```
{ BREAK DELETE }  [ #breakpoint_number                           ]
{ BD           }  [ p-code_offset [ ,segment [ ,system ] ]       ]
                  [ ALL                                          ]
```

Parameters

#break-point_number	The number assigned to the breakpoint to be deleted. You can identify this number by listing the breakpoints that are set. You must specify "#" to indicate that you want to delete the breakpoint by number rather than by p-code offset and segment. For example, to delete the third breakpoint, you would use #3.
p-code_offset	The p-code offset corresponding to the breakpoint to be deleted.
segment	The segment number that corresponds to the breakpoint to be deleted. The default for this parameter is 0.
system	A string representing the name of the system in which you want to delete the breakpoint. The currently-executing system is the default.
ALL	A keyword that deletes all set breakpoints.

Discussion

This command allows you to delete breakpoints set at Transact/iX statements. If you set breakpoints from within NMDEBUG, you must also delete these breakpoints from within NMDEBUG. The BREAK DELETE command only knows about breakpoints that were set with a corresponding BREAK SET command.

To execute this command you must do one of the following:

- Specify the number of the desired breakpoint.
- Specify the segment and p-code offset of the breakpoint you want to delete.

You can also use the BREAK DELETE command without any parameters by specifying BREAK DELETE or BD and pressing (Return). TRANDEBUG displays each breakpoint and prompts you to either keep it or delete it.

Examples

The following examples show two methods of deleting breakpoints. In the first example, a BREAK LIST command is used to determine the number corresponding to the desired breakpoint. In the second example, the breakpoint is deleted based on the p-code offset and segment number.

```
TRANDEBUG> bl

    CURRENT BREAKPOINTS:

        SYSTEM  SEGMENT    OFFSET   COUNT
        ---------------------------------
   0. TEST          0        24       1
            CMD LIST>>>>> DISPLAY MATCH;STEP
   1. TEST          1        28       1
   2. TEST1         1        10       1
            CMD LIST>>>>> DISPLAY KEY;

TRANDEBUG> break delete #1

  BREAKPOINT DELETED:

        SYSTEM  SEGMENT   OFFSET   COUNT
        --------------------------------
   1. TEST1         1       28       1

TRANDEBUG> bd 24

  BREAKPOINT DELETED:

        SYSTEM  SEGMENT   OFFSET   COUNT
        --------------------------------
   0. TEST          0       24       1
            CMD LIST>>>>> DISPLAY MATCH;STEP
```

BREAK LIST

Lists the breakpoints that are set in the Transact/iX program.

Syntax

```
{ BREAK LIST }
{ BL         }
```

Parameters

None.

Discussion

This command allows you to display currently-set breakpoints. The command only lists breakpoints set by the BREAK SET statement and not breakpoints set from within NMDEBUG. An asterisk listed in front of a breakpoint indicates the point where your program stopped.

Example

The following example shows the listing of a typical set of breakpoints.

```
TRANDEBUG> break list

    CURRENT BREAKPOINTS:

        SYSTEM  SEGMENT   OFFSET   COUNT
        --------------------------------
  *  0. TEST          0       24       1
            CMD LIST>>>>> DISPLAY MATCH;STEP
     1. TEST          1       28       1
     2. TEST1         1       10       1
            CMD LIST>>>>> DISPLAY KEY;
```

BREAK SET

Sets a breakpoint at the specified location.

Syntax

```
{ BREAK SET } { p-code_offset [ ,segment] [ ,system] } [ ,count] [ ,{cmdlist} ]
{ BS        } { system                               }
```

Parameters

p-code_offset The p-code offset corresponding to the Transact instruction at which you want to stop program execution. The range of valid p-code offsets is 0 to 16383. If the specified p-code offset does not match an actual p-code value, then the next smaller actual p-code will be used.

segment The segment number corresponding to the instruction location. The range of valid segments is 0 to 125. The default is the current segment.

system A string representing the name of the system in which you want to set the breakpoint. If you only specify this parameter, a breakpoint is set at the beginning of the system. Do this if you want to stop at the beginning of a child system. The default is the current system.

count A number indicating to the program to stop at the breakpoint every *n* times it is reached. For example, to stop at a breakpoint every third time the p-code offset is executed, you would specify 3 for count. The range of count is 1 to 1,000,000. The default is 1.

cmdlst A set of commands executed every time the breakpoint is reached. This parameter must consist of valid TRANDEBUG commands separated by semicolons and cannot be NMDEBUG commands. For example, `{AUTORPT;STEP;;;;MODIFY STATUS -15;CONTINUE}`. (Each of the semicolons following `STEP` acts like a carriage return if AUTORPT is turned on.) The *cmdlst* parameter can be up to 80 characters long.

Discussion

This command lets you set a breakpoint at a specified segment number and p-code offset within a Transact/iX program. By default, the segment number is the current segment, so you do not need to specify the location if it resides in the current segment. The system name parameter does not have to be specified if the location resides in the system that is currently executing. If you want to set a breakpoint at the beginning of a system to be called from the current system, just specify the system name. If the breakpoint is set successfully, a message is displayed showing information about the breakpoint.

The range of valid breakpoints is 0 to 63. A maximum of 64 breakpoints can be set with the BREAK SET command. A combined total of up to 64 additional breakpoints can be set with the DATA BREAK REGISTER and DATA BREAK SET commands.

Examples

The following examples illustrate setting breakpoints in TRANDEBUG.

```
TRANDEBUG> break set 24

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   1. CHILD         0       24      1

TRANDEBUG> break set 28,1

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   2. CHILD         1       28       1

TRANDEBUG> break set 10,,PARENT

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   3. PARENT        0       10       1

TRANDEBUG> break set 20,,,3,{DISPLAY MATCH}

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   1. PARENT        0       20       3
         CMD LIST>>>>> DISPLAY MATCH

TRANDEBUG> continue

                BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

                    SYSTEM          SEGMENT       OFFSET
                    ------------------------------------
                    PARENT                0           20
EXECUTING BREAKPOINT CMDLIST:  DISPLAY MATCH;
```

```
                MATCH REGISTER:
                ITEM1      EQ  :   123        AND
                ITEM2      EQ  :   ABC        OR
                ITEM2      EQ  :   DEF

TRANDEBUG> break set 10,,myprog

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   0. MYPROG        0       10       1

TRANDEBUG> bs 3,1, myprog,,{DISPLAY MATCH}

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   1. MYPROG        1        3       1
          CMD LIST>>>>> DISPLAY MATCH

TRANDEBUG> bs 183,5,,6

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   2. MYPROG        5      183       6
```

CONTINUE

Continues execution of the Transact/iX program until a breakpoint is encountered or the program completes execution.

Syntax

```
{ CONTINUE }
{ C        }
```

Discussion

This command resumes execution of the Transact/iX program from within TRANDEBUG. Execution continues until a breakpoint is encountered or the program completes execution. If the trace facility is turned on with the TRACE CODE command, the segment and offsets are displayed before they are executed.

Example

This example shows a typical use of the CONTINUE command.

```
TRANDEBUG> continue

Customer       Dollars Spent
----------------------------
John Smith     $100.25
Jane Doe       $201.75
Joe Customer   $ 21.45

END OF PROGRAM
:
```

DATA BREAK DELETE

Deletes a specified data breakpoint.

Syntax

```
                      [ #breakpoint_number                        ]
{ DATA BREAK DELETE } [ [ item_name     ]                         ]
{ DBD               } [ [ register_name ] [ ,system ]             ]
                      [ ALL                                       ]
```

Parameters

#break-point_number	The number assigned to the data breakpoint you want to delete. You can identify this number by listing the data breakpoints that are set. You must specify "#" when you want to delete by number. For example, to delete the third data breakpoint, type #3.
item_name	The item name at which the data breakpoint is set.
register_name	The register name of the data breakpoint that you want to delete.
system	A string representing the system name in which you want to delete the data breakpoint. The currently executing system is the default.
ALL	A keyword that deletes the data breakpoints currently set.

Discussion

This command allows you to delete breakpoints set at Transact/iX data items. To execute this command, do one of the following:

- Specify the number corresponding to the desired breakpoint.
- Specify the segment and p-code offset of the breakpoint you want to delete.

You can also use the DATA BREAK DELETE command without any parameters by specifying DATA BREAK DELETE or DBD and pressing (Return). TRANDEBUG displays each breakpoint and prompts you to either keep it or delete it.

Examples

In the first example, a DATA BREAK LIST command shows the current data breakpoints. In the second example, DATA BREAK DELETE is used to delete the breakpoint number 0. In the third example, DATA BREAK DELETE deletes the breakpoint using the data item name ITEM2.

DATA BREAK DELETE

```
TRANDEBUG> data break list

             CURRENT DATA BREAKPOINTS:

                SYSTEM  ITEM NAME             LENGTH  COUNT    TYPE
               -----------------------------------------------------
           0. TEST    ITEM1                   4       1      CHANGE
                  CMD LIST>>>>> DISPLAY MATCH;STEP
           1. TEST    ITEM2                   8       1      VALUE
                  EQ VALUE>>>>> abcdefgh


TRANDEBUG> data break delete #0

             BREAKPOINT DELETED:

                SYSTEM  ITEM NAME             LENGTH  COUNT    TYPE
               -----------------------------------------------------
           0. TEST    ITEM1                   4       1      CHANGE
                  CMD LIST>>>>> DISPLAY MATCH;STEP

TRANDEBUG> data break delete ITEM2

             BREAKPOINT DELETED:

                SYSTEM  ITEM NAME             LENGTH  COUNT    TYPE
               -----------------------------------------------------
           1. TEST    ITEM2(1)                8       1      VALUE
                 EQ  VALUE>>>>> abcdefgh
```

DATA BREAK LIST

Lists the data breakpoints currently set in the Transact/iX program.

Syntax

```
{ DATA BREAK LIST }
{ DBL             }
```

Parameters

None.

Discussion

This command allows you to display the data breakpoints that are currently set.

Example

This example shows the listing of a typical set of data breakpoints.

```
TRANDEBUG> data break list

                CURRENT DATA BREAKPOINTS:

                   SYSTEM  ITEM NAME             LENGTH  COUNT    TYPE

                  ----------------------------------------------------
              0. TEST    ITEM1                   4       1      CHANGE
                     CMD LIST>>>>> DISPLAY MATCH;STEP
              1. TEST    ITEM2                   8       1      VALUE
                     LT  VALUE>>>>> abcdefgh
              2. TEST1   ITEM1                   4       1      CHANGE
                     CMD LIST>>>>> DISPLAY KEY;
              3. TEST    ITEM3                   8       1      VALUE
                     NE  VALUE>>>>> 12345678
              4. TEST    MATCH REGISTER                  1      CHANGE
```

The column TYPE shows the data breakpoint type VALUE or CHANGE. VALUE means that a breakpoint is related to a specified value. The value that causes the breakpoint is displayed after the string VALUE>>>>> and can take multiple lines. The relationship between the item name and the value that causes the breakpoint is displayed in front of the string VALUE>>>>>. CHANGE indicates a data breakpoint when the *item_name's* value changes. The command list associated with the breakpoint is listed after the string CMD LIST>>>>> and can also take multiple lines.

DATA BREAK REGISTER

Sets a breakpoint at the specified register.

Syntax

```
{ DATA BREAK REGISTER }
{ DBR                 } register [ ,count [ ,{cmdlist} ] ]
```

Parameters

register The name of the Transact/iX register at which you want the data breakpoint set. Breakpoints can be set at the input, key, match, status, statusin, or update registers. When a register value changes, a data breakpoint occurs and control returns to TRANDEBUG.

count A number indicating how often you want to stop at the data breakpoint, that is, every *n* times it is reached. The default for this parameter is 1. To stop at a data breakpoint every third time the register changes in value, specify 3 for count.

cmdlst A set of commands executed every time the breakpoint is reached. The *cmdlist* parameter must consist of valid TRANDEBUG commands separated by semicolons. It cannot consist of NMDEBUG commands. For example, `{AUTORPT;STEP;;;;MODIFY STATUS -15;CONTINUE}`. (Each of the semicolons following `STEP` acts like a carriage return if AUTORPT is turned on.) The entire DATA BREAK REGISTER command, including the *cmdlst* parameter, can be up to 80 characters long.

Discussion

This command allows you to set a data breakpoint at a specified register. Any time the register value changes, a data breakpoint occurs, and the control returns to TRANDEBUG. This command determines which instruction is modifying a specified register. The data breakpoint always occurs at the instruction *after* the one that modified the register.

To determine the specific instruction that modified the value, just look at the instruction previous to the current one. If the data breakpoint is successfully set, a message is displayed showing information about the breakpoint. You can only set register breakpoints in the current system.

Example

The following example shows how to set data breakpoints at registers.

```
TRANDEBUG> data break register match,,{DISPLAY MATCH}

              DATA BREAKPOINT SET:
```

```
        SYSTEM  ITEM NAME     LENGTH COUNT                  TYPE
        --------------------------------------------------------
    0. CHILD   MATCH REGISTER           1                CHANGE
           CMD LIST >>>>> DISPLAY MATCH

TRANDEBUG> continue

    DATA BREAKPOINT ENCOUNTERED, EXECUTION STOPPED
    ---> MATCH REGISTER CHANGED

    Match Register:
    item1      EQ   :  123        AND
    item2      EQ   :  ABC        OR
    item2      EQ   :  DEF
```

DATA BREAK SET

Sets a breakpoint at the specified data item or data-item value.

Syntax

```
{DATA BREAK SET} item_name [ [ ,length [ ,count [ ,{ cmdlist } ] ] ] ]
{DBS           }           [ ,rel-op ,value [ ,{ cmdlist } ]         ]
```

Parameters

item_name The name of a Transact/iX data item at which the data breakpoint is set. Marker items are not valid item names.

length The number of bytes of the item that will be checked if a data breakpoint occurs. If any portion of the item (up to the specified length) has changed, a data breakpoint occurs. The default value for length is the storage length multiplied by occurrence count. (See DEFINE(ITEM) in Chapter 8, "Transact Verbs.")

count A number indicating how often you want to stop at the data breakpoint, that is, every *n* times it is reached. The default for this parameter is 1.

rel-op An operator that indicates the relationship between the value that is supplied and the data item value that will cause a breakpoint. Valid operators are LT, LE, EQ, GE, GT, and NE.

value The value to compare against. If the data item is of type X or U, the comparison between the data item and the supplied *value* parameter is based on the shorter of the two.

cmdlst A set of commands executed every *n* times the breakpoint is reached. The *cmdlist* parameter must consist of valid TRANDEBUG commands separated by semicolons. It cannot consist of NMDEBUG commands or commands that contain command lists. For example, `{AUTORPT;STEP;;;;MODIFY STATUS-15;CONTINUE}`. (Each of the semicolons following `STEP` acts like a carriage return if AUTORPT is set.) The entire DATA BREAK SET command, including the *cmdlst* parameter, can be up to 80 characters long.

Discussion

This command allows you to set a data-change breakpoint or a data-value breakpoint at a specified item within the Transact/iX program. If a data-change breakpoint is set, the breakpoint *count* value is decremented each time the item value changes. When the count reaches zero, control returns to TRANDEBUG, and the breakpoint information is displayed.

If a data-value breakpoint is set, anytime the item value changes and the desired relation between the data item and the supplied *value* becomes true, a data breakpoint occurs and control returns to TRANDEBUG.

For either breakpoint type, if the item is listed again or if the item is removed from the list register, a data breakpoint occurs and control returns to TRANDEBUG.

This command can be used to determine which instruction is modifying a specific data item. The data breakpoint will always occur at the instruction *after* the one that modified the data item. TRANDEBUG displays a message if the data breakpoint is set successfully. You can only set data breakpoints in the current system.

The range of valid breakpoints is 0 to 63. A combined total of up to 64 breakpoints can be set using the DATA BREAK SET and DATA BREAK REGISTER commands. Another set of up to 64 breakpoints can be set using the BREAK SET command.

Examples

The following examples show the setting of data breakpoints in TRANDEBUG.

```
TRANDEBUG> data break set item1

             DATA BREAKPOINT SET:

                SYSTEM  ITEM NAME            LENGTH  COUNT    TYPE
               -----------------------------------------------------
           0. MYPROG  ITEM1                  4       1      CHANGE


TRANDEBUG> data break set item2,,,{DISPLAY MATCH}

             DATA BREAKPOINT SET:

                SYSTEM  ITEM NAME            LENGTH  COUNT    TYPE
               -----------------------------------------------------
           1. MYPROG  ITEM2                   4       1    CHANGE
                 CMD LIST>>>>> DISPLAY MATCH

TRANDEBUG> continue

           DATA BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:
           'ITEM2' VALUE HAS CHANGED:
           OLD VALUE>>>>> 5
           NEW VALUE>>>>> -1

           MATCH REGISTER:

           ITEM1      EQ    :  123
```

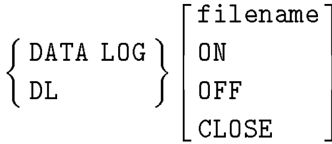
```
TRANDEBUG> data break set item3, EQ,abcd

          DATA BREAKPOINT SET:

             SYSTEM  ITEM NAME            LENGTH  COUNT    TYPE
            -----------------------------------------------------
        2. MYPROG  ITEM3                  4       1      VALUE
              EQ VALUE>>>>>> abcd


TRANDEBUG> dbs item4,LE,3,{DISPLAY MATCH}

          DATA BREAKPOINT SET:

             SYSTEM  ITEM NAME            LENGTH  COUNT    TYPE
            -----------------------------------------------------
        1. MYPROG  ITEM4                  4       1      VALUE
              LE VALUE>>>>> 3
              CMD LIST>>>>> DISPLAY MATCH


TRANDEBUG> continue

        DATA BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:
        'ITEM4' DATA VALUE BREAKPOINT
                LE VALUE>>>>> 3

        MATCH REGISTER:

        ITEM1      EQ    :  123
```

The column TYPE shows the data breakpoint type VALUE or CHANGE. VALUE means that a breakpoint is related to a specified value. The value that causes the breakpoint is displayed after the string VALUE>>>>> and can take multiple lines. The relation between the item name and the value that causes the breakpoint is displayed in front of the string VALUE>>>>>. CHANGE indicates a data breakpoint when the *item_name* value changes. The command list associated with the breakpoint is listed after the string CMD LIST>>>>> and can also take multiple lines.

DATA LOG

Allows logging of TRANDEBUG commands and output to a file.

Syntax



Parameters

filename — The filename to which the TRANDEBUG commands and output are logged. After the file is opened, an implicit 'DATA LOG ON' command executes to begin logging to this file.

`ON` — Activates logging for the currently open data log file.

`OFF` — Deactivates logging for the currently open data log file.

`CLOSE` — Saves the current data log file as a permanent MPE file. This parameter is also the last command written to the data log file.

Discussion

This command allows you to log TRANDEBUG commands and output to a file, then review the output at another time. If you issue this command without specifying parameters, the current data log file name and its status (ON/OFF) are displayed. If the log file already exists, it is purged prior to logging.

Example

The following example uses the DATA LOG command.

```
TRANDEBUG> DATA LOG logfile1
TRANDEBUG> BREAK SET 24,,,,{DISPLAY MATCH;STEP}

             BREAKPOINT SET:

               SYSTEM  SEGMENT   OFFSET   COUNT          COMMAND LIST
               ------------------------------------------------------
           1. TEST          0       24       1  DISPLAY MATCH;STEP

TRANDEBUG> BREAK LIST

    CURRENT BREAKPOINTS:

        SYSTEM  SEGMENT   OFFSET   COUNT          COMMAND LIST
        ------------------------------------------------------
    1. TEST          0       24       1  DISPLAY MATCH;STEP
```

```
TRANDEBUG> DATA LOG CLOSE
TRANDEBUG> :print logfile1
TRANDEBUG> BREAK SET 24,,,,{DISPLAY MATCH;STEP}

    BREAKPOINT SET:

        SYSTEM  SEGMENT   OFFSET   COUNT          COMMAND LIST
        ------------------------------------------------------
    1. TEST          0       24       1  DISPLAY MATCH;STEP

TRANDEBUG> BREAK LIST

    CURRENT BREAKPOINTS:

        SYSTEM  SEGMENT   OFFSET   COUNT          COMMAND LIST
        ------------------------------------------------------
    1. TEST          0       24       1  DISPLAY MATCH;STEP

TRANDEBUG> DATA LOG OFF
TRANDEBUG> DATA LOG CLOSE
```

DEFN

Displays information about the definition of a particular item.

Syntax

DEFN *item_name*

Parameters

item_name The name of the item to define.

Discussion

This command provides information about items within a Transact/iX program. For example, this command is useful if you want to see the size (in bytes) of an item in the data register.

Example

The following example uses the DEFN command:

```
TRANDEBUG> DEFN item1

            ----------------------------------------------
            ITEM1         5 Z+ (   4,   2,   8) = PARENT(1)

            ALIAS ITEMS:                 ALIAS1
                                         ALIAS2
                                         ALIAS3

            DEFINED IN SEGMENT:               0
            ----------------------------------------------
```

DISPLAY BASE

Displays information for specific databases.

Syntax

$$\left\{\begin{array}{l}\texttt{DISPLAY BASE}\\ \texttt{DB}\end{array}\right\}[\,\textit{base_name}\,]$$

Parameters

base_name The database name from which to display information. This can be a database defined in either the current or global system. If you omit this parameter, TRANDEBUG displays information for all databases defined in the current system.

Discussion

This command provides information about a particular database in the system. Information can only be obtained for databases defined in the current or global system. This coincides with the Transact/iX philosophy pertaining to database scoping. A database can be accessed in a system only if it is defined in the current or global system.

Example

The following example displays information about a single database.

```
TRANDEBUG> DISPLAY BASE base1

BASE1("PASS1",1,0,HP3000_16)
  OPEN STATUS: OPEN
  LOCK STATUS: LOCKED   LOCK TYPE: BASE LEVEL
```

The following example displays information about three databases. The database password is only displayed when the user enters the database name. The database password is not displayed if it is hard coded in the application.

```
TRANDEBUG> DISPLAY BASE

BASE1("PASS1",1,0,HP3000_16)
  OPEN STATUS: OPEN
  LOCK STATUS: LOCKED   LOCK TYPE: BASE LEVEL

BASE2("********",1,0,HP3000_32)
  OPEN STATUS: CLOSED
  LOCK STATUS: UNLOCKED

BASE3("PASS1",1,0,HP3000_16)
  OPEN STATUS: OPEN
  LOCK STATUS: LOCKED   LOCK TYPE: BASE LEVEL
```

DISPLAY CALLS

Displays the CALL stack from the currently executing system back to the main program.

Syntax

```
{ DISPLAY CALLS }
{ DCA           }
```

Parameters

None.

Discussion

This command displays the CALL stack. It is useful when you would like to see which system has called the current system.

Example

This is an example of displaying the CALL stack.

```
TRANDEBUG> DISPLAY CALLS

           CURRENT SYSTEM==>  CHILD3
                              CHILD2
                              CHILD1
                              PARENT
           END OF CALL STACK.
```

DISPLAY COMAREA

Displays the contents of the currently active VPLUS communication area (comarea).

Syntax

```
{ DISPLAY COMAREA }
{ DCO             }
```

Parameters

None.

Discussion

This command allows you to display the contents of the currently-active VPLUS comarea. This is useful for obtaining information such as the current form name and next form name.

Example

This is an example of using the DISPLAY COMAREA command.

```
TRANDEBUG> DISPLAY COMAREA

           CURRENTLY ACTIVE COMAREA:
           ------------------------------------------
           CURRENT FORM NAME:           FORM1
           NEXT FORM NAME:              FORM2
           ------------------------------------------
           CSTATUS:            0    ERRFILENUM:      14
           LANGUAGE:           2    FORMSTORESIZE:    2
           COMAREALEN:        60    NUMRECS:          0
           USERBUFLEN:         0    RECNUM:           0
           CMODE:              0    TERM_FILEN:      12
           LASTKEY:            0    RETRIES:          3
           NUMERRS:            0    TERM_OPTIONS:     0
           WINDOWENH:          1    ENVIRON:         37
           MULTIUSAGE:         0    USERTIME:         4
           LABELOPTIONS:       1    IDENTIFIER:      10
           REPEATAPP:          1    LABELINFO:        8
           FREEZAPP:           1    DELETEFLAG:       0
           CFNUMLINES:        20    SHOWCONTROL:      0
           DBBUFLEN:         120    PRINTFILNUM:     12
           LOOKAHEAD:          0    FILEERRNUM:      15
           ------------------------------------------
```

DISPLAY FILE

Displays information from specific MPE/KSAM files.

Syntax

```
{ DISPLAY FILE }
{ DF           } [filename]
```

Parameters

file_name The name of the MPE/KSAM file from which you want to display information. This can be a file defined in the current or global system. If this parameter is omitted, information is displayed for all files defined in the current system.

Discussion

This command offers a way to obtain information about a particular file in the system. Information can be displayed for files defined in the current or global system. This coincides with the Transact/iX philosophy pertaining to file scoping. A file in a system can be accessed only if it is defined in the current or global system.

Example

This is an example of displaying file information.

```
TRANDEBUG> DISPLAY FILE file1

           FILE1(UPDATE(OLD,LOCK,HP3000_16)
                 -80,1,3000,10,1)
           OPEN STATUS: OPEN
           LOCK STATUS: LOCKED

TRANDEBUG> DISPLAY FILE ksam1

           KSAM1(READ(OLD,LOCK,HP3000_16)
           OPEN STATUS: OPEN
           LOCK STATUS: UNLOCKED

TRANDEBUG> DISPLAY FILE

           FILE1(UPDATE(OLD,LOCK,HP3000_16)
                 -80,1,3000,10,1)
           OPEN STATUS: OPEN
           LOCK STATUS: LOCKED

           KSAM1(READ(OLD,LOCK,HP3000_16)
           OPEN STATUS: OPEN
           LOCK STATUS: UNLOCKED
```

DISPLAY INPUT

Displays the contents of the input register.

Syntax

```
{ DISPLAY INPUT }
{ DIN           }
```

Parameters

None.

Discussion

This command allows you to display the contents of the input register.

Example

This is an example of displaying the contents of the input register.

```
TRANDEBUG> DISPLAY INPUT

 INPUT REGISTER:     GALAXY
```

DISPLAY ITEM

Displays the value of a single item, several items, or all items in the data register.

Syntax

$$\left\{\begin{array}{l}\texttt{DISPLAY ITEM}\\ \texttt{DIT}\end{array}\right\}\left[\,item_name_list\,\right]$$

$$item_name_list = \left\{\begin{array}{l} item1\left[\,\texttt{(}subscript\texttt{)}\,\right]:item2\ \left[\,\texttt{(}subscript\texttt{)}\,\right] \\ item1\left[\,\texttt{(}subscript\texttt{)}\,\right]\left[\,,\ \ldots\ itemN\left[\,\texttt{(}subscript\texttt{)}\,\right]\,\right] \\ item1\left[\,\texttt{(}subscript\texttt{)}\,\right]: \\ :item1\left[\,\texttt{(}subscript\texttt{)}\,\right] \end{array}\right\}$$

Parameters

item1, *item2*, *itemN* — The names of the valid 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, selected items, and all items in the data register.

```
TRANDEBUG> DISPLAY ITEM item1

ITEM1:   ABCD
```

```
TRANDEBUG> DISPLAY ITEM item1:item3

ITEM1:   ABCD
ITEM3:   56.78


TRANDEBUG> DISPLAY ITEM

ITEM1:   ABCD
ITEM2:   1234
ITEM3:   56.78
ITEM4:   XYZ
```

DISPLAY KEY

Displays the item in the key register and the corresponding value in the argument register.

Syntax

```
{ DISPLAY KEY }
{ DK          }
```

Parameters

None.

Discussion

This command displays the name of the current item in the key register and the value in the argument register.

Example

The following example displays contents of the key register.

```
TRANDEBUG> DISPLAY KEY

 KEY REGISTER:       ITEM1
 ARGUMENT REGISTER:  12345
```

DISPLAY MATCH

Displays the contents of the match register.

Syntax

```
{ DISPLAY MATCH }
{ DM            }
```

Parameters

None.

Discussion

This command allows you to view the contents of the match register. The output consists of each entry in the match register. The item name is shown along with the relational operator and a connector to the next entry, if applicable. Any special options such as leader, trailer, or scan are included.

Example

The following example displays the match register.

```
TRANDEBUG> DISPLAY MATCH

 MATCH REGISTER:
ITEM1      EQ     :  123
ITEM2      EQ     :  ABC, TR    OR
ITEM2      EQ     :  DEF, SC
```

DISPLAY PERFORMS

Displays the current PERFORM stack.

Syntax

```
{ DISPLAY PERFORMS } [ ALL ]
{ DP               }
```

Parameters

ALL — This parameter displays the PERFORM stack from the beginning of the main program to the current system. This allows you to observe the execution flow from one system to the next.

Discussion

This command traces the perform stack from the current position in the program to the beginning of the perform stack. The command is useful when determining the flow of control within a Transact/iX program. By specifying the ALL option, PERFORM stack information can also be obtained for a system from which the current system was called.

Example

The following example shows a PERFORM stack trace.

```
TRANDEBUG> DISPLAY PERFORMS

 SYSTEM NAME: MYPROG         SEGMENT   OFFSET
---------------------------------------------
 CURRENT POSITION==>             0         54
                                 0         30
                                 0         10
                                 1         78
 END OF PERFORM STACK

TRANDEBUG> DISPLAY PERFORMS ALL

 SYSTEM NAME:  CHILD1         SEGMENT    OFFSET
-----------------------------------------------
 CURRENT POSITION==>             0          54
                                 0          30
                                 0          10
                                 1          78
-----------------------------------------------
 CALLED FROM: PARENT             1          35
                                 1          17
                                 1         123
                                 1          80
------------------------------------------------
END OF PERFORM STACK
```

DISPLAY STATUS

Displays the value in the status register.

Syntax

```
{ DISPLAY STATUS }
{ DS             }
```

Parameters

None.

Discussion

This command displays the contents of the status register.

Example

The following example displays the status register.

```
TRANDEBUG> DISPLAY STATUS

 STATUS REGISTER:   -21
```

DISPLAY STATUSDB

Displays the contents of the database status array returned by the last TurboIMAGE/iX call.

Syntax

```
{ DISPLAY STATUSDB }
{ DSDB             }
```

Parameters

None.

Discussion

This command allows read access to the TurboIMAGE/iX status array set by the last database intrinsic call. The array is displayed in 16-bit format, with the integer value for each 16 bits displayed. The entire array of ten 16-bit words is displayed.

Example

The following example shows how the database status array is displayed.

```
TRANDEBUG> DISPLAY STATUSDB

 DATABASE STATUS ARRAY: -21   0   0   54   58
                          0   0   0    0    0
```

DISPLAY STATUSIN

Displays the value in the statusin register.

Syntax

```
{ DISPLAY STATUSIN }
{ DSIN             }
```

Parameters

None.

Discussion

This command displays the contents of the statusin register.

Example

The following example shows how the statusin register is displayed.

```
TRANDEBUG> DISPLAY STATUSIN

 STATUSIN REGISTER:   -1
```

DISPLAY UPDATE

Displays the specified item in the update register.

Syntax

```
{ DISPLAY UPDATE }
{ DU             }
```

Parameters

None.

Discussion

This command displays the contents of the update register. The output consists of each entry in the update register. The item name is shown along with the relational operator and a connector to the next entry, if applicable.

Example

The following example shows how the update register is displayed.

```
TRANDEBUG> DISPLAY UPDATE

 UPDATE REGISTER:
ITEM1     :  123
ITEM2     :  ABC
```

EDIT

Invokes HP EDIT from within TRANDEBUG. This enables you to edit or browse source files.

Syntax

EDIT [*filename*]

Parameters

filename The name of the file that you want to edit or browse. This file can be qualified by its MPE/iX group and account.

Discussion

This command lets you to edit or browse files while debugging programs. If you do not specify the filename parameter, HP EDIT prompts you for the name of the file that you want to edit.

There are two ways to return to TRANDEBUG from HP EDIT. You can use HP EDIT's EXIT command, which terminates the editing session and returns control to TRANDEBUG. Or, you can press Ctrl P to activate the previous process (such as TRANDEBUG). Then, by typing EDIT and the TRANDEBUG prompt, you return to HP EDIT with your file already entered. See "Activate Previous Process" in Chapter 4 of the *HP EDIT Reference Manual* for more information.

To use HP EDIT, your program file and the MPE/iX group in which it resides must have PH (process handling) capability.

Note HP EDIT is a product sold separately from Transact.

Example

The following example shows the process of invoking and exiting HP EDIT.

```
TRANDEBUG> EDIT

HP EDIT  HP32656A.00.00    (c) COPYRIGHT Hewlett-Packard Co. 1988

FRI, SEP  9, 1989,  9:34 AM

File: Enter file to edit here

      HP EDIT is invoked.....

Ctrl P from within HP EDIT
TRANDEBUG             (c) COPYRIGHT Hewlett-Packard Co. 1988

TRANDEBUG>           << Enter TRANDEBUG commands here >>
```

HELP

Provides online assistance for TRANDEBUG.

Syntax

```
            ┌              ┌ALL      ┐┐
 { HELP }   │ command      │PARMS    ││
 { ?    }   │              └EXAMPLES ┘│
            │ HELPINSTRUCTIONS        │
            └                         ┘
```

Parameters

command	The TRANDEBUG command for which you need information.
`ALL`	Provides information about all of the TRANDEBUG commands.
`PARMS`	Provides information about the parameters for the desired command.
`EXAMPLES`	Provides examples of the execution of this command.
`HELPINSTRUC-TIONS`	Displays a general description of the HELP facility.

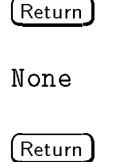
Discussion

The HELP command describes the TRANDEBUG commands and their syntax with examples. Type HELP to invoke the HELP command shell, then type the desired command to obtain information. After receiving the command description, press (Return) for parameter information and press (Return) again for examples. To obtain information on the parameters or if you want to see a specific example, you should specify the corresponding keyword with the command. EXIT returns you to TRANDEBUG.

Example

The following example shows how to obtain information on viewing the contents of the match register.

```
TRANDEBUG> HELP
> DM

        Displays the contents of the match register.

            (Return)
```



```
            None
                   :
            (Return)
```

```
            MATCH REGISTER:
ITEM1       EQ      :  123
ITEM2       EQ      :  ABC, TR    OR
ITEM2       EQ      :  DEF, SC

>exit
```

LABEL BREAK SET

Sets a breakpoint at the specified label.

Syntax

{ LABEL BREAK SET | LBS } *label*[,*segment*[,*count*[,{ *cmdlist* }]]]

Parameters

label A label within the active Transact system. This does not include command labels.

segment The segment number corresponding to the label. If this parameter is omitted, the entire system will be searched for the label. However, if the same label is used in multiple segments of the system, an error message will be returned if the segment is not specified.

count The number of times that the breakpoint label is encountered before stopping. The default count value is 1.

cmdlist A set of commands executed every time the breakpoint count condition is met. The *cmdlist* parameter must consist of valid TRANDEBUG commands separated by semicolons. NMDEBUG commands cannot be used.

Discussion

This command allows you to set a breakpoint in the active Transact system by specifying a label. The segment value is needed if the same label is used in multiple segments. The label is resolved to a p-code offset and a segment which is then processed as if a BREAK SET command was issued.

Note

The output of the LABEL BREAK SET, BREAK LIST, and BREAK DELETE commands do not show the label that was used to set the breakpoint. Only the resulting segment and p-code offset are shown. Additionally, labels cannot be used to delete breakpoints.

Examples

```
TRANDEBUG> label break set mylabel

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   0. MYPROG        0       10       1


TRANDEBUG> label break set seg1label,,,{DISPLAY MATCH}

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   1. MYPROG        1        3       1
         CMD LIST>>>>> DISPLAY MATCH

TRANDEBUG>lbs duplabel,5,6

    BREAKPOINT SET:

       SYSTEM  SEGMENT   OFFSET   COUNT
       --------------------------------
   2. MYPROG        5      183       6
```

LABEL JUMP

Moves the source code window to a specific label in the code.

Syntax

$$\left\{\begin{array}{l}\texttt{LABEL JUMP}\\ \texttt{LJ}\end{array}\right\} \textit{label}\left[\,,\textit{segment}\right]$$

Parameters

label A label within the active Transact system. This does not include command labels.

segment The segment number corresponding to the label. If this parameter is omitted, the entire system is searched for the label. However, if the same label is used in multiple segments of the system, an error will be returned if the segment is not specified.

Discussion

This command lets you change the code displayed in the source code window by specifying a label. If a valid label is specified, the source code window changes to display the source code around the label. The label is resolved to a p-code offset and segment that is then processed as if a page jump command were issued.

Note

The label specified may not appear in the code window because the last occurrence of the p-code offset associated with that label is centered in the code window.

Examples

```
TRANDEBUG> label jump mylabel
```

```
TRANDEBUG> lj duplabel,6
```

LOC

Indicates the p-code offset, segment, and system of the Transact/iX statement that executes next.

Syntax

```
LOC
```

Parameters

None.

Discussion

This command allows you to determine your location in a program when control returns to TRANDEBUG. This command could be used if breakpoints were set in NMDEBUG.

For example, you could set a breakpoint at a call to a TurboIMAGE intrinsic, such as DBGET. By returning to TRANDEBUG and issuing the LOC command, you can determine which Transact/iX statement follows the one that called the DBGET intrinsic. Because the TRANDEBUG and NMDEBUG command sets are two disjointed sets of commands, when the EXIT command is issued from NMDEBUG execution continues until the next Transact/iX statement. At this point, TRANDEBUG regains control and issues a prompt. For this reason, the LOC command can be used only to determine the location of the Transact/iX statement after the NMDEBUG breakpoint.

Example

The following example uses the LOC command.

```
TRANDEBUG> NMDEBUG
nmdebug> b dbget
added: NM     [1] USER ac.001c81b0 dbget
nmdebug> c
Break at: NM  [1] USER ac.001c81b0 dbget
nmdebug> exit
TRANDEBUG> LOC

              CURRENT LOCATION:

              SYSTEM          SEGMENT       OFFSET
              ------------------------------------
              PARENT                0           10
```

LOG

Allows you to log the TRANDEBUG commands to an MPE/iX file.

Syntax

```
      [ filename ]
LOG   [ ON       ]
      [ OFF      ]
      [ CLOSE    ]
```

Parameters

filename — The file to which the TRANDEBUG commands are logged. If this file already exists, it is purged automatically when this command executes. After the file is opened, an implicit 'LOG ON' command is executed to initiate logging to this file.

ON — Activates logging for the currently-open log file.

OFF — Deactivates logging for the currently-open log file.

CLOSE — Saves the current log file as a permanent MPE/iX file. This command is written to the log file as the last command in the log file.

Discussion

This command allows you to log TRANDEBUG commands to an MPE/iX file. The corresponding USE command can be executed to read commands from a log file. If parameters are omitted, the current log file name is displayed along with its status (ON/OFF).

Example

The following example shows a typical command sequence using the LOG commands.

```
TRANDEBUG> LOG logfile1
TRANDEBUG> break set 24,,,,{DISPLAY MATCH;STEP}

               BREAKPOINT SET:

                   SYSTEM  SEGMENT   OFFSET   COUNT         COMMAND LIST
                   ------------------------------------------------------
               1. TEST         0       24       1  DISPLAY MATCH;STEP
TRANDEBUG> break set 28,1

               BREAKPOINT SET:

                   SYSTEM  SEGMENT   OFFSET   COUNT         COMMAND LIST
                   ------------------------------------------------------
               2. TEST         1       28       1
```

```
TRANDEBUG> break list

    CURRENT BREAKPOINTS:

      SYSTEM  SEGMENT   OFFSET   COUNT          COMMAND LIST
      ------------------------------------------------------
  1. TEST          0       24       1  DISPLAY MATCH;STEP
  2. TEST          1       28       1

TRANDEBUG> LOG CLOSE
TRANDEBUG> :print logfile1
break set 24,,,,{DISPLAY MATCH;STEP}
break set 28,1
break list
LOG CLOSE
```

MODIFY INPUT

Modifies the contents of the input register.

Syntax

```
{ MODIFY INPUT }
{ MIN          } [new_value]
```

Parameters

new_value A literal value to place in the input register at the specified location. The *new_value* parameter can be enclosed in quotes if embedded, preceding or trailing blanks are desired. If this parameter is omitted, the current value of the of the input register is displayed. Either a new value can be entered or Return can be pressed to leave the contents of the input register unchanged.

Discussion

This command allows you to change the contents of the input register. If you do not specify a new value for the input register, the current value is displayed. You can then enter the new value or press Return to leave the input register unchanged.

Example

The following example shows how to change the contents of the input register.

```
TRANDEBUG> MODIFY INPUT YES
TRANDEBUG> DISPLAY INPUT

 INPUT REGISTER:     YES

TRANDEBUG> MODIFY INPUT
 INPUT REGISTER:     <YES> := <            >
 Return

TRANDEBUG> DISPLAY INPUT

 INPUT REGISTER:     YES
```

MODIFY ITEM

Modifies the value of a data item in the data register.

Syntax

$$\left\{\begin{array}{l}\texttt{MODIFY ITEM}\\ \texttt{MIT}\end{array}\right\} item_name\ \left[\ (subscripts)\ \right]\left[\ new_value\right]$$

Parameters

item_name The name of the data item in the data register to modify.

subscripts 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 modified is an array, only the first element in the array will be modified. If this parameter is specified and the item is not an array, an error message is displayed and the item is left unchanged.

new_value A literal value to place in the data register at the specified item location. The *new_value* parameter can be enclosed in quotes if embedded, preceding or trailing blanks are desired. If this parameter is omitted, the current value of the item is displayed; you can either enter a new value or press (Return) to leave the item unchanged.

Discussion

This command allows you to change the value of an item in the data register. If the new value cannot fit into the storage length of the item an error message is displayed. This truncation can lead to incorrect values in the data register, so be careful when you modify items. The main data types to be concerned with are X, U, 9, Z, and P.

Examples

The first example shows how to change item1. The second example shows how to use embedded blanks within the new_value parameter.

```
TRANDEBUG> MODIFY ITEM item1 ABCD
TRANDEBUG> DISPLAY ITEM item1

ITEM1          :   ABCD



TRANDEBUG> MODIFY ITEM item1 "ABCD EFGH"
TRANDEBUG> DISPLAY ITEM item1

ITEM1          :   ABCD EFGH
```

MODIFY KEY

Modifies the item in the key register.

Syntax

```
{ MODIFY KEY }
{ MK         } [ new_item ]
```

Parameters

new_item The name of the item to place in the key register. If you do not specify a new value for the key register, the current value is displayed. You can then either type the new value or press Return to leave the key register unchanged.

Discussion

This command allows you to change the item in the key register. After the new item is placed in the key register, its value in the data register is placed in the argument register. You are then shown its new value and are given the option of changing it. If you do not want to change the argument register, press Return.

Examples

The following examples show how to change the item in the key register.

```
TRANDEBUG> MODIFY KEY item1
 ARGUMENT REGISTER: <abcde> := <fghij>

TRANDEBUG> DISPLAY KEY

 KEY REGISTER:      ITEM1
 ARGUMENT REGISTER: fghij


TRANDEBUG> MODIFY KEY

 KEY REGISTER       <ITEM1   > := <ITEM2   >
 ARGUMENT REGISTER  <xyz           > := <zyx     >
```

MODIFY MATCH

Modifies the specified item in the match register.

Syntax

```
{ MODIFY MATCH }
{ MM           } item_name [ new_value ]
```

Parameters

item_name The name of the data item in the match register to modify.

new_value The value to be assigned to the specified item in the match register. The *new_value* parameter can be enclosed in quotes if embedded, preceding or trailing blanks are desired. If you do not specify a new value for the match register item, the current value is displayed. Then, you can either enter the new value or press (Return) to leave the match register item unchanged.

Discussion

This command allows you to selectively modify items in the match register. If the item occurs more than once in the match register, you are prompted for which occurrence to change. The new value is converted to the data type of the item before it is placed in the match register. If the value cannot fit into the storage length of the item an error message is displayed. This truncation can cause erroneous results during program execution, so before modifying an item, be aware of the storage length of the item you are modifying. You cannot use this command to add or delete items from the match register.

Example

The following example shows the modification of the match register. When item2 is modified, TRANDEBUG prompts to determine which of the occurrences of item2 should be modified.

```
TRANDEBUG> MODIFY MATCH item1 123
TRANDEBUG> DISPLAY MATCH

 MATCH REGISTER:
ITEM1      EQ  :   123
ITEM2      EQ  :   ABC        OR
ITEM2      EQ  :   DEF

TRANDEBUG> MODIFY MATCH item2 XYZ

1. ITEM2    EQ  :   ABC
2. ITEM2    EQ  :   DEF
 ENTER NUMBER OF ONE TO MODIFY: 1
```

```
TRANDEBUG> DISPLAY MATCH

 MATCH REGISTER:
ITEM1      EQ  :   123
ITEM2      EQ  :   XYZ        OR
ITEM2      EQ  :   DEF
```

MODIFY STATUS

Modifies the value in the status register.

Syntax

```
{ MODIFY STATUS } [ new_value ]
{ MS            }
```

Parameters

new_value The new value to place into the status register. This value must be a valid 32-bit integer value; a warning message appears if it is an invalid value. If you do not specify a new value for the STATUS register, the current value is displayed. Then, you can either enter the new value or press (Return) to leave the status register unchanged.

Discussion

This command allows modification of the status register. If you type a non-integer value, a warning message appears.

Examples

The following examples illustrate modification of the status register.

```
TRANDEBUG> MODIFY STATUS -21
TRANDEBUG> DISPLAY STATUS

 STATUS REGISTER:   -21

TRANDEBUG> MODIFY STATUS

 STATUS REGISTER: <-21> := 4

TRANDEBUG> DISPLAY STATUS
 STATUS REGISTER: 4
```

MODIFY UPDATE

Modifies the specified item in the update register.

Syntax

$$\left\{\begin{array}{l}\texttt{MODIFY UPDATE}\\ \texttt{MU}\end{array}\right\} item_name\ \left[\ (subscript)\ \right]\left[\ new_value\ \right]$$

Parameters

item_name — The name of the item to modify in the UPDATE register.

subscript — A numeric value used to select a particular element in a single dimensional array. If this parameter is omitted and the item being modified is an array, only the first element in the array is modified. If this parameter is specified and the item is not an array, a warning message is displayed and the item is not modified.

new_value — The value to be assigned to the specified item in the update register. If you do not specify a new value for the update register item, the current value is displayed. Then, you can either enter the new value or press (Return) to leave the update register item unchanged.

Discussion

This command allows you to modify items in the update register selectively. The new value is converted to the data type of the item before it is placed in the update register. If the value does not fit into the storage length of the item, it is truncated and a warning message is displayed. This truncation can cause erroneous results during the program execution, so before modifying an item, be aware of the storage length of the item you are modifying. You cannot use this command to add or delete items from the update register.

Example

The following example shows the modification of the update register.

```
TRANDEBUG> MODIFY UPDATE item1 123
TRANDEBUG> DISPLAY UPDATE

 UPDATE REGISTER:
ITEM1      EQ  :   123
ITEM2      GT  :   ABC

TRANDEBUG> MODIFY UPDATE item2

ITEM2          : <ABC>     := <DEF>
```

```
TRANDEBUG> DISPLAY UPDATE

 UPDATE REGISTER:
ITEM1      EQ  :   123
ITEM2      GT  :   DEF
```

NMDEBUG

Transfers control from TRANDEBUG to NMDEBUG.

Syntax

```
{ NMDEBUG }
{ NM      }
```

Parameters

None.

Discussion

This command allows you to enter NMDEBUG. Breakpoints issued within NMDEBUG are not recognized by TRANDEBUG. We recommend that you do not manipulate, from within this command shell, breakpoints set at Transact/iX statements.

Example

The following example shows how you can execute NMDEBUG commands from within TRANDEBUG.

```
TRANDEBUG> NMDEBUG
$1 ($34) nmdebug> b FWRITE
$2 ($34) nmdebug> b DBOPEN
$3 ($34) nmdebug> won
$4 ($34) nmdebug> exit
```

PAGE BACK

Pages the source code window backwards through the source file.

Syntax

```
{ PAGE BACK }
{ PB        }
```

Parameters

None.

Discussion

This command pages the source code window backward and allows you to browse the source code while a program is being debugged.

PAGE FORWARD

Pages the source code window forward through the source file.

Syntax

```
{ PAGE FORWARD }
{ PF           }
```

Parameters

None.

Discussion

This command pages the source code window forward and allows you to browse the source code while it is being executed.

PAGE JUMP

Moves the source code window to a specified segment and offset in the program.

Syntax

$$\left\{\begin{array}{l}\texttt{PAGE JUMP}\\ \texttt{PJ}\end{array}\right\} \textit{offset}\ \left[\,,\textit{segment}\right]$$

Parameters

offset	The specified p-code offset to which the window jumps.
segment	The specified segment to which the window jumps. The current segment is the default.

Discussion

This command allows you to jump to a specified location in the listing file. If the offset does not exist, you are placed at the closest offset that TRANDEBUG can find. If you specify an offset larger than the last segment and offset in the program, an error message is displayed.

PRINT

Alters or displays the Transact/iX PRINT option.

Syntax

```
        [ON ]
PRINT   [OFF]
```

Parameters

ON — Turns on the PRINT flag to direct the output generated from any DISPLAY or OUTPUT verbs to the line printer. This action is the same as issuing a SET(OPTION) PRINT statement within the Transact/iX source program.

OFF — Turns off the PRINT flag.

Discussion

This command allows you to change, programmatically, the current value for the Transact/iX PRINT option. By toggling this option, you can direct output to the line printer. If neither ON or OFF is specified, the current value of the PRINT option is displayed and you are prompted to either keep it or modify it.

Example

The following example alters the PRINT option.

```
TRANDEBUG> PRINT

           PRINT FLAG CURRENTLY OFF

CHANGE OPTION? (Y/N):  N

TRANDEBUG> CONTINUE

Customer       Dollars Spent
----------------------------
John Smith     $100.25
Jane Doe       $201.75
Joe Customer   $ 21.45

                BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

                    SYSTEM          SEGMENT       OFFSET
                    ------------------------------------
                1. PARENT                 0           28

TRANDEBUG> PRINT ON
```

```
TRANDEBUG> CONTINUE { output is being redirected }

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

           SYSTEM          SEGMENT       OFFSET
           ------------------------------------
       1. PARENT                0            28

TRANDEBUG> PRINT OFF
```

REPEAT

Alters or displays the Transact/iX REPEAT option.

Syntax

```
REPEAT [ON ]
       [OFF]
```

Parameters

ON Turns on the REPEAT flag to repeat the current Transact/iX command sequence. This is the same as issuing a SET(OPTION) REPEAT statement within the Transact/iX source file.

OFF Turns off the REPEAT flag.

Discussion

This command provides a programmatic method of changing the Transact/iX repeat option. If neither ON nor OFF is specified, the current value of the REPEAT option is displayed and you can modify its value.

Example

The following example shows how to turn on and turn off the repeat flag using the REPEAT command.

```
TRANDEBUG> REPEAT

             REPEAT FLAG CURRENTLY OFF

 CHANGE OPTION?(Y/N) N

TRANDEBUG> CONTINUE
Customer      Dollars Spent
----------------------------
John Smith     $100.25
Jane Monroe    $201.75
James Lorenz   $ 21.45

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

              SYSTEM          SEGMENT       OFFSET
              ------------------------------------
          1. PARENT                0            28

TRANDEBUG> REPEAT ON
```

```
TRANDEBUG> CONTINUE
Customer       Dollars Spent
----------------------------
Joe Customer   $ 50.45
Jane Monroe    $201.75
James Lorenz   $100.25
Customer       Dollars Spent
----------------------------
Joe Customer   $ 21.45
Jane Doe       $201.75
John Smith     $100.25
Customer       Dollars Spent
----------------------------
Joe Customer   $ 30.00
Jane Doe       $201.75
John Smith     $100.25
<Ctrl-Y>

TRANDEBUG> REPEAT OFF
```

SORT

Alters or displays the Transact/iX SORT option.

Syntax

```
        [ ON  ]
   SORT [ OFF ]
```

Parameters

ON — Turns on the SORT flag to force the OUTPUT verb to sort any records that are selected. This is the same as issuing a SET(OPTION) SORT statement within the Transact/iX source file.

OFF — Turns off the SORT flag.

Discussion

This command allows you to change, programmatically, the SORT option to force sorting of the data for the OUTPUT verb. If neither ON or OFF is specified the current value of the SORT option is displayed, and you can change it. The items in the LIST register are used for sort keys; the precedence is set by the order they are listed. The primary key is the item that was listed first, the secondary key is the item listed second, and so on.

Example

The following example shows how to use the SORT command.

```
TRANDEBUG> SORT

             SORT FLAG CURRENTLY OFF

 CHANGE OPTION?(Y/N)? N

TRANDEBUG> CONTINUE
Customer       Dollars Spent
----------------------------
Jane Doe       $201.75
John Smith     $100.25
Joe Customer   $ 21.45

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

               SYSTEM         SEGMENT      OFFSET
               -----------------------------------
           1. PARENT               0          28

TRANDEBUG> SORT ON
```

```
TRANDEBUG> CONTINUE
Customer       Dollars Spent
----------------------------
Joe Customer   $ 21.45
Jane Doe       $201.75
John Smith     $100.25

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

               SYSTEM          SEGMENT       OFFSET
               ------------------------------------
           1. PARENT               0            28

TRANDEBUG> SORT OFF
```

STEP

Continues execution of the Transact/iX program for a specified number of statements.

Syntax

$\left\{ \begin{array}{l} \texttt{STEP} \\ \texttt{S} \end{array} \right\} [\,\textit{number_of_steps}\,]$

Parameters

number_of_steps The desired number of Transact statements to execute until control returns to TRANDEBUG. The default value of 1 is used if you omit this parameter.

Discussion

This command allows you to single-step through the execution of a program on a statement-by-statement basis. This allows you to check item values and register values after each statement has executed. As you stop at each statement, the p-code offset and segment number for that statement is displayed.

Example

The following example shows how to use the STEP command and the TRACE CODE command in conjunction with STEP.

```
TRANDEBUG> STEP

           EXECUTION STOPPED:

              SYSTEM          SEGMENT       OFFSET
              ------------------------------------
              PARENT                0           28

TRANDEBUG> STEP 5

           EXECUTION STOPPED:

               SYSTEM          SEGMENT       OFFSET
               ------------------------------------
               PARENT                1           56

TRANDEBUG> TRACE CODE ON
```

```
TRANDEBUG> STEP 5

            EXECUTION TRACE:
               SYSTEM          SEGMENT       OFFSET
               ------------------------------------
               PARENT                0           28
               PARENT                0           34
               PARENT                0           36
               PARENT                1           10
               PARENT                1           56
```

TPRINT

Alters or displays the Transact/iX TPRINT option.

Syntax

```
        [ON ]
TPRINT  [OFF]
```

Parameters

ON — Turns on the TPRINT flag to format the output generated from the DISPLAY or OUTPUT verb for printing. This action is the same as the Transact/iX SET(OPTION) TPRINT statement.

OFF — Turns off the TPRINT flag.

Discussion

This command allows you to modify, programmatically, the Transact/iX TPRINT option. This allows you to selectively turn on or off the line-printer formatting for DISPLAY or OUTPUT verb. If neither ON or OFF is specified, the current value of the TPRINT option is displayed, and you can change it.

Example

The following example shows how to use the TPRINT command.

```
TRANDEBUG> TPRINT

             TPRINT FLAG CURRENTLY OFF

CHANGE OPTION?(Y/N) N

TRANDEBUG> CONTINUE
Customer       Dollars Spent
----------------------------
John Smith     $100.25
Jane Doe       $201.75
Joe Customer   $ 21.45

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

           SYSTEM          SEGMENT       OFFSET
           ----------------------------------
       1. PARENT                0           28

TRANDEBUG> TPRINT ON
```

```
TRANDEBUG> CONTINUE  { output is in line printer format }
Customer        Dollars Spent
----------------------------
John Smith      $100.25
Jane Doe        $201.75
Joe Customer    $ 21.45

           BREAKPOINT ENCOUNTERED, EXECUTION STOPPED:

           SYSTEM           SEGMENT       OFFSET
           ------------------------------------
       1. PARENT                0            28

TRANDEBUG> TPRINT OFF
```

TRACE

Turns the trace flag on or off for the specified type of trace.

Syntax

```
{ TRACE } { CODE  } [ ON  ]
{ TR    } { IMAGE } [ OFF ]
          { MPE   }
```

Parameters

CODE	Allows tracing of p-code offsets and segment numbers prior to their execution.
IMAGE	Displays the parameters and return status for TurboIMAGE intrinsic calls made by Transact/iX.
MPE	Displays parameters for MPE/KSAM file reads and writes done by Transact/iX.
ON	Turns on the specified trace flag.
OFF	Turns off the specified trace flag.

Discussion

This command provides a tracing mechanism for the execution of the Transact/iX program. You can trace any combination of IMAGE, MPE/KSAM, and source execution. If neither ON or OFF is specified, the current value of the specified TRACE flag is displayed and you can change it. Only intrinsic calls made explicitly by Transact/iX appear in the trace. Those made by the PROC verb do not appear.

Examples

The first example shows how to use the TRACE command with the CODE option.

```
TRANDEBUG> TRACE CODE ON
TRANDEBUG> CONTINUE

           EXECUTION TRACE:

           SYSTEM          SEGMENT       OFFSET
           ------------------------------------
           PARENT                0           28
           PARENT                0           34
           PARENT                1           10
           PARENT                1           56

END OF PROGRAM
:
```

The second example shows how to use the TRACE command with the IMAGE option.

```
TRANDEBUG> TRACE IMAGE ON
TRANDEBUG> CONTINUE

-------------------------------------------------------
FIND(SERIAL)  DB COND: 0   STATUS: 0   RECNO: 35
              BASE:  BASE1        SET:  SET1

POSN:   LIST:                  DATA:
 0       ITEM1                  value1
 8       ITEM2                  value2
-------------------------------------------------------

-------------------------------------------------------
FIND(SERIAL)  DB COND: 11  STATUS: -11 RECNO: 35
              BASE:  BASE1        SET:  SET1

POSN:   LIST:                  DATA:
 0       ITEM1                  value1
 8       ITEM2                  value2
-------------------------------------------------------

END OF PROGRAM
:
```

The third example shows how to use the TRACE command with the MPE option.

```
TRANDEBUG> TRACE MPE ON
TRANDEBUG> CONTINUE

-------------------------------------------------------
FIND(SERIAL)  CCODE: CCE   STATUS: 0   RECNO: 35
              FILE:  FILE1

POSN:   LIST:                  DATA:
 0       ITEM1                  value1
 8       ITEM2                  value2
-------------------------------------------------------

-------------------------------------------------------
FIND(SERIAL)  CCODE: CCG   STATUS: -1  RECNO: 35
              BASE:  BASE1        SET:  SET1

POSN:   LIST:                  DATA:
 0       ITEM1                  value1
 8       ITEM2                  value2
-------------------------------------------------------

END OF PROGRAM
:
```

USE

Reads TRANDEBUG commands from the specified MPE/iX file.

Syntax

USE *filename*

Parameters

filename The MPE/iX filename to use as input to TRANDEBUG.

Discussion

This command allows you to use an MPE/iX file as input to TRANDEBUG. You can either create this file using an editor or you can use the LOG command. Each line in the file is treated as one TRANDEBUG command and execution of this file continues until the EOF is reached or an error occurs.

Example

The following example shows how the USE command works.

```
TRANDEBUG> :print logfile1
break set 24,,,,{DISPLAY MATCH;STEP}
break set 28,1
break list
LOG CLOSE
TRANDEBUG> USE logfile1
TRANDEBUG> break set 24,,,,{DISPLAY MATCH;STEP}

              BREAKPOINT SET:

                SYSTEM  SEGMENT   OFFSET   COUNT         COMMAND LIST
                ------------------------------------------------------
             1. TEST          0       24       1  DISPLAY MATCH;STEP

TRANDEBUG> break set 28,1

              BREAKPOINT SET:

                SYSTEM  SEGMENT   OFFSET   COUNT         COMMAND LIST
                ------------------------------------------------------
             2. TEST          1       28       1
```

```
TRANDEBUG> break list

           CURRENT BREAKPOINTS:

           SYSTEM  SEGMENT   OFFSET   COUNT         COMMAND LIST
           ------------------------------------------------------
       1. TEST          0       24       1  DISPLAY MATCH;STEP
       2. TEST          1       28       1


TRANDEBUG> LOG CLOSE
TRANDEBUG> :print logfile1
break set 24,,,,{DISPLAY MATCH;STEP}
break set 28,1
break list
LOG CLOSE
```

VERSION

Outputs the current version of the Transact/iX run-time library.

Syntax

```
VERSION
```

Parameters

None.

Discussion

This command allows you to determine the version of the Transact/iX run-time library being used. This version should match the version on the banner when the compiler is invoked.

Example

```
TRANDEBUG> VERSION
TRANSACT/iX Library Version:  A.05.00
```

WINDOW LENGTH

Adjusts the size of the source code window.

Syntax

```
{ WINDOW LENGTH }
{ WL            } new_size
```

Parameters

new_size The size to which the window is to be adjusted. The new size must be in the range of 1-18 or an error message is displayed.

Discussion

This command allows you to adjust the size of the source code window.

WINDOW OFF

Turns off the source code window.

Syntax

```
⎧ WINDOW OFF ⎫
⎩ WOFF       ⎭
```

Parameters

None.

Discussion

This command turns off the source code window. If you want to display a series of values, you should first use this command to turn off the window.

WINDOW ON

Turns on the source code window.

Syntax

```
{ WINDOW ON }
{ WON       }
```

Parameters

None.

Discussion

This command turns on the source code window. After issuing this command, you can then view the next Transact/iX source to be executed.

A

Flowcharts of File and Database Operations

The flowcharts in this appendix provide an overview of the major database, file system, and VPLUS intrinsic calls issued by Transact to perform data management operations. The charts provide a basic understanding of the steps that occur when a verb executes. However, they should not be viewed as definitive explanations of all events caused by a given verb.

Calls that are shown in brackets occur only when circumstances dictate.

Rules that govern use of calls for locking, unlocking, and opening files are as follows;

DBLOCK
: If a database or data set is not already locked, DBLOCK is applied according to the rules given in Table 6-2 (if optimized locking is used) or Table 6-4.

DBUNLOCK
: This call is applied in either of the following circumstances:

  If the current verb invoked the original lock; if the database or data set is locked; and if the lock option is not used

  OR

  The last record has been accessed for a multiple record operation (for example, CHAIN, SERIAL, RCHAIN, or RSERIAL) when the lock option is used.

FOPEN
: This call is used if the file is not already open.

FLOCK
: This call is used if the file is not already locked, according to the rules given in Table 6-4.

FUNLOCK
: This call is applied if the program is at the same instruction which invoked the FLOCK.

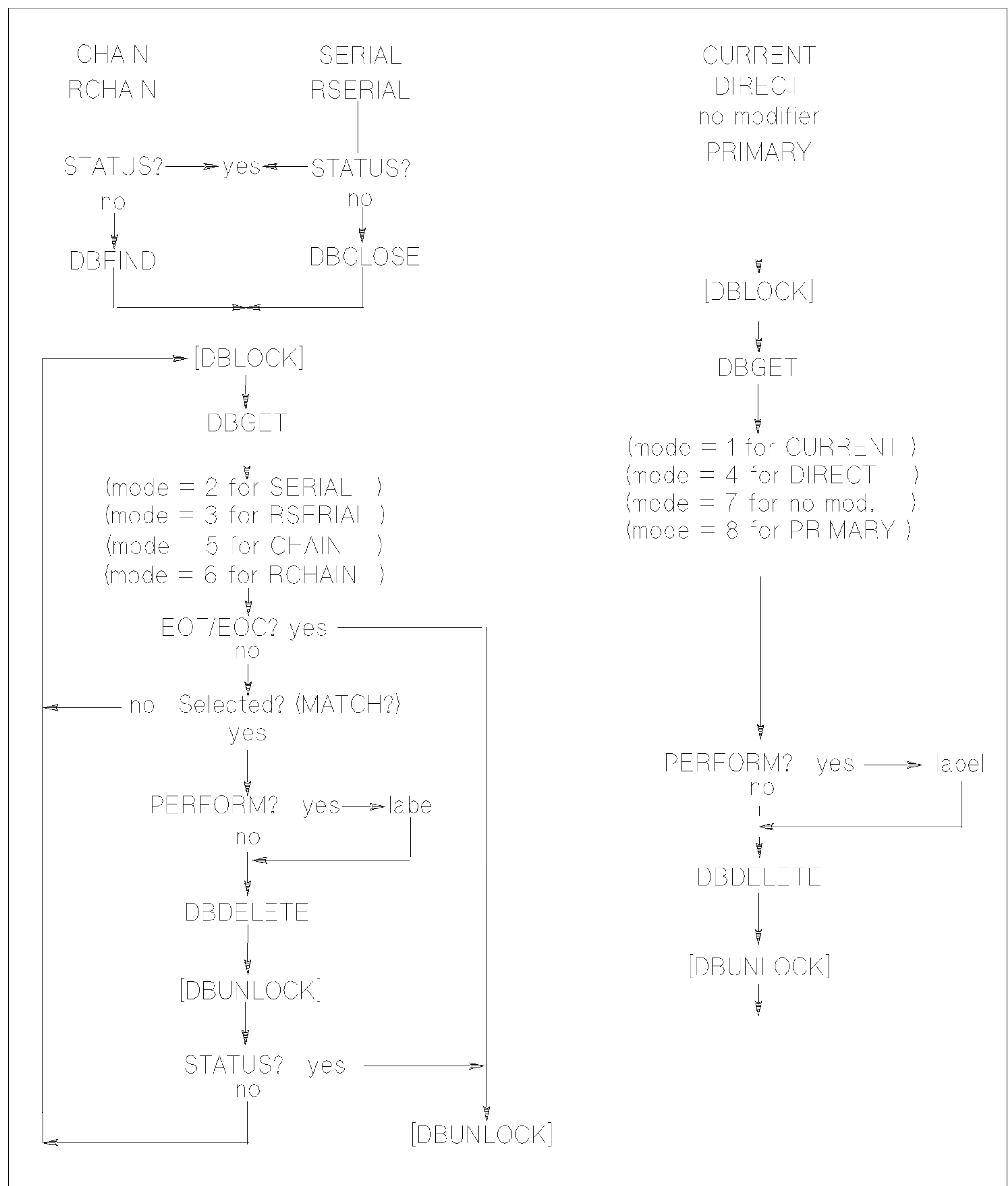
Separate flowcharts are included for each verb to describe database and file access.

Flowcharts are given for the following verbs:

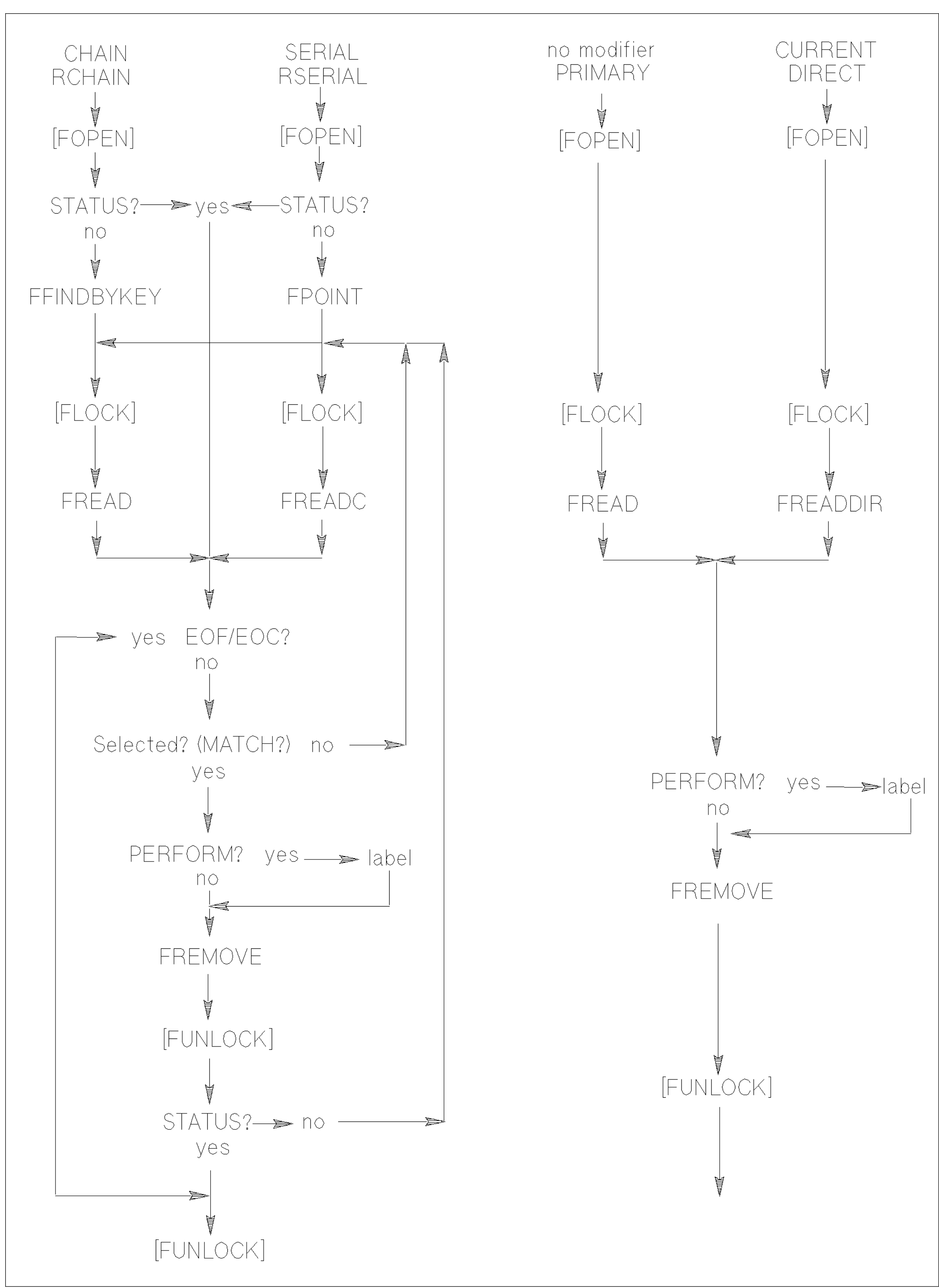
DELETE	- for a data set or a KSAM file operation
FIND	- for a data set or a KSAM or MPE file operation
GET	- for a data set or a KSAM, MPE, or VPLUS file operation
OUTPUT	- for a data set or a KSAM or MPE file operation
PATH	- for a data set or a KSAM file operation
PUT	- for a data set or a KSAM, MPE, or VPLUS file operation
REPLACE	- for a data set or a KSAM or MPE file operation
SET	- for a VPLUS file operation
UPDATE	- for a data set, or a KSAM, MPE, or VPLUS file operation

DELETE Charts

Execution of a DELETE verb for a TurboIMAGE data set access results in the following:

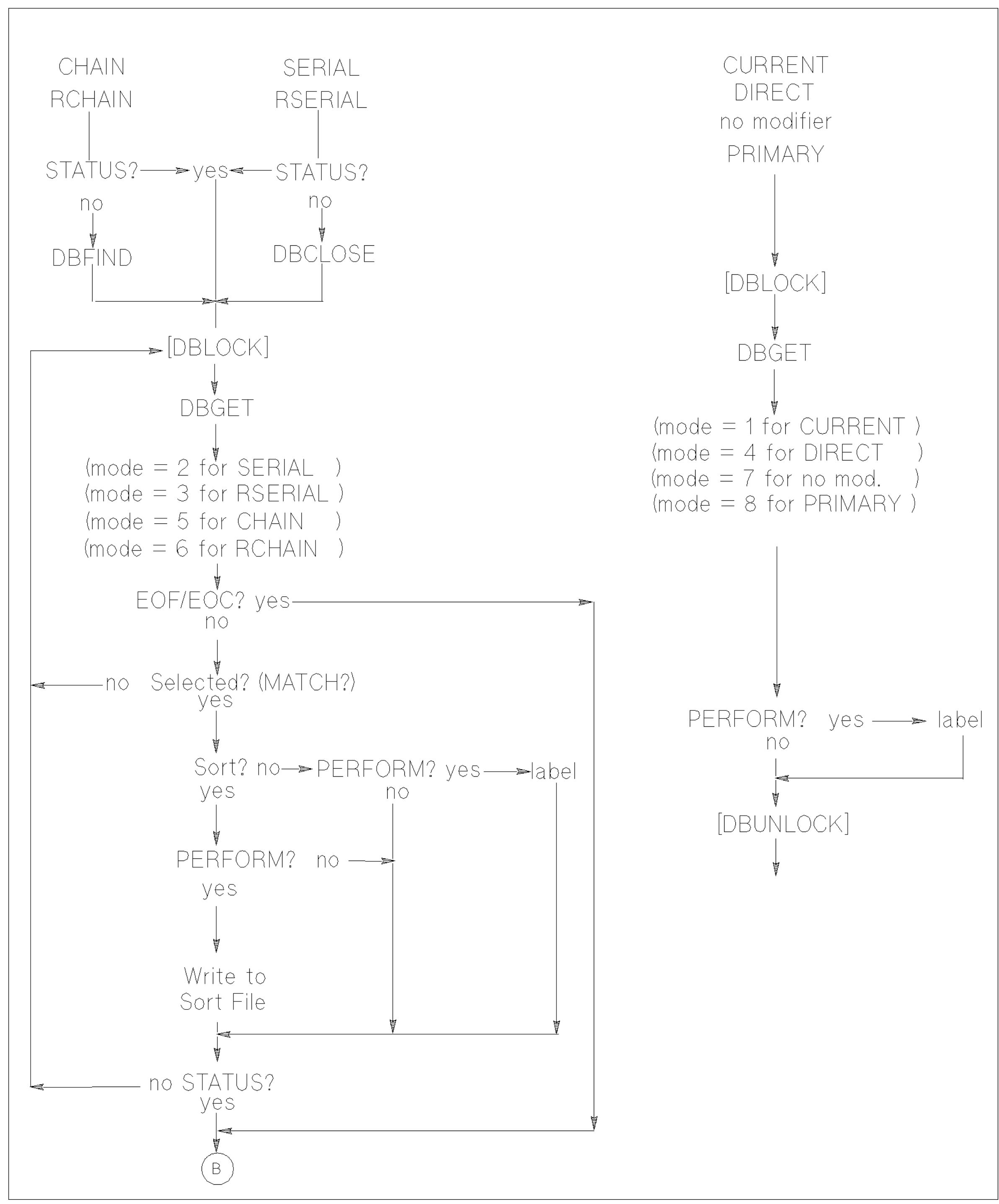


Execution of a DELETE verb for a KSAM file results in the following:

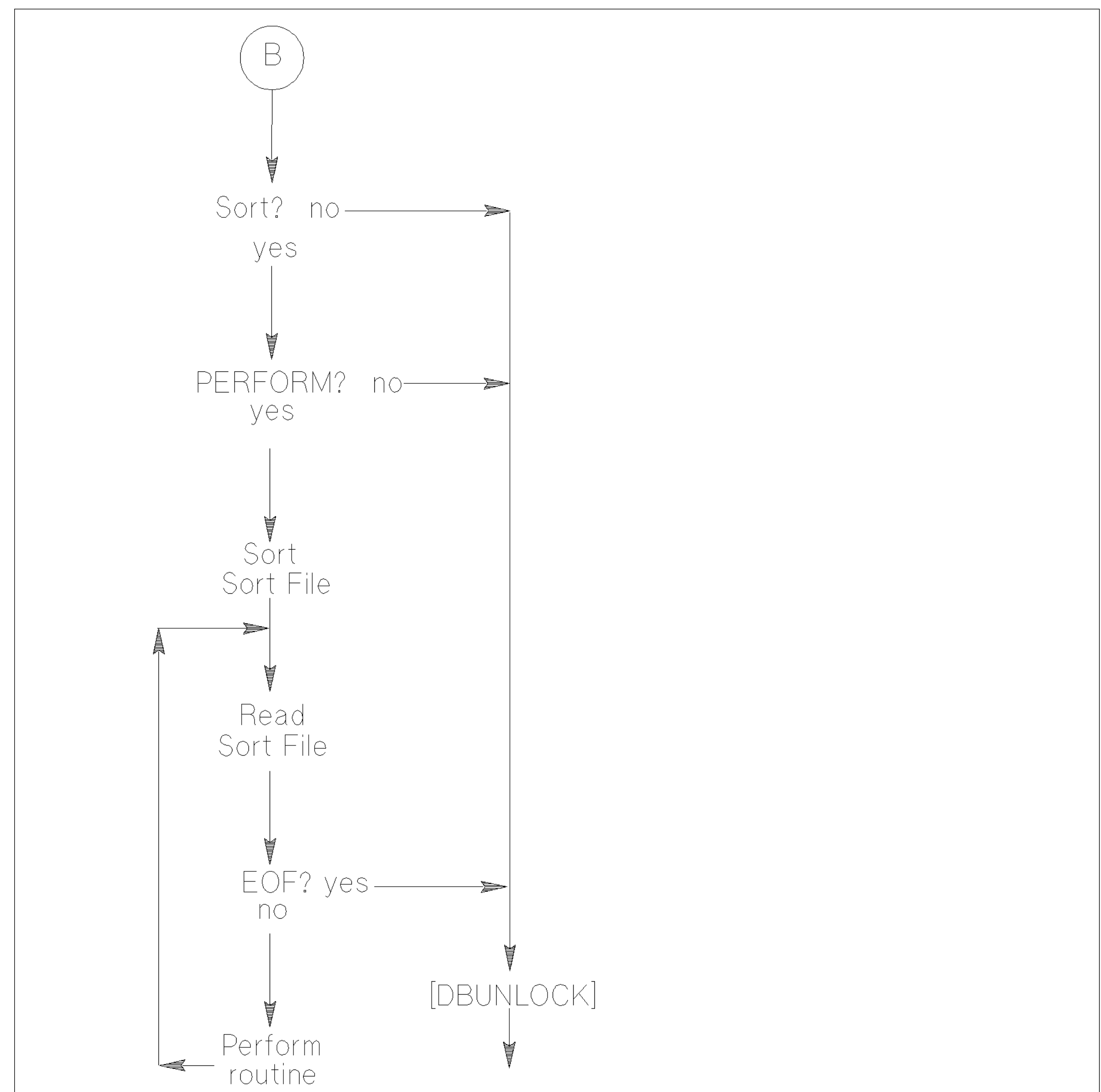


FIND Charts

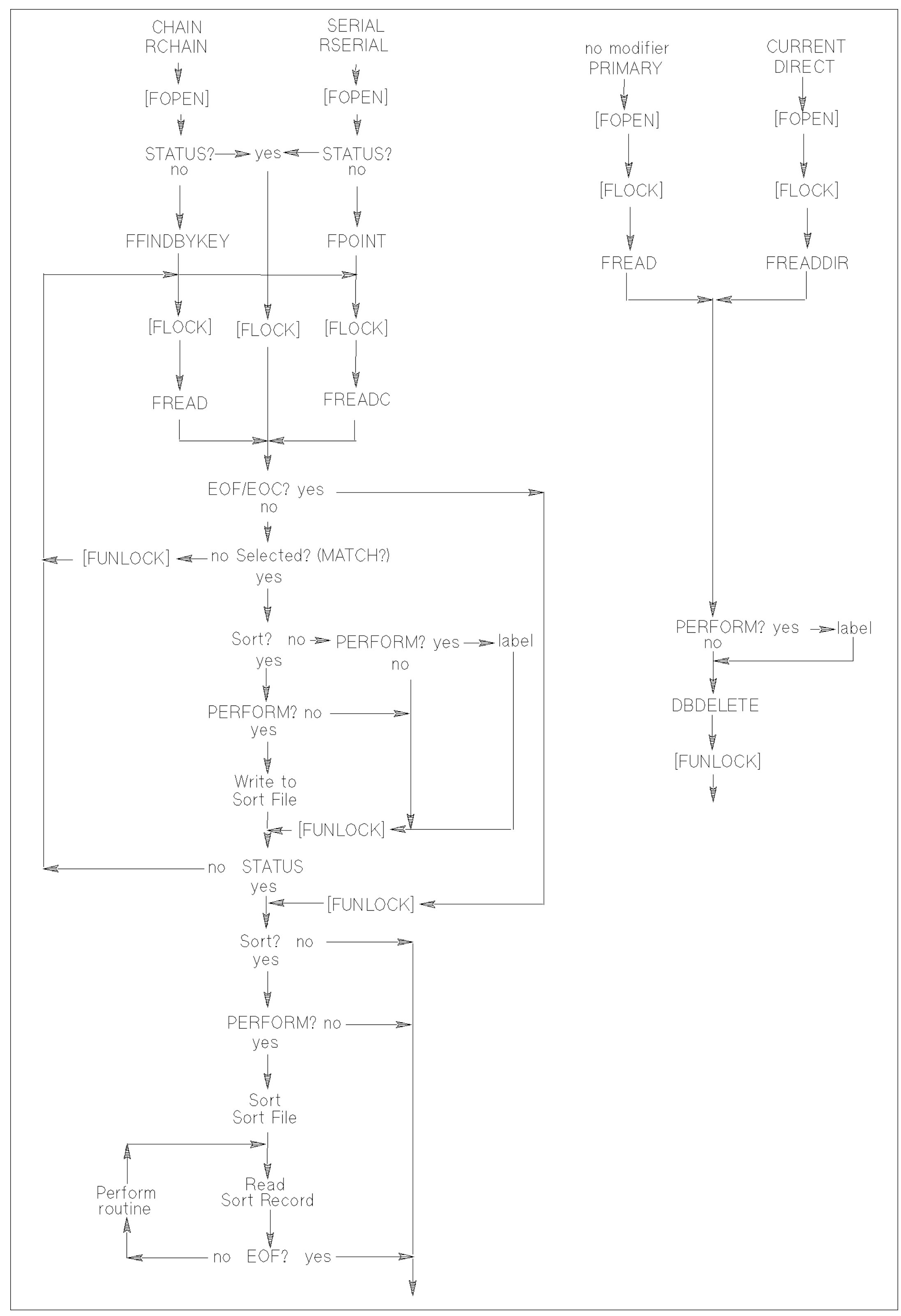
Execution of the FIND verb for a TurboIMAGE data set access results in the following:



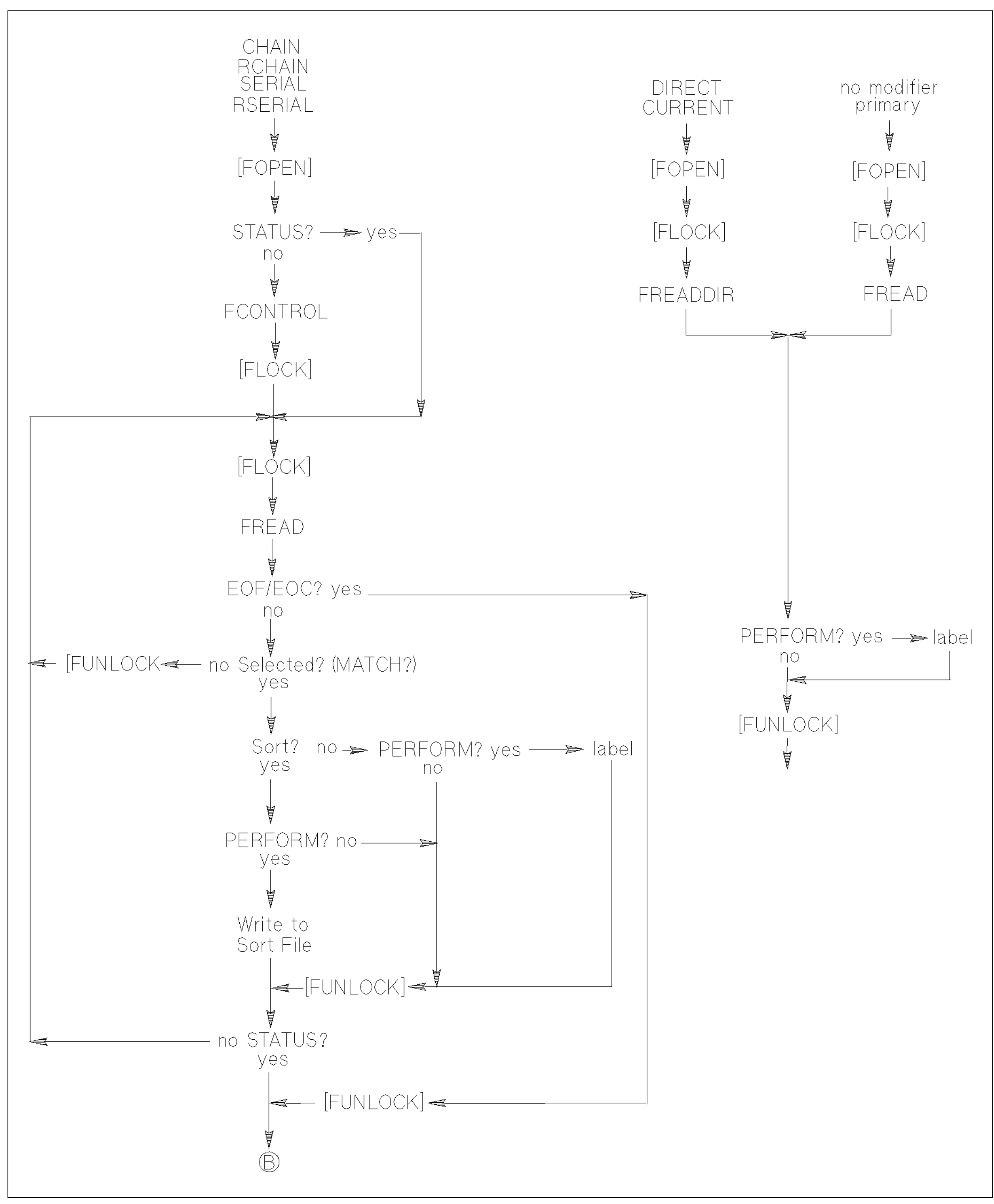
Execution of the FIND verb for a TurboIMAGE data set access (continued):



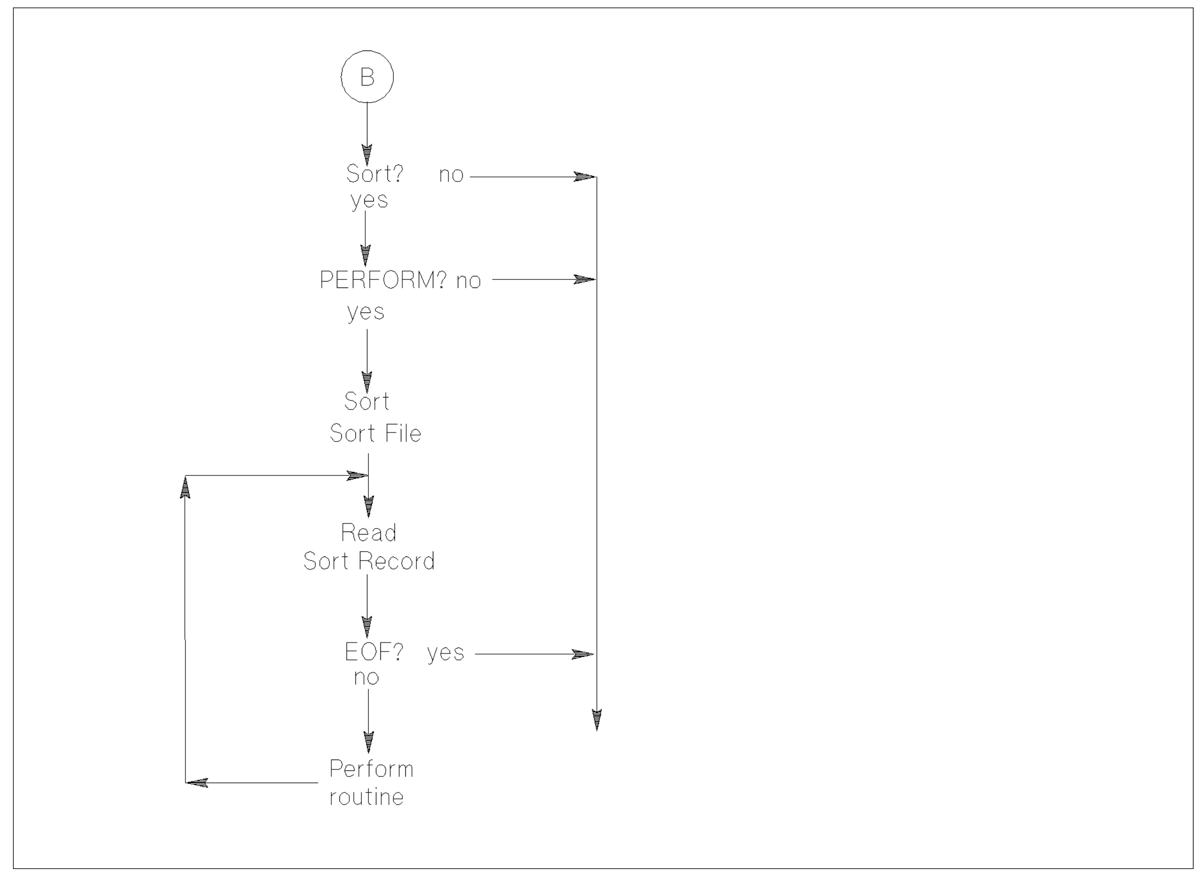
Execution of a FIND verb for a KSAM file is shown below:



Execution of a FIND verb for an MPE file results in the following:

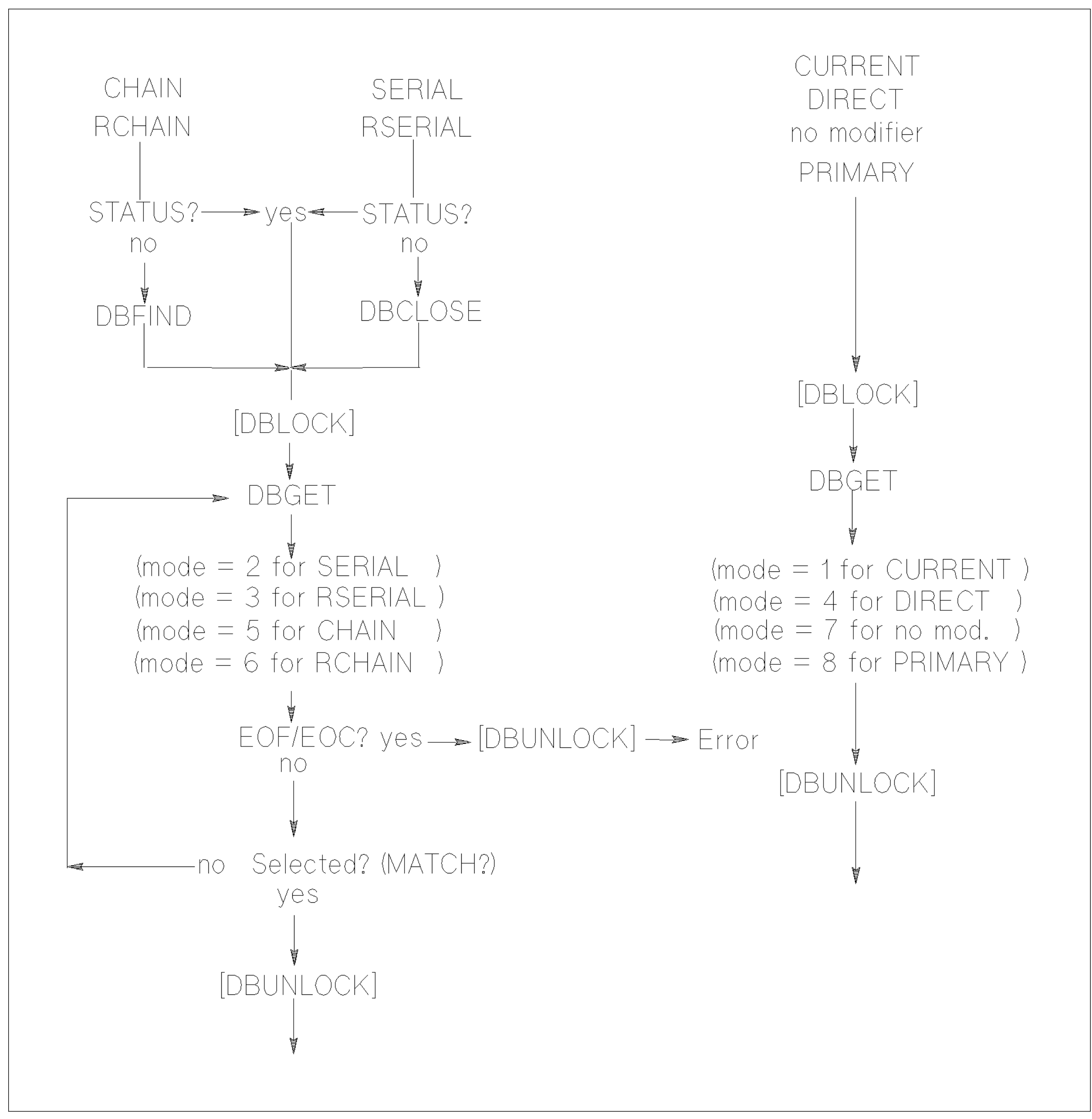


Execution of a FIND verb for an MPE file results in the following (continued):

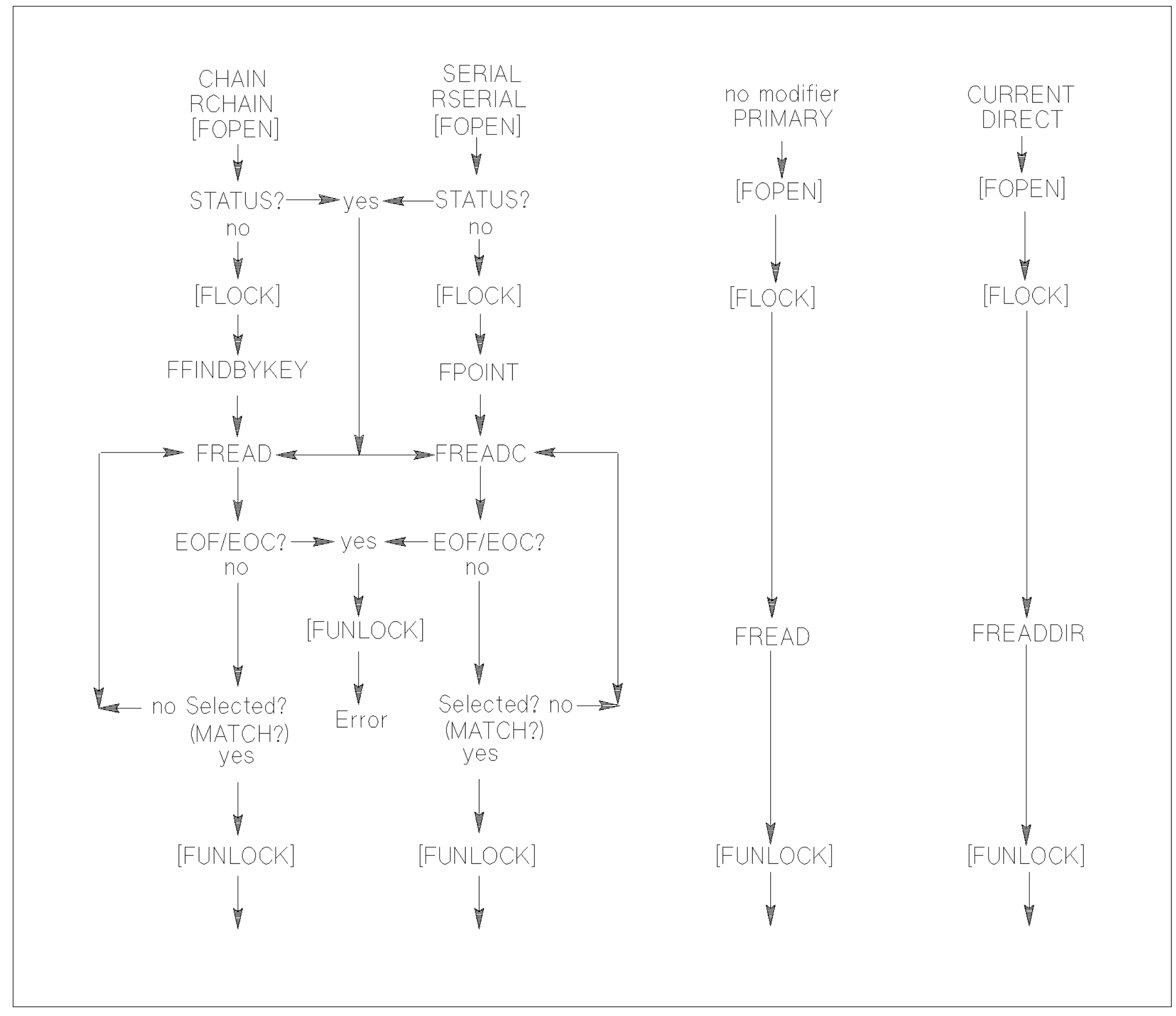


GET Charts

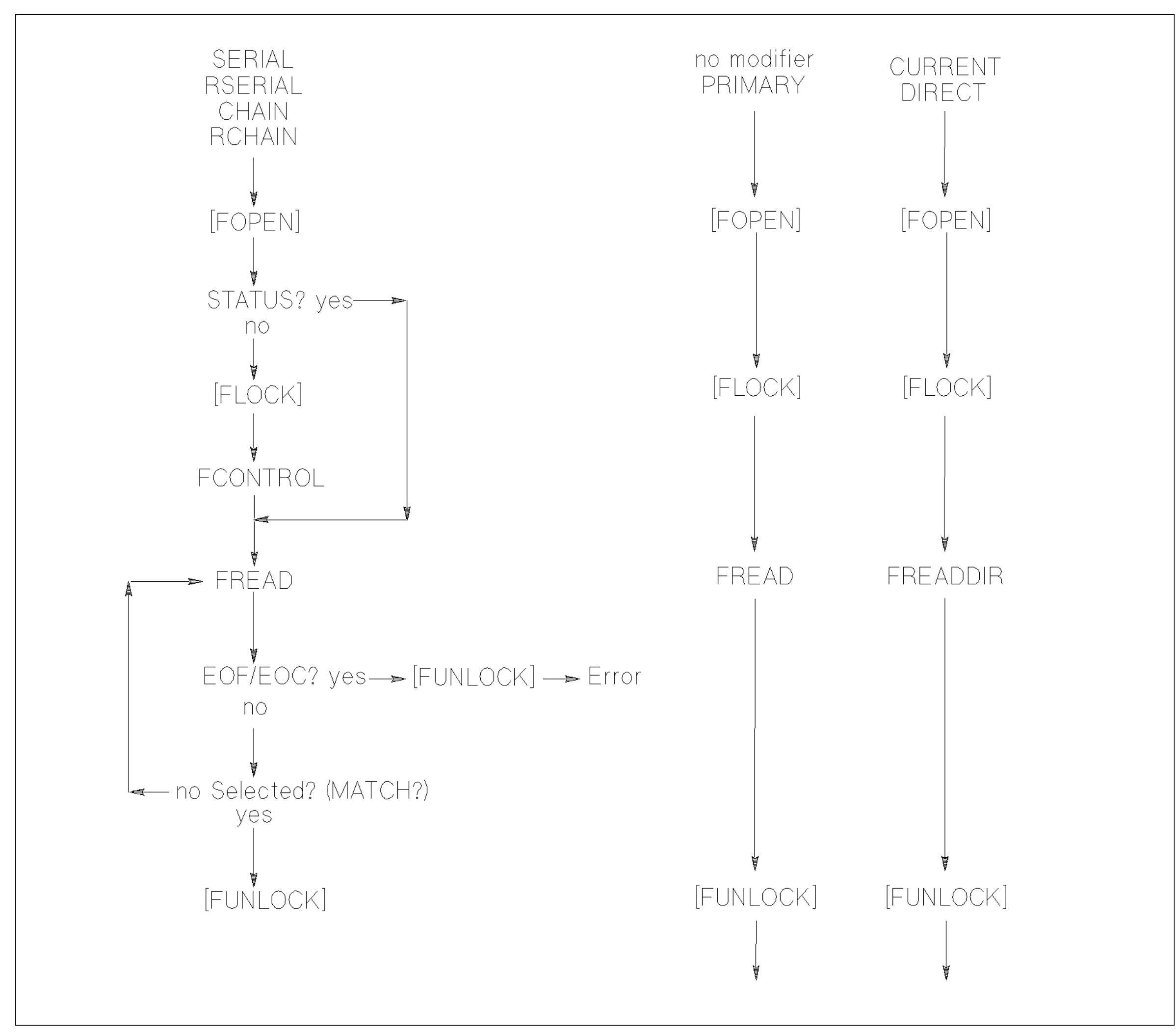
Execution of the GET verb for a TurboIMAGE data set access results in the following:



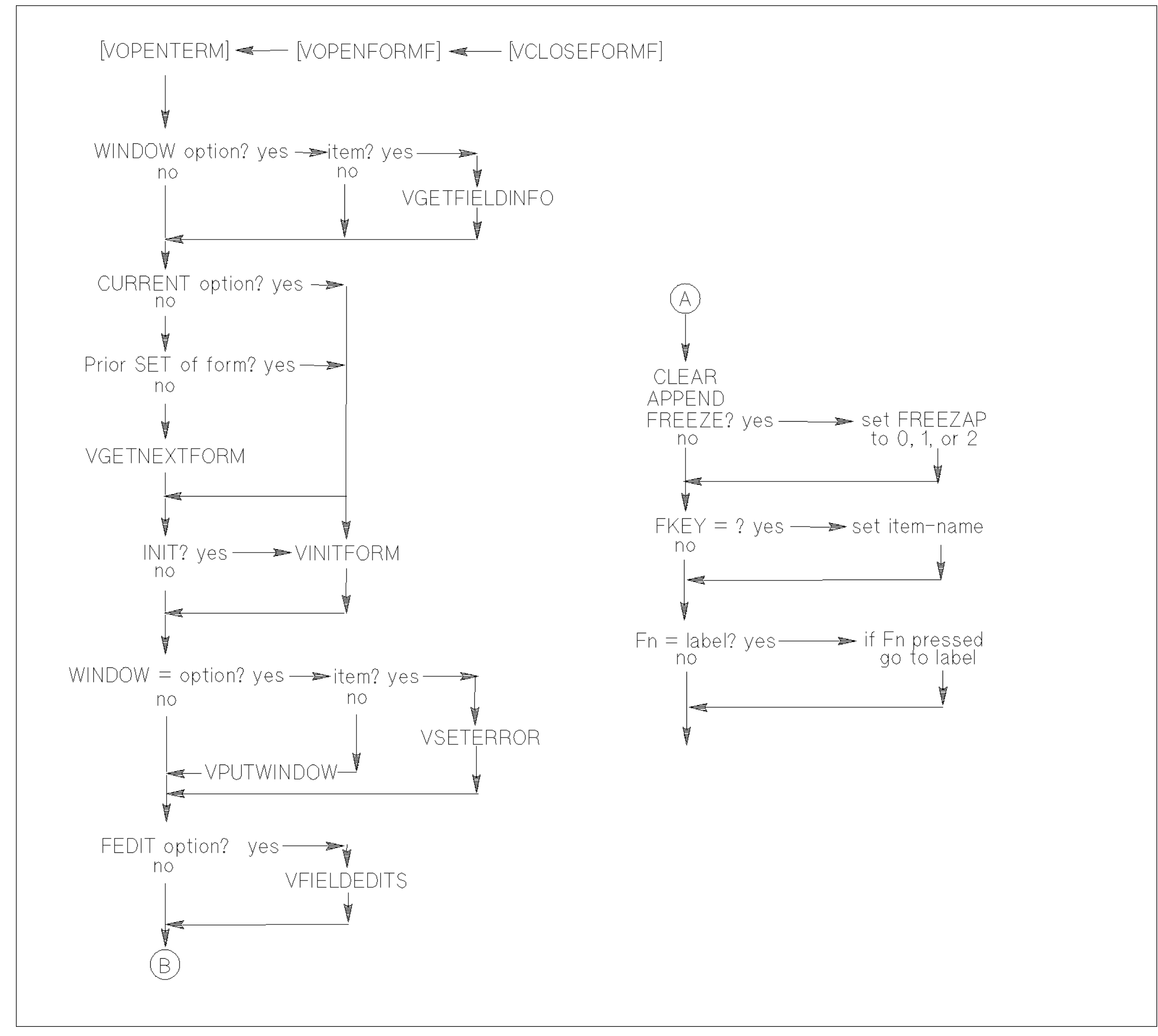
Execution of the GET verb for KSAM access results in the following:



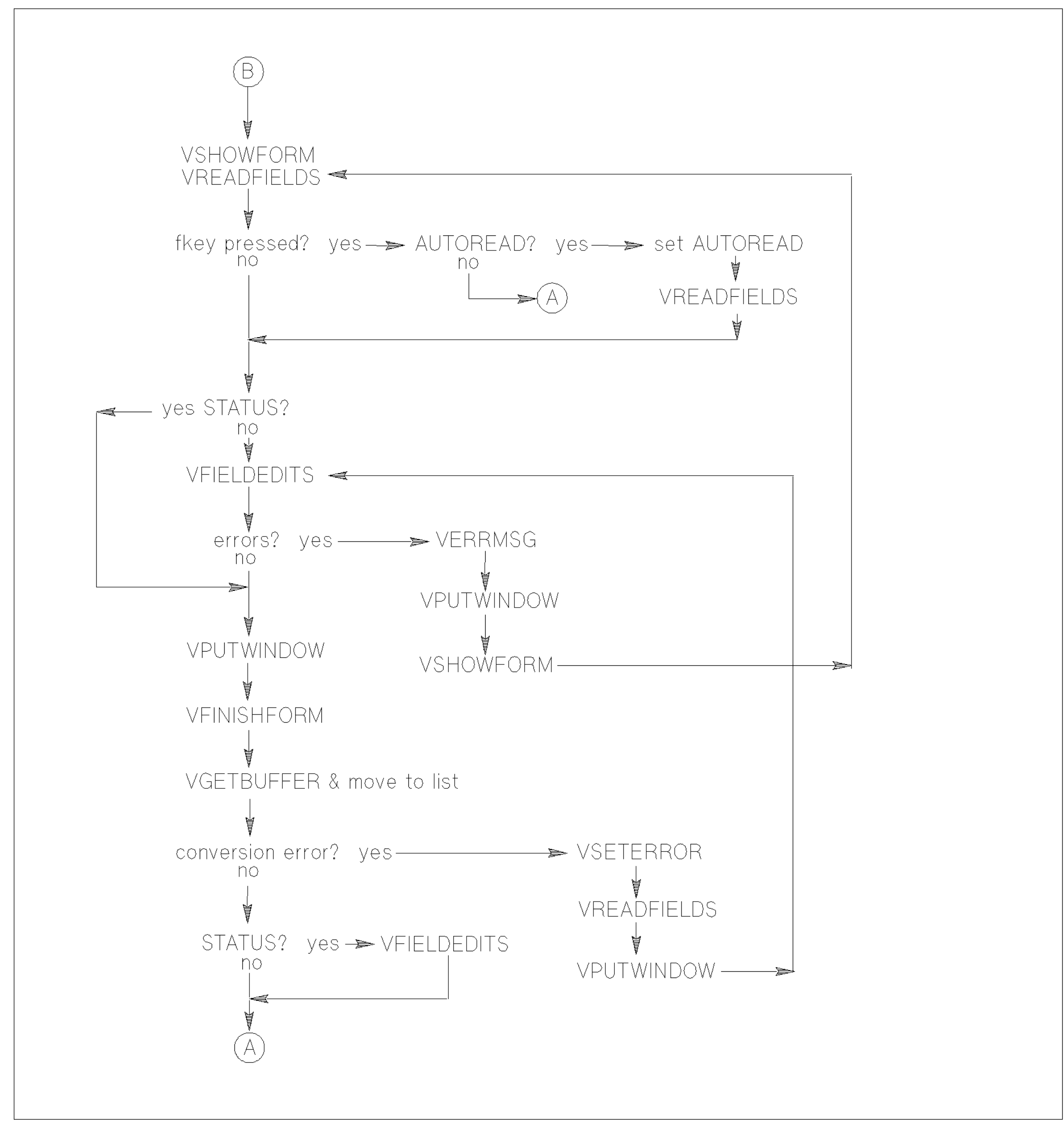
Execution of a GET verb for an MPE file results in the following:



Execution of a GET(FORM) verb for a VPLUS form results in the following:

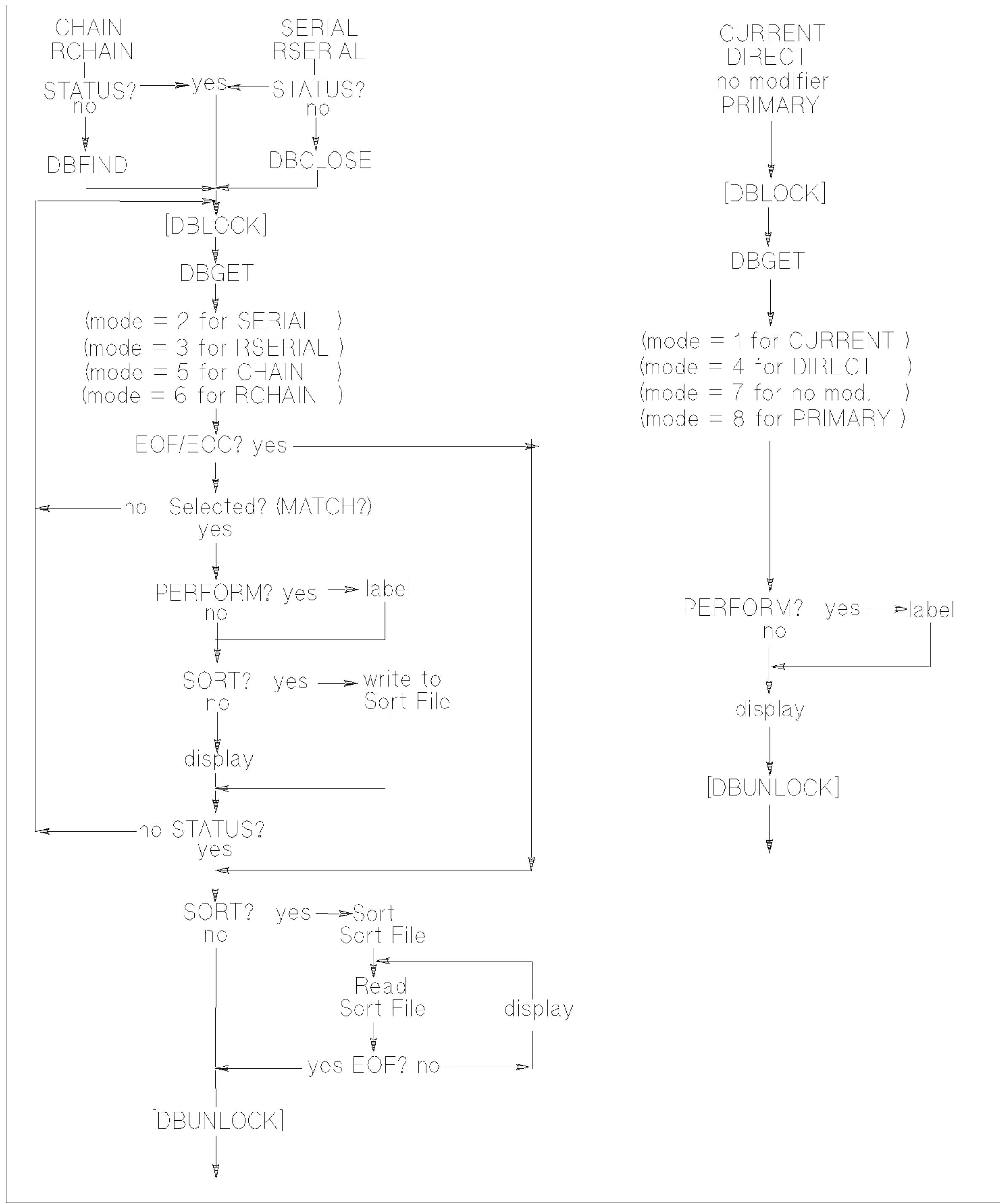


Execution of a GET(FORM) for a VPLUS form results in the following: (continued)

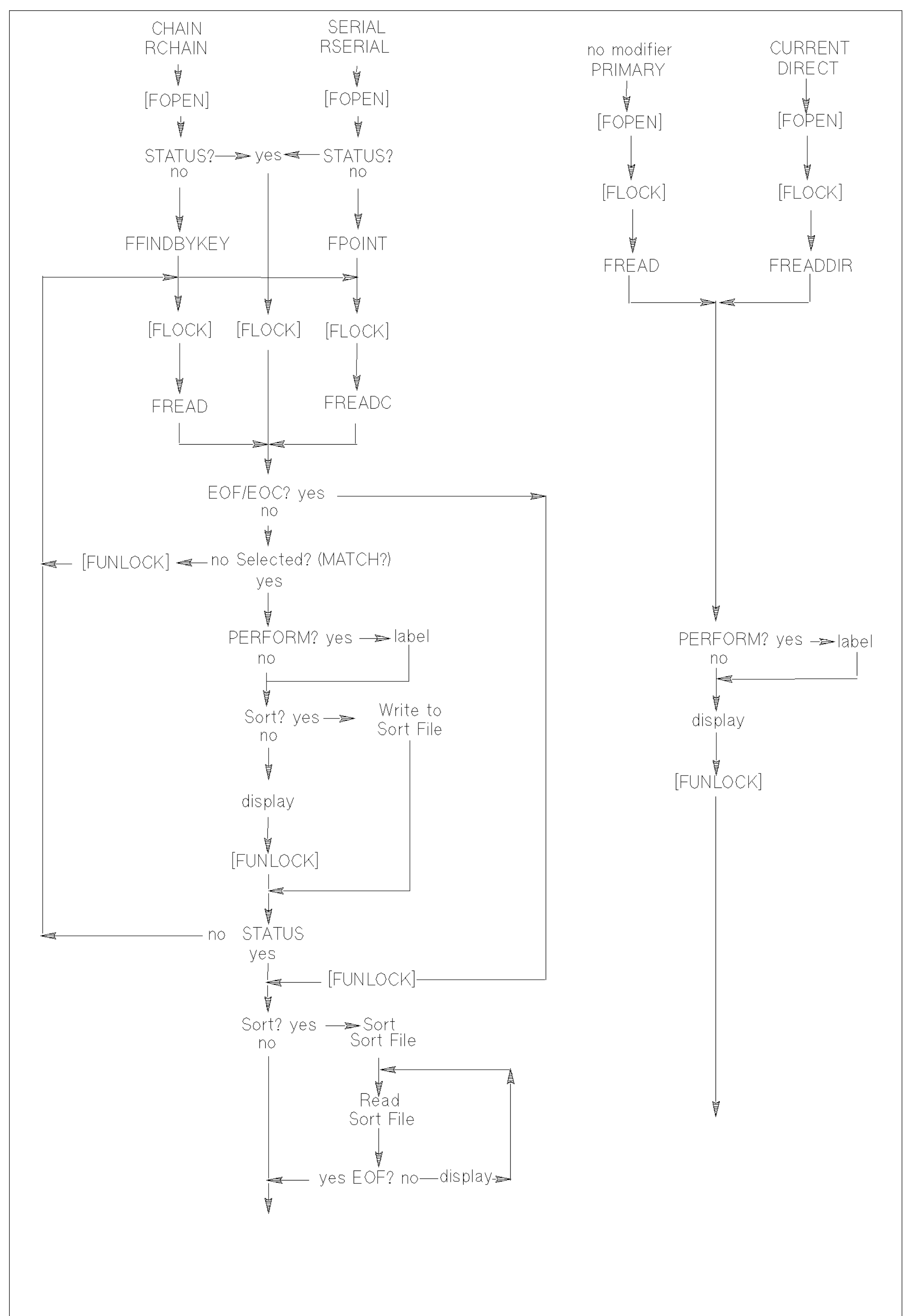


OUTPUT Charts

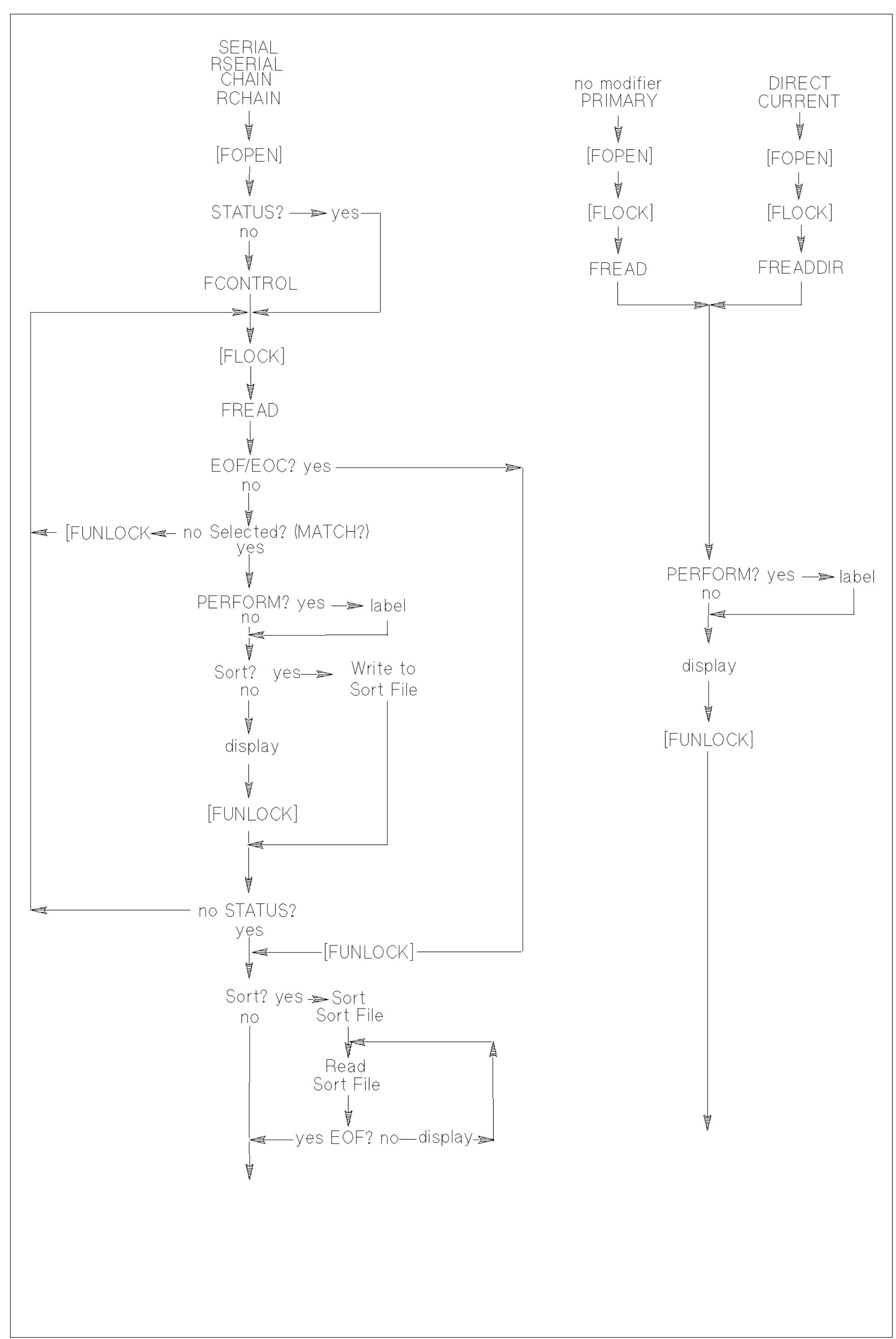
Execution of the OUTPUT verb for a TurboIMAGE data set access results in the following:



Execution of the OUTPUT verb for a KSAM file results in the following:

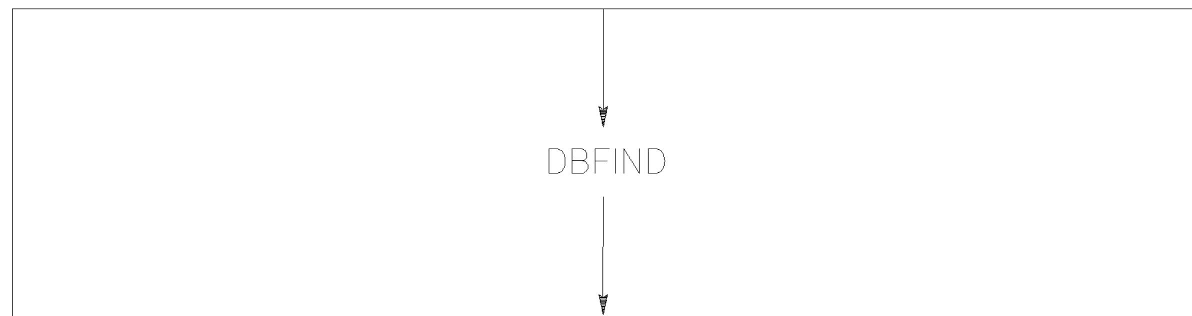


Execution of an OUTPUT verb for an MPE file results in the following:

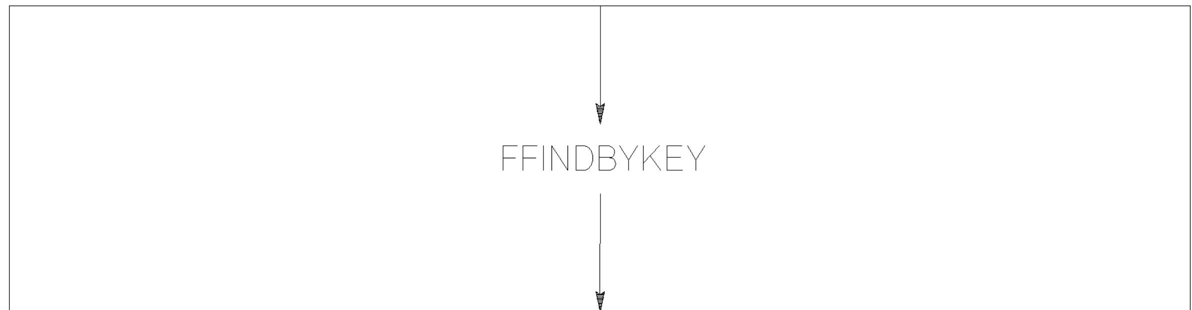


PATH Charts

Execution of a PATH verb for a TurboIMAGE data set access results in the following:

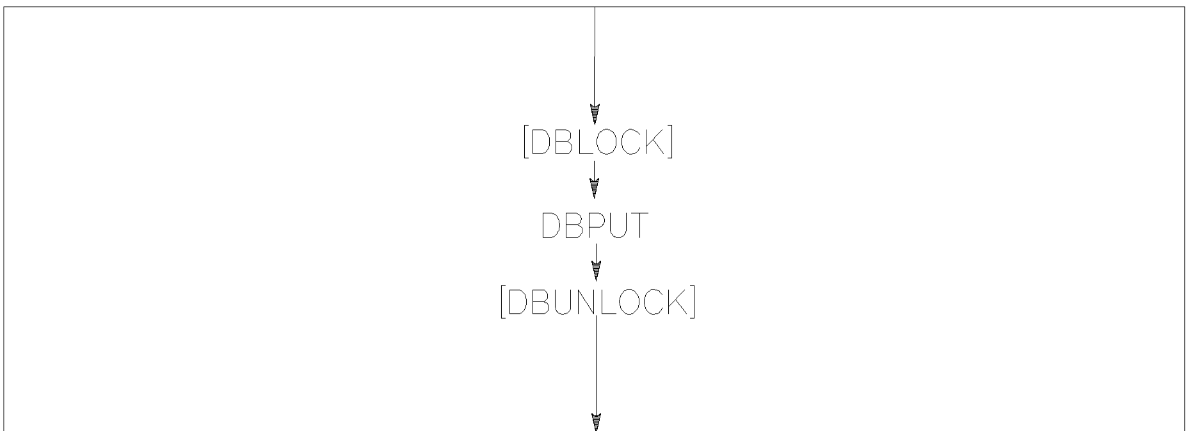


Execution of a PATH verb for a KSAM file results in the following:

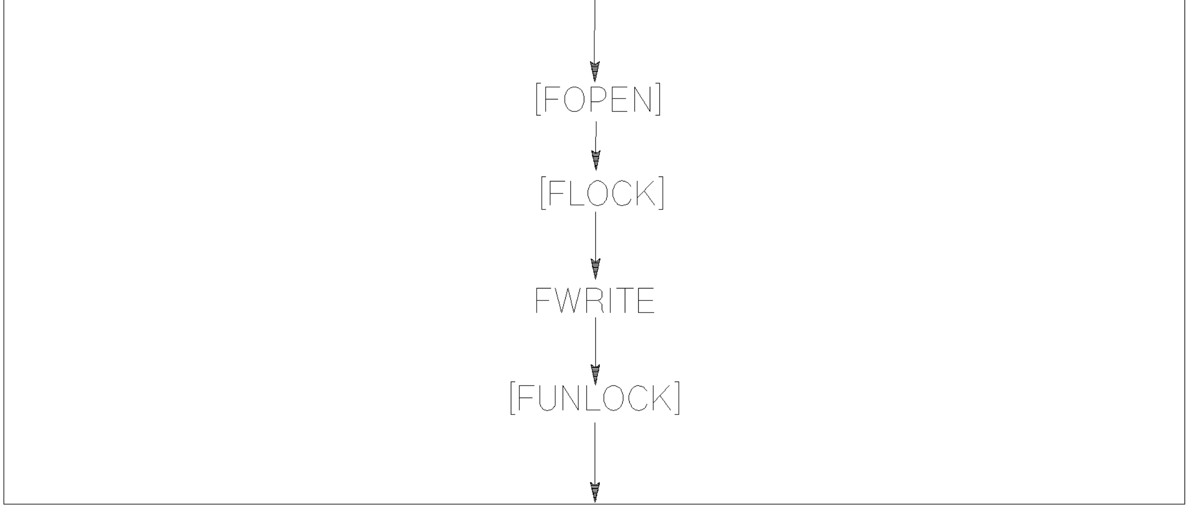


PUT Charts

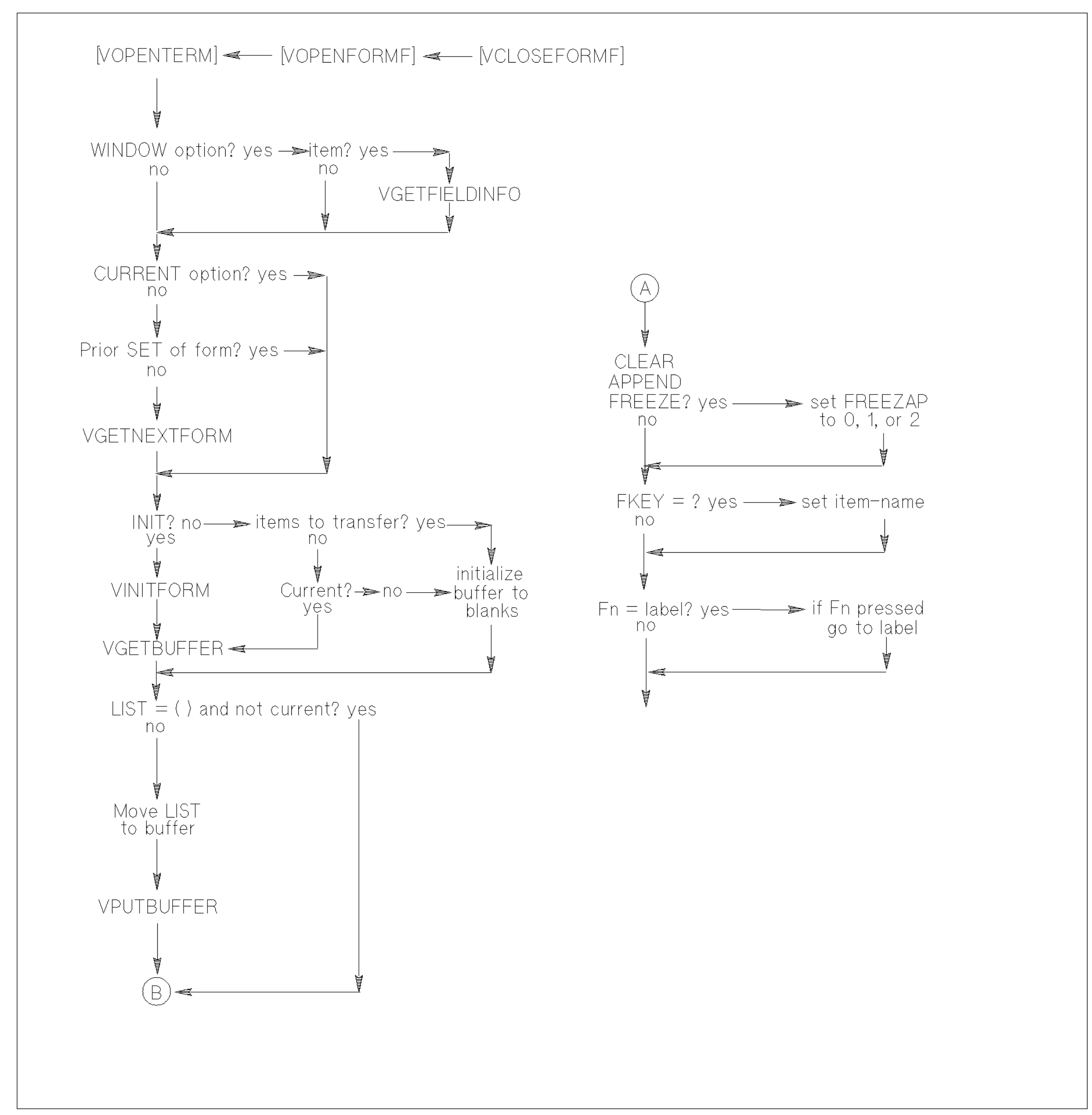
Execution of a PUT verb for a TurboIMAGE data set results in the following:



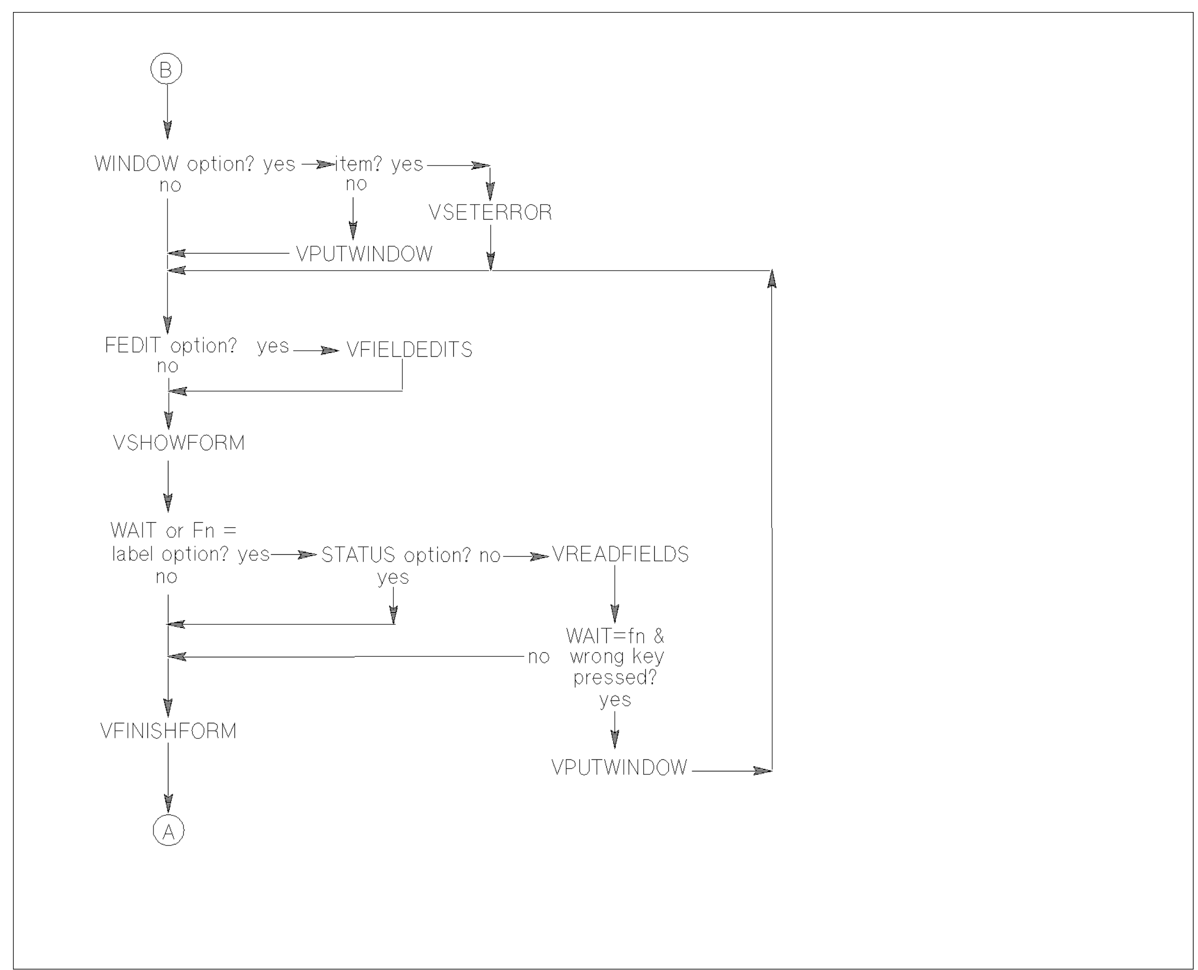
Execution of a PUT verb for a KSAM or MPE file results in the following:



Execution of a PUT(FORM) verb on a VPLUS form results in the following:

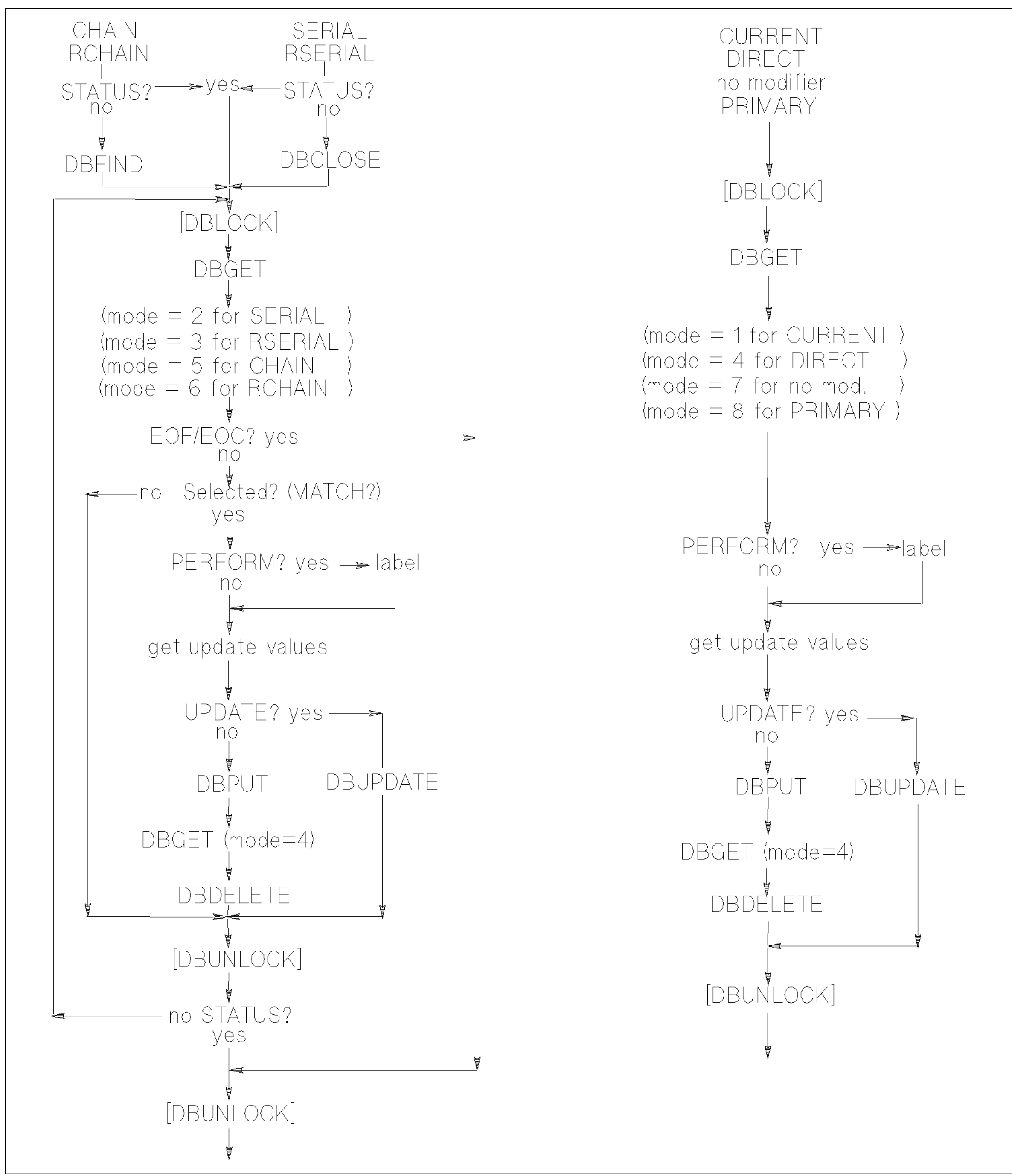


Execution of a PUT(FORM) verb on a VPLUS form results in the following: (continued)

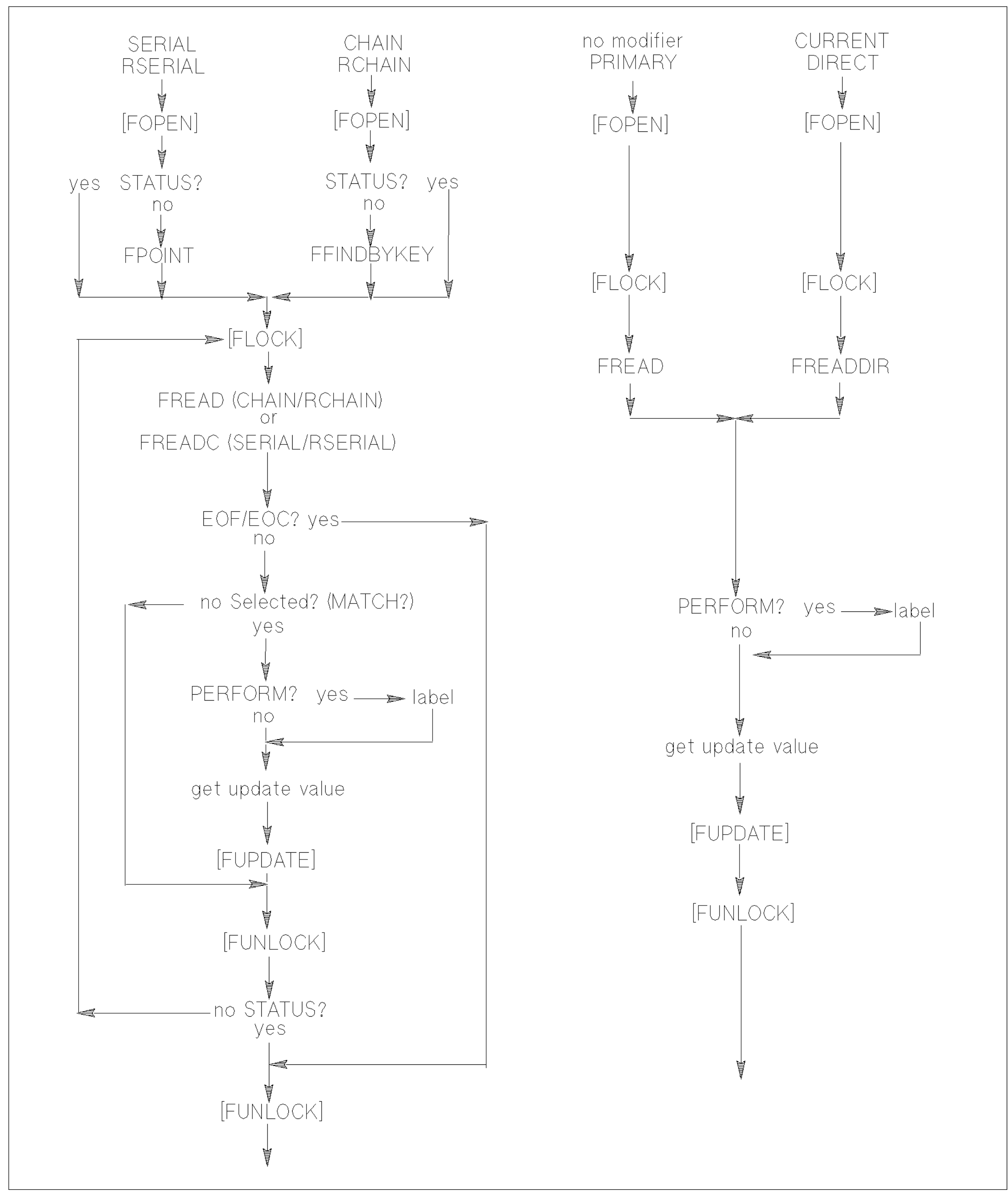


REPLACE Charts

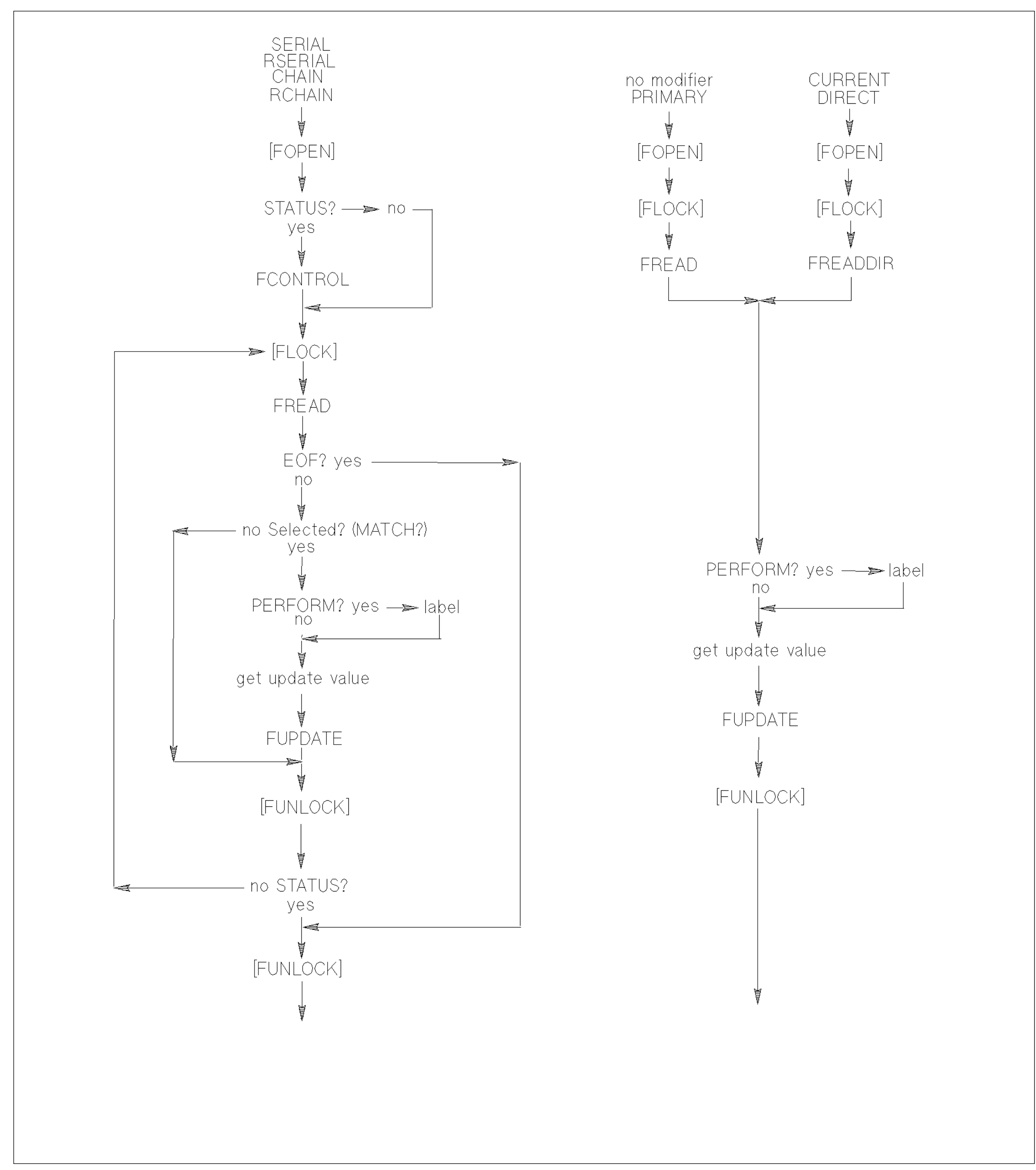
Execution of the REPLACE verb for a TurboIMAGE data set access results in the following:



Execution of a REPLACE verb for a KSAM file results in the following:

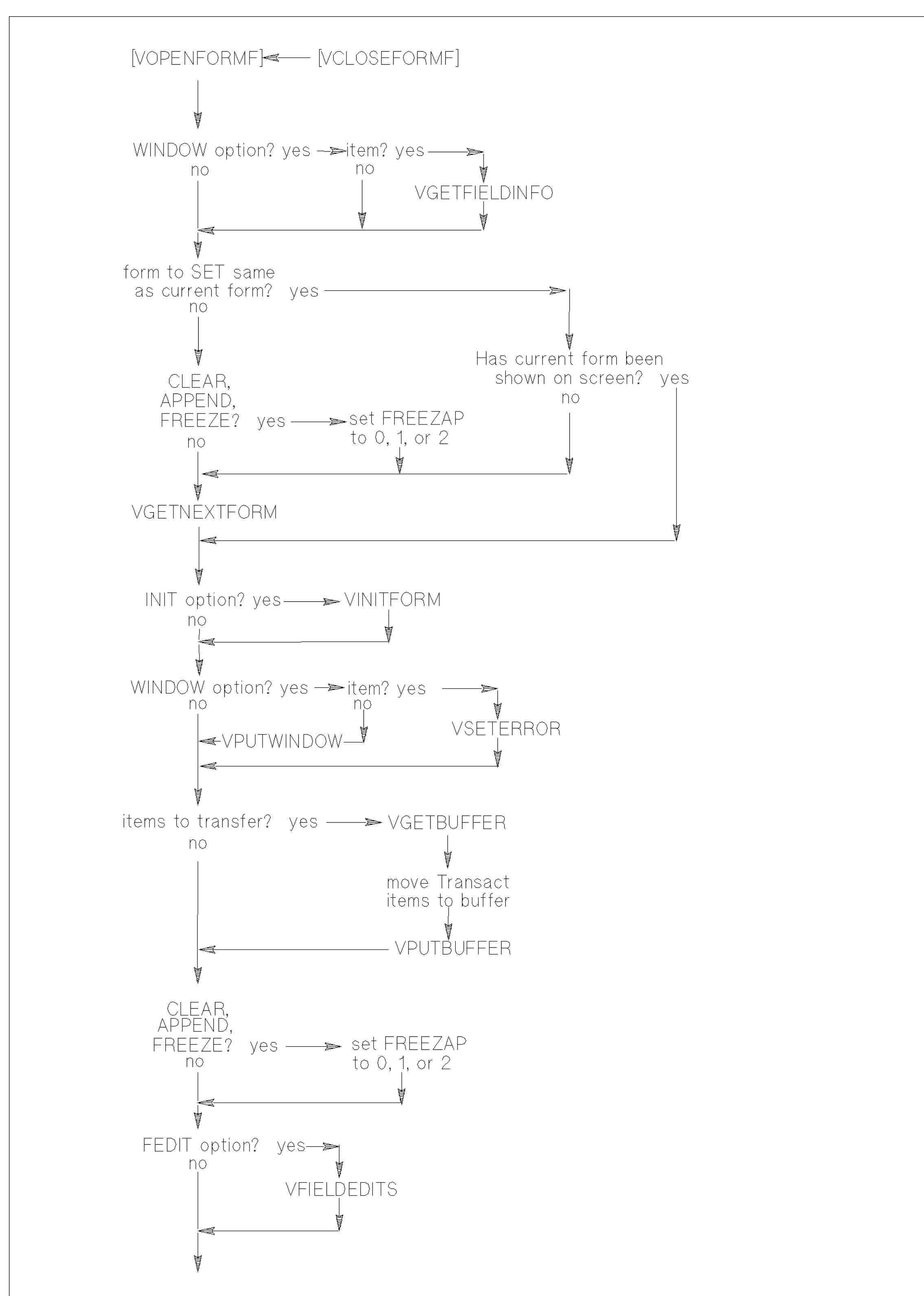


Execution of a REPLACE verb for an MPE file results in the following:



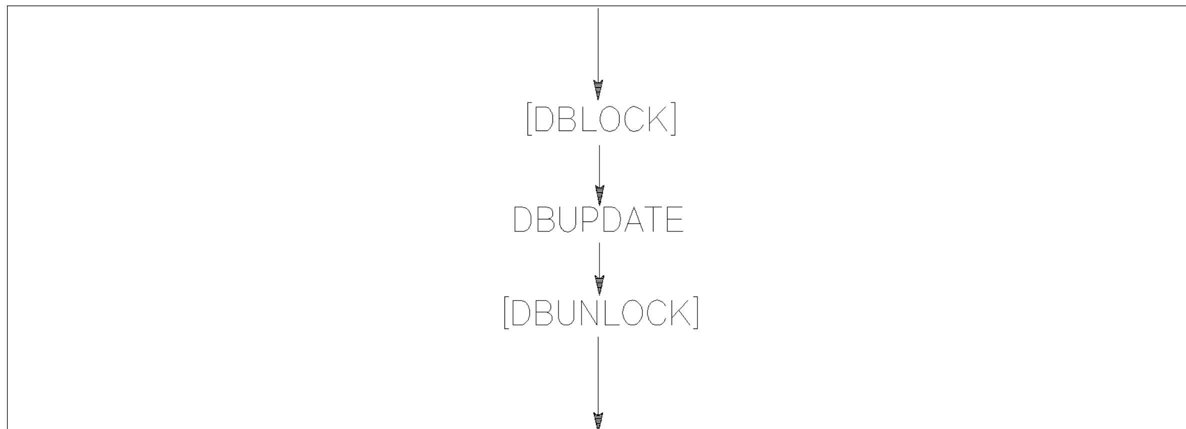
SET Charts

Execution of a SET(FORM) verb for a VPLUS form results in the flowchart on the next page:

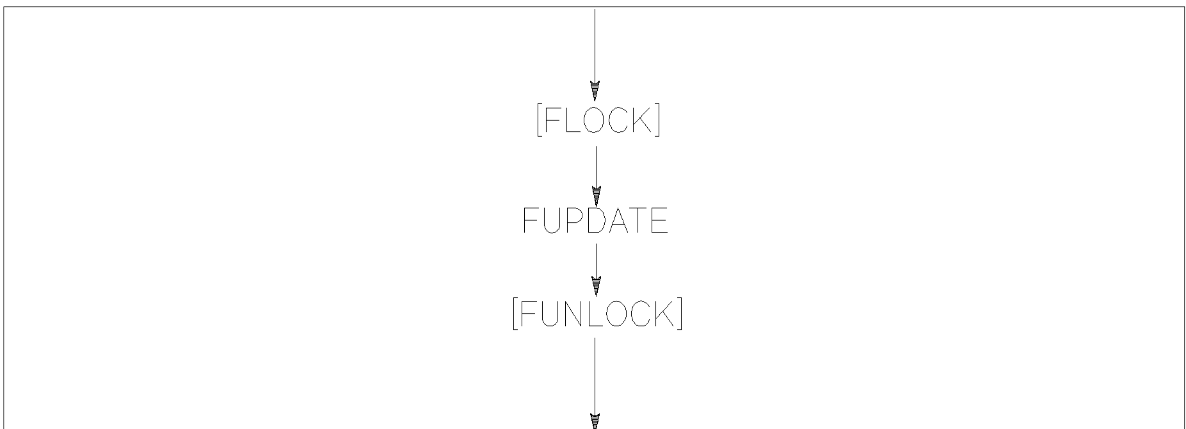

[VOPENFORMF]
[VCLOSEFORMF]
WINDOW option? yes
no
item? yes
no
VGETFIELDINFO
form to SET same
as current form? yes
no
Has current form been
shown on screen? yes
no
CLEAR,
APPEND,
FREEZE? yes
no
set FREEZAP
to 0, 1, or 2
VGETNEXTFORM
INIT option? yes
no
VINITFORM
WINDOW option? yes
no
item? yes
no
VSETERROR
VPUTWINDOW
items to transfer? yes
no
VGETBUFFER
move Transact
items to buffer
VPUTBUFFER
CLEAR,
APPEND,
FREEZE? yes
no
set FREEZAP
to 0, 1, or 2
FEDIT option? yes
no
VFIELDEDITS

UPDATE Charts

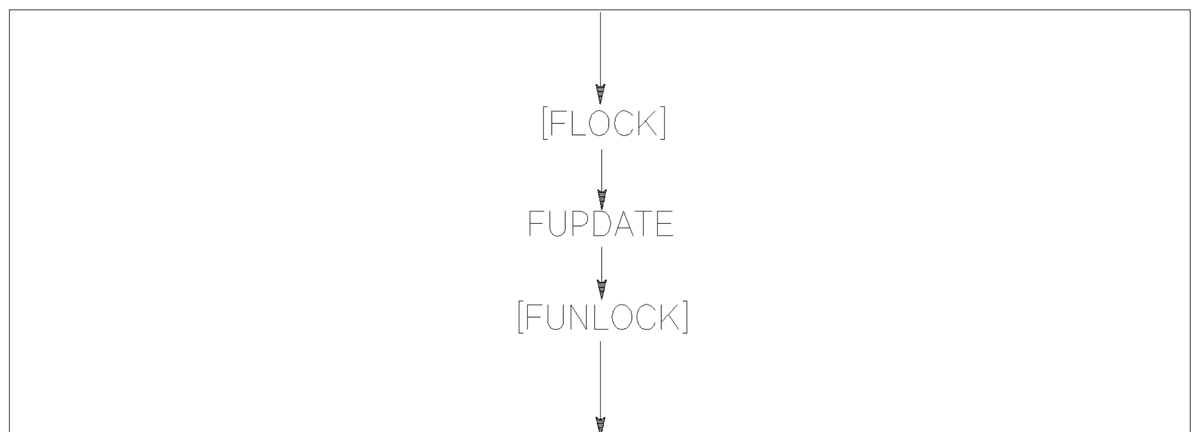
Execution of an UPDATE verb for a TurboIMAGE data set access results in the following:



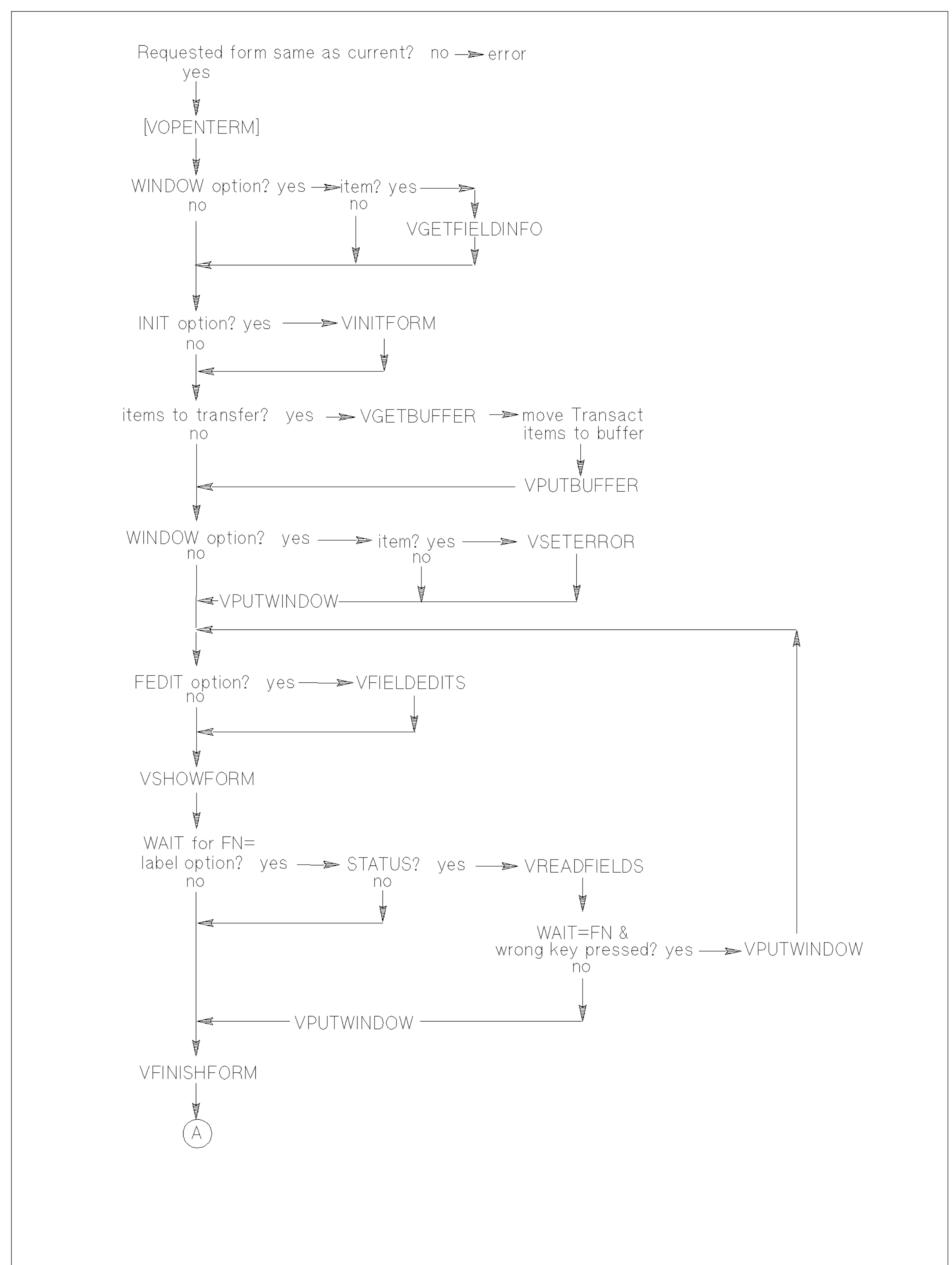
Execution of an UPDATE verb for a KSAM file results in the following:



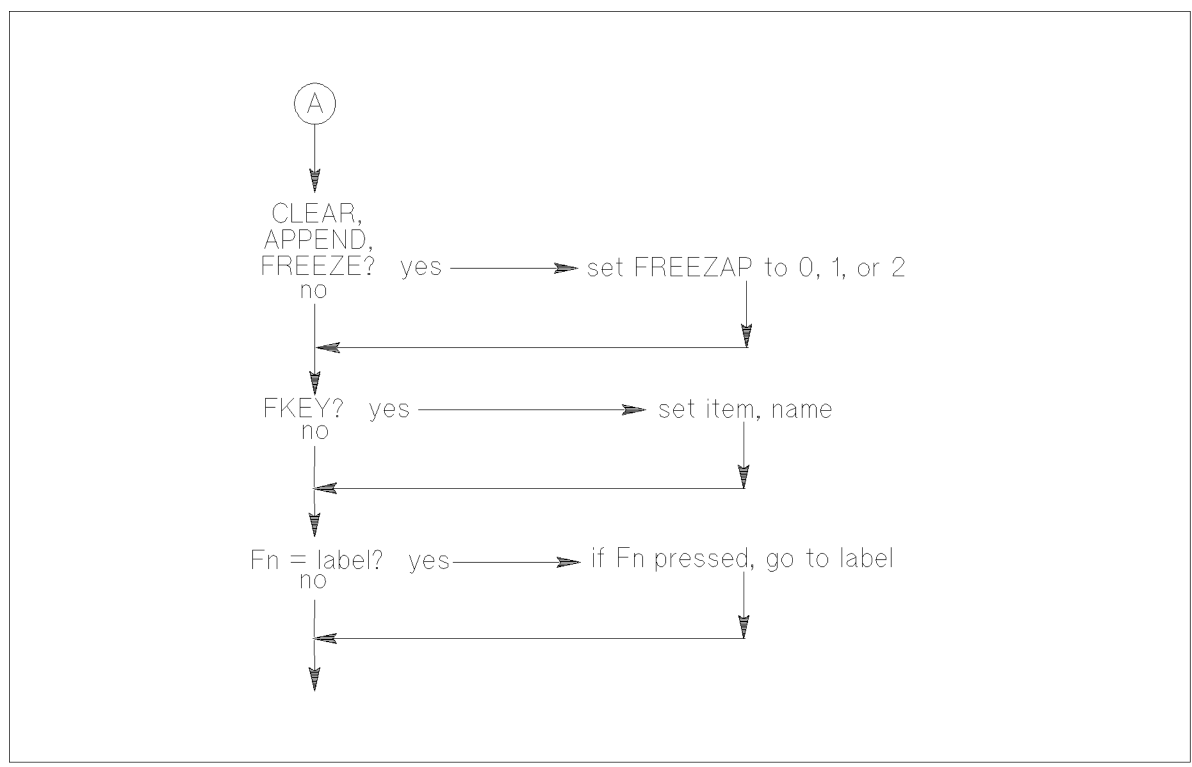
Execution of an UPDATE verb for an MPE file results in the following:



Execution of an UPDATE(FORM) verb for a VPLUS form results in the following:



Execution of an UPDATE(FORM) verb for a VPLUS form results in the following (continued):



B

Native Mode Transact/iX Migration Guide

This appendix discusses guidelines for experienced Transact programmers who want to migrate Transact/V programs to native mode Transact/iX programs on an MPE/iX system. Because minimal changes are required for migration, there is no migration utility available, and the changes must be made manually.

This appendix contains the following sections:

Exclusive Transact/iX Features	Describes the new features available in native mode Transact/iX that are not available in Transact/V.
Exclusive Transact/V Features	Describes the features available in Transact/V that are not available in Transact/iX.
Features that Differ Between Transact/V and Transact/iX	Describes the features that are available in both Transact/V and native mode Transact/iX that may have different interpretations or actions on the two systems.
Source Program Migration	Explains how to migrate source programs from Transact/V to native mode Transact/iX.
Data File Migration	Explains how to convert data files for migration.
Migration Examples	Provides examples of program conversion and data file conversion.
Migration Checklist	Provides a checklist to help you migrate programs from Transact/V to native mode Transact/iX.

The following manuals provide additional information that can help you migrate Transact/V programs to native mode Transact/iX.

Title	Part Number
Introduction to MPE XL for MPE V Programmer's Migration Guide	30367-90005
Switch Programming User's Guide	32650-90014
MPE V to MPE XL: Getting Started Self-Paced Training	30367-90002
Data Types Conversion Programmer's Guide	32650-90015
MPE Intrinsics Reference Manual	32033-90007
MPE/iX Intrinsics Reference Manual	32650-90028
Migration Process Guide	30367-90007

Exclusive Transact/iX Features

The features available in native mode Transact/iX that are not available in Transact/V are:

- Additional compiler options
- Options on the PROC verb
- IEEE and HP floating point format support
- Dynamic roll-back
- Critical item update
- TRANDEBUG symbolic debugger

Additional Compiler Options

New compiler options ease the migration of Transact/V programs to native mode Transact/iX. (See "Transact/iX Compiler Options," in Chapter 9.)

Options on the PROC Verb: Parameters Passed by Byte Address

Options available on the PROC verb can be used to specify the alignment of individual references passed by the address. These options are discussed under the PROC verb in Chapter 8.

Floating Point Formats

The code generated by the Transact/iX compiler supports both IEEE and HP floating point formats. Under native mode MPE/iX, real numbers are always stored internally in IEEE format.

Translation between IEEE and HP formats from and to files and databases occurs after the read and before the write on I/O. If Transact/iX calls a procedure written in another language, Transact/iX will pass real numbers in IEEE format. The HPFPConvert intrinsic may be used to convert the storage format in the data register if the called language expects HP3000 floating point real numbers.

If no format is specified for a file or database, IEEE real numbers are assumed. The compiler option HP3000_16 is available for defining a floating point format for all of the files and databases. (See "Transact/iX Compiler Options" in Chapter 9.) If floating point format for an individual file or database is different from that specified by the compiler option, you can express the requirements in the FILE or BASE specification of the SYSTEM statement. See the SYSTEM statement in Chapter 8 for more information.

Caution

When passing parameters or data that access real numbers via PROC or CALL, the subprogram must be compiled with the same real number format as the main program.

Dynamic Roll-Back

Transact/iX supports the TurboIMAGE dynamic roll-back feature beginning in Transact/iX version A.04.00. (See the "Dynamic Roll-Back" section in Chapter 6 and the description of the LOGTRAN verb in Chapter 8.)

Critical Item Update

Transact/iX allows TurboIMAGE database search and sort items to be updated. (See descriptions of the REPLACE and UPDATE verbs in Chapter 8.)

Symbolic Debugger

TRANDEBUG is a symbolic debugger that is included in Transact/iX to replace the test modes used in Transact/V. (See Chapter 11 for a complete description of TRANDEBUG.)

Exclusive Transact/V Features

The features of Transact/V that are not available in native mode Transact/iX consist of the following:

- MPE V-related compiler options
- Run-time data item attribute resolution
- INITIALIZE
- Calls to Transact/V subprogram
- UNLOAD and NOLOAD options on the PROC verb
- TRANIN

MPE V-Related Compiler Options

The following Transact/V compiler options are ignored by Transact/iX: DICT, CODE, OBJT, STAT, XERR, OBJO, OBJH. (See "TRANCOMP Options Available to the Transact/iX Compiler" in Chapter 9.)

Run-Time Item Attribute Resolution (Binding)

Syntax option 6 of the DEFINE statement is not allowed, because the Transact/iX compiler does not provide run-time support for Dictionary/V or for System Dictionary. Therefore, the following form of the DEFINE(ITEM) verb is not allowed:

```
DEFINE(ITEM) itemname *;
```

Any items defined in this way cause the Transact/iX compiler to display an informational message:

`*INFO: UNDEFINED ITEM:` *nnn*

where *nnn* is the name of the data item. If the compiled program runs, a run-time error occurs:

`*ERROR: UNDEFINED DATA ITEM:` *nnn*

Resolution of data item definitions from a dictionary is limited to compile time when the Transact/iX compiler uses TRANCOMP. Therefore, whenever changes are made to data definitions in the dictionary, the Transact/iX program must be recompiled for the changes to be carried forward to the Transact/iX object code files.

Test Modes

The Transact/iX compiler does not provide run-time support for the existing Transact/V test modes. Transact/iX provides the TRANDEBUG symbolic debugger instead. See Chapter 11 for a complete description of TRANDEBUG.

If you want to use the Transact/V test modes for program development and debugging, it is necessary to do these activities in compatibility mode, then move the application to native mode for production runs.

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

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.

Source Program Migration

This section describes how to convert source programs written in Transact/V to native mode Transact/iX source.

Since there are few, if any, changes to be made when converting from a Transact/V source program to a native mode Transact/iX source program, no automatic conversion utility is provided. Any changes required must be made to the Transact/V source program to convert it to a native mode Transact/iX program.

If the program accesses data from files or databases created on a MPE V based system and the data includes real numbers, the native mode Transact/iX program should be compiled using the HP3000_16 option until data conversion programs are written to convert the data to the MPE/iX standard format for real numbers.

Conversion

If your programs do not use the PROC verb, there should be few if any changes to make. See the Migration Checklist later in this appendix for more related information.

When migrating a Transact/V program on an MPE V based system to a native mode Transact/iX program on an MPE/iX based system, you should check to see if any of the missing and changed features listed in the previous sections affect the program. If there are any missing or changed features in the program you are migrating, refer to the appropriate section in this manual for information on altering your program so that it can run successfully with native mode Transact/iX.

If the program uses any of the missing compiler options—the SWAP option on the CALL verb, or the UNLOAD and NOLOAD options on the PROC verb—you can ignore the compiler informational messages because program execution is not affected.

For the commands used to compile and run Transact programs under MPE/iX, see Chapter 9.

Data File Migration

Data file migration is necessary only if real numbers exist in a data file. This section explains how to convert data files that contain real numbers previously used as input to Transact/V programs on MPE V systems for use with native mode Transact/iX programs running on MPE/iX systems.

File Format Conversion

The standard format for real numbers on MPE/iX-based systems is the IEEE format. This is different from the format for real numbers on MPE V-based systems. By default, for improved performance, native mode Transact/iX assumes that real numbers are in IEEE format.

If you want to continue to use the MPE V format for real numbers, specify the HP3000_16 option in the INFO string when the native mode Transact/iX program is compiled. This option instructs the native mode Transact/iX compiler to read and write all real numbers in the MPE V format. If this option is used, no data file conversion is necessary. The MPE V-based data file can be restored onto the MPE/iX system and used without conversion.

To migrate the data file to the MPE/iX standard format, you must write a native mode Transact/iX program that reads the file with the HP3000_16 option and writes the data to a new file with the HP3000_32 option. Native mode Transact/iX automatically converts the real numbers after reading them from the input file and before writing them to the output file.

An example program is included in the next section.

Caution

The internal representation of real numbers on MPE V is different from the IEEE format used on MPE/iX. This may cause individual values to change slightly during conversion.

Compatibility mode Transact/iX programs cannot read IEEE format data. Do not migrate data files until all programs accessing those files have been converted to native mode Transact/iX.

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 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.>>
```

Migration Checklist

The checklist in this section will help you migrate Transact/V programs on MPE V to native mode Transact/iX on MPE/iX. There should be few, if any, changes to make in migrating a program to an MPE/iX-based system if the PROC verb is not used to access system intrinsics and the CALL verb is not used to call other Transact/V programs.

1. Use the MPE STORE and RESTORE commands to transfer your Transact/V source files on MPE V by tape to the MPE/iX-based system.
2. Check each Transact/V program by answering the following questions. Does the program:
   a. Use the PROC verb to call system intrinsics (such as PROC ASCII)?
   b. Use the PROC verb to call option-variable system intrinsics (such as PROC FOPEN)?
   c. Use the PROC verb to call system intrinsics that have different types of parameters than are expected by the intrinsic?
   d. Use the PROC verb to call subroutines written in other languages?
   e. Use the CALL verb to call a Transact/V program?
   f. Access files that contain real numbers?
   g. Delay variable definitions until run time?
   h. Rely upon the INITIALIZE command to switch programs?
   i. Use the FASTRAN compiler?
3. For those questions you answered "yes", take one of the following corrective actions (these actions are keyed by letter to the questions in item 2, above):
   a. If you have PROC calls to intrinsics, compile with the new option PROCINTRINSIC. The statement DEFINE(INTRINSIC) is not recommended.
   b. If you use PROC to call option-variable intrinsics, make sure all value parameters of 32 bits or more are passed. Alternatively, make sure that only one comma is used to denote each parameter.
   c. If an intrinsic that is called expects a different type of parameter, write a routine in another language such as COBOL or Pascal to duplicate the functionality of the intrinsic or to merely call the intrinsic itself. Place the routine in an RL or XL to be resolved during linking. Replace calls to the intrinsic with calls to the new routine. Continue to pass parameters in the same way.
   d. If your program uses PROC to call a routine in another language, determine how the compiler of the subroutine generates entry names. There may be differences between MPE V based compilers and MPE/iX based compilers. If the entry name has been changed by the compiler, change the reference to it in the native mode Transact/iX source program. For example, the MPE/iX based COBOL compiler converts hyphens to underscores. The MPE V based COBOL compiler leaves hyphens unchanged.

      Also, the libraries to be searched must be named during linking or running whereas Transact/V automatically searches SLs.

e. If your program uses the CALL verb to call a Transact/V program, compile the called Transact/V program with the native mode Transact/iX compiler using the SUBPROGRAM option. Place the program in an RL or XL to be resolved during linking.

f. If your program accesses files that use real numbers, use the HP3000_16 option to continue processing the file using the Transact/V storage format, or write a data conversion program that reads the MPE V format file with the HP3000_16 option and writes to a new file with the HP3000_32 option. This conversion must not be done until all programs accessing the data file have been migrated to Transact/iX.

g. If your program delays variable definition until run time, define all variables at compile time.

h. If your program contains the INITIALIZE option or command, change user procedures to exit the program and run a second program (for instance, at the MPE XL command level).

i. If your program was compiled using FASTRAN, recompile it with Transact/V. Resolve any errors generated by Transact/V before compiling it with Transact/iX.

4. If you answered "no" to all the questions in #2, or if you made all the changes suggested in #3, compile your program and try to run it.

Note

FASTRAN is owned and developed by Performance Software Group.

C

Optimizing Transact Applications

This appendix suggests ways you can optimize the run-time efficiency of Transact/V applications that run under MPE/V. This will be of special interest to experienced Transact/V programmers who are responsible for large applications. How to fine-tune *individual* programs is a very application-dependent problem, but the guidelines presented here should help you make some of the trade-offs. The material focuses on minimizing stack space and maximizing processing speed.

The following topics are discussed to help you optimize your Transact/V applications:

- Run-time stack, including a discussion of the components that make up the stack
- Compiler statistics
- Single-segment programs
- Multiple-segment programs
- Using CALL without the SWAP option
- Using CALL with the SWAP option
- Stack usage comparison
- Processing time optimization

Run-Time Stack

The size and composition of the run-time stack vary with:

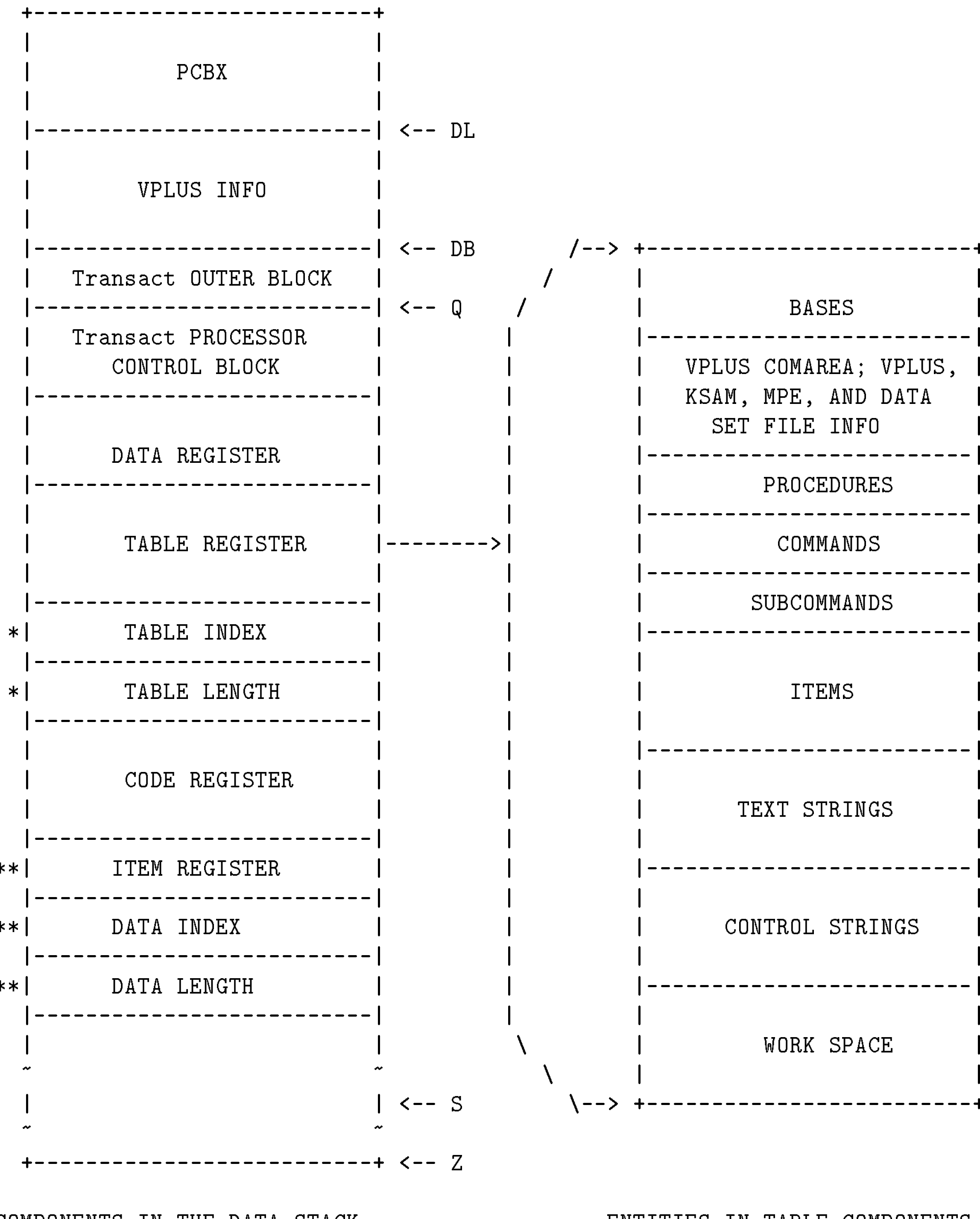
- VPLUS utilization
- Number and size of program segments
- Design of subprograms and whether swapping is used during processing
- Transact processor register utilization

Figure C-1 provides a profile of the data stack, including a breakdown of the table register components. The stack profile describes a *single-segment program*, but it illustrates components that can occur on the data stack regardless of program structure.

Use test mode 4 initially to determine the stack requirements of your program or portions of it. Use the information provided below for individual components to change their size selectively, if desired.

The data stack components are defined as follows:

- PCBX: Process Control Block Extension is a control area for MPE. The size of this area is operating-system dependent, but can be reduced slightly by running Transact with the NOCB option. Use this option to avoid stack overflow *only* on a short-term basis. In the long run, applications should be structured and optimized so that it is not necessary to use the NOCB option.
- VPLUS INFO: This area appears on the data stack if your Transact program uses VPLUS forms files. This area is used by the VPLUS subsystem. You can use fast forms files to minimize the size of this area.
- Transact OUTER BLOCK and Transact PROCESSOR CONTROL BLOCK: These areas contain data and pointers for Transact processor control. The size of these areas depends on the version and the installation.
- DATA REGISTER: The Transact data register. (See Chapter 4 for an explanation of how this register works.) The default size of this area is 1024 words. The *data-length* parameter of the DATA= option in the SYSTEM statement can be used to control the size of this area. Use test mode 3 or 102 to determine which values to specify for the DATA= option. The data register for multiple-segment programs or programs using CALLs must be large enough to accommodate all segments or subprograms. If one part of the application requires much more data register space than any other part, invoke it using MPE's process handling feature.



* Used to manage the TABLE REGISTER
** Used to manage the DATA REGISTER

Figure C-1. Data Stack Layout for a Single-Segment Transact Program

- TABLE REGISTER: This is an area used to manage files, PROC calls, built-in and programmer-defined commands, sub-commands, and data items and strings. This register's entities are identified in the right-hand diagram of Figure C-1. These entities are defined and ways to optimize them are suggested later in this section.

  VPLUS forms files significantly affect the size of the TABLE REGISTER. If an application requires many VPLUS forms, you can conserve stack space by doing any of the following:

  - using a CALL structure rather than a multiple-segment program structure.
  - specifying only forms used by the main program and each subprogram in the SYSTEM statement of the main program and each subprogram. If only a forms file name is specified in a SYSTEM statement, Transact allocates TABLE REGISTER space for each form in the file and for all items associated with each form.
- TABLE INDEX and TABLE LENGTH: These areas are used to manage the TABLE REGISTER component. These areas consist of indexes and lengths, respectively, that correspond to entities of the TABLE REGISTER.
- CODE REGISTER: An area that contains p-code data.
- ITEM REGISTER, DATA INDEX, and DATA LENGTH: Areas used to manage the DATA REGISTER component. Each of these areas has a default size of 128 words. You can use the DATA= option of the SYSTEM statement to control the size of these areas. Use test mode 3 or 102 to determine which value to specify in the *data-count* parameter of the DATA= option.
- DL, DB, Q, S, and Z: These are stack pointers. Transact requires 4K of the space between DATA LENGTH and S. The stack requirements of SORT, HP2680, and other subsystems lie between S and Z. Use test mode 4 to locate stack pointers DL, Q, S, and Z for various portions of your program.
- The TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components manage the following entities. In general, as the number of these entities used by your Transact program increases, so does the table register space required:

  - BASES: databases.
  - VPLUS COMAREA; VPLUS, KSAM, MPE, AND DATA SET FILE INFO: forms files, forms, MPE and KSAM files, and data sets.
  - PROCEDURES: calls to user procedures or system intrinsics.
  - COMMANDS: built-in commands and command qualifiers. The 11 built-in commands (for example, PRINT, SORT, REPEAT, and EXIT) require 65 words of stack space. Programmer-defined commands increase these needs.
  - SUBCOMMANDS: programmer-defined subcommands.
  - ITEMS: data items defined with the DEFINE statement or defined in the data dictionary. You can optimize this area by using the DEFINE(ITEM) statement with the OPT option as well as by compiling with OPT@, OPTE, OPTH, OPTI or OPTP compiler options. Space is allocated for *all* data item textual names and any edit masks, headings, and entry/prompt texts found in the data dictionary, unless these options are invoked. See syntax option 3 under DEFINE in Chapter 8 and OPTI in Chapter 9.
  - TEXT STRINGS: literal ASCII strings. The stack requirements increase with the number of MOVE and DISPLAY statements with literals, WINDOW= options for VPLUS, and

so on. You can optimize the TEXT STRINGS component by keeping all application messages in a message file instead of embedding them in the p-code. This practice both saves stack space and allows for easy message customization. Also consider keeping messages in forms files instead of using the WINDOW= option of the VPLUS verbs.

- CONTROL STRINGS: internal representations of DISPLAY and FORMAT statements and complex arithmetic expressions.
- WORK SPACE: work area used for sort items and match, update, input, key, and argument registers. By default, 256 words are allocated for the work space portion of the TABLE REGISTER and 64 words for the work space portions of the TABLE INDEX and TABLE LENGTH components.

  The WORK= option of the SYSTEM statement can be used to control the size of the work space areas. Run test mode 3 or 102 to determine the requirements for your program. To override the defaults, specify a *work-length* value for the work space portion of the TABLE REGISTER and a *work-count* value for the work space portions of the INDEX and TABLE LENGTH registers.

  Do not underestimate WORK SPACE requirements, because the recovery procedure invoked to re-use work spaces increases processing time. Maximize the usefulness of test mode 3 or 102 results by ensuring that all program options and branches are exercised several times.

Multiple-segment programs and programs using the CALL statement have additional data stack components:

- Multiple-segment programs use the data stack for keeping track of where the segments are located on disk and for storing segment offsets. Code registers for a root segment and the current segment are also required.
- Programs containing CALLs without the SWAP option use the data stack to control both the main program and the current subprogram.
- Programs containing CALLs that use the SWAP option require data stack components very similar to those that do not use this option, but they do not all need to be present simultaneously on the stack.

Although these three structures require additional data stack components, they require less total stack space than a single-segment program if they contain more than two segments. The data stack requirements of each structure are described in detail later in this section.

Note

Under some circumstances, the error messages "FSERR 74" or "Items not found in dictionary" may be issued when there is no stack overflow and the items do in fact exist in the dictionary. In this context, both of these errors may be due to segmentation problems. To overcome this problem in Transact/V, try compiling with the Transact/V NOCB and STACK = 2000 or use the OPT@ option. If none of these solutions work, you will have to segment your program further and try again.

Compiler Statistics

Figure C-2 shows the compiler listing that is produced when the statistics option is in effect during compilation. The format shown is for *single-segment* programs, but is virtually the same for the other three structures being examined in this appendix. The fields are defined as follows:

COMPILE TIME STATISTICS

STACK= x — The number of words the Transact compiler put on its data stack during compilation.

TABLE= x — The portion of the data stack used for table space during compilation, in words.

RUN TIME STATISTICS

PCODE= x — The number of words of p-code data in the current segment, plus each segment compiled before it.

SCODE= x — The number of words of p-code for a particular segment, main program, or subprogram.

PARTIAL TABLE REGISTER

```
   BASE= x, y  The number of words that the TABLE REGISTER, the TABLE
   FILE= x, y  INDEX, and the TABLE LENGTH components require. Refer
    SET= x, y  to Figure C-3 to map these compiler notations to the
   PROC= x, y  entities in these components. Note that the x values
 $$CMD= x, y  pertain to TABLE INDEX and TABLE LENGTH and that
   $CMD= x, y  the y values pertain to TABLE REGISTER.
   ITEM= x, y
  STRNG= x, y
  CNTRL= x, y
       ------
         x, y  The total number of words in the PARTIAL TABLE REG.
               SUMMARY.
```

```
FINAL TABLE REG. SUMMARY
```

WORK AREA=	x, y	The number of words in the WORK SPACE portions of the TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components. The x value reflects the work space in the TABLE INDEX and TABLE LENGTH components, and the y value reflects the work space in the TABLE REGISTER.
TABLE REG.=	y	The total number of words that the TABLE REGISTER occupies. This value is the sum of the y values in the PARTIAL TABLE REG. SUMMARY and the y value in WORK AREA.
TABLE INDX=	x	The total number of words that the TABLE INDEX requires. This value is the sum of the x values in the PARTIAL TABLE REG. SUMMARY and the x value in WORK AREA.
TABLE LEN.=	x	The total number of words that the TABLE LENGTH needs. This value is the same as that for TABLE INDX=.

```
 RUN TIME STACK SUMMARY

    DATA  REG.=    x  The number of words in the DATA REGISTER component.
    TABLE REG.=    x  The number of words in the TABLE REGISTER component.
    TABLE INDX=    x  The number of words in the TABLE INDEX component.
    TABLE LEN.=    x  The number of words in the TABLE LENGTH component.
    ROOT  SEG.=    x  The number of words in the CODE REGISTER component.
    ITEM  REG.=    x  The number of words in the ITEM REGISTER component.
    DATA INDEX=    x  The number of words in the DATA INDEX component.
    DATA  LEN.=    x  The number of words in the DATA LENGTH component.
              ------

                   x  The total number of words in the RUN TIME STACK SUMMARY.
```

The following differences in format occur if the program is a multiple-segment program or if it uses CALLs:

- For multiple-segment programs, the listing includes a PARTIAL TABLE REG. SUMMARY for each segment. The FINAL TABLE REG. SUMMARY and the RUN TIME STACK SUMMARY contain information that applies to the largest segment.
- For programs using the CALL statement with or without the SWAP option, the information shown in Figure C-2 is provided for the main program compilation and for each subprogram compilation.

```
*****COMPILE TIME STATISTICS****
     STACK=          x
     TABLE=          x

*******RUN TIME STATISTICS******
     PCODE=          x
     SCODE=          x

***PARTIAL TABLE REG. SUMMARY***
      BASE=    x,    y
      FILE=    x,    y
       SET=    x,    y
      PROC=    x,    y
     $$CMD=    x,    y
      $CMD=    x,    y
      ITEM=    x,    y
     STRNG=    x,    y
     CNTRL=    x,    y
          ------------
               x,    y

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    x,    y
          ------------
TABLE REG.=          y
TABLE INDX=          x
TABLE LEN.=          x

*****RUN TIME STACK SUMMARY*****
DATA  REG.=          x
TABLE REG.=          x
TABLE INDX=          x
TABLE LEN.=          x
ROOT  SEG.=          x
ITEM  REG.=          x
DATA INDEX=          x
DATA  LEN.=          x
          ------------
                     x

CODE FILE STATUS: REPLACED

0 COMPILATION ERRORS
PROCESSOR TIME=xx:xx:xx
ELAPSED TIME=xx:xx:xx
```

Figure C-2. Transact Compiler Statistics

```
  +----------------------+                      +----TABLE INDEX and TABLE
  |                      |                      |    LENGTH entities
  |        PCBX          |                      |
  |                      |                      | +--TABLE REGISTER entities
  |----------------------|                      | |
  |                      |                      | |
  |     VPLUS INFO       |                      v v
  |                      |
  |----------------------|                         +-------------------------+
  | Transact OUTER BLOCK |                         |                         |
  |----------------------|                BASE=x,y|         BASES           |
  | Transact PROCESSOR   |                         |-------------------------|
  |   CONTROL BLOCK      |                         |  VPLUS COMAREA; VPLUS,  |
  |----------------------|                FILE=x,y|  KSAM, MPE, AND DATA    |
  |                      |                 SET=x,y|    SET FILE INFO        |
* |   DATA REGISTER      |DATA REG.=x              |-------------------------|
  |----------------------|                PROC=x,y|       PROCEDURES        |
  |                      |                         |-------------------------|
  |    TABLE REGISTER    |               $$CMD=x,y|        COMMANDS         |
  |                      |TABLE REG.=x             |-------------------------|
  |----------------------|                $CMD=x,y|       SUBCOMMANDS       |
**|    TABLE INDEX       |TABLE INDX=x             |-------------------------|
  |----------------------|                         |                         |
**|    TABLE LENGTH      |TABLE LEN.=x             |          ITEMS          |
  |----------------------|                ITEM=x,y|                         |
  |                      |                         |-------------------------|
  |    CODE REGISTER     |ROOT SEG.=x              |                         |
  |                      |               STRNG=x,y|       TEXT STRINGS      |
  |----------------------|                         |                         |
 *|   ITEM REGISTER      |ITEM REG.=x              |-------------------------|
  |----------------------|                         |                         |
 *|   DATA INDEX         |DATA INDEX=x             |     CONTROL STRINGS     |
  |----------------------|               CNTRL=x,y|                         |
 *|   DATA LENGTH        |DATA LEN.=x              |-------------------------|
  |----------------------|                         |                         |
  |                      |           WORK AREA=x,y|       WORK SPACE        |**
  ~                      ~                         |                         |
  |                      |                         +-------------------------+
  ~                      ~
  +----------------------+
```

COMPONENTS IN THE DATA STACK *ENTITIES IN TABLE COMPONENTS*

* Can be changed by using the DATA= option of the SYSTEM statement
** Can be changed by using the WORK= option of the SYSTEM statement

Figure C-3. Compiler Statistics Fields and Data Stack Components

Single-Segment Programs

Single-segment programs generally execute faster than multiple-segment programs because the processor does not have to overlay information on the data stack when switching from segment to segment.

Figure C-4 shows the compiler listing produced when a single-segment program was compiled with the STAT option. Figure C-5 and Figure C-6 map the compiler statistics to individual components and entities in the run-time data stack.

```
*****COMPILE TIME STATISTICS****
     STACK=      23368
     TABLE=      14482

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=       3765

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=   12,  176
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   82, 1047
     STRNG=  195, 2192
     CNTRL=  116,  916
          ------------
             455, 4950

****FINAL TABLE REG. SUMMARY****
WORK  AREA=   30,  100
          ------------
TABLE REG.=       5050
TABLE INDX=        485
TABLE LEN.=        485

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
TABLE REG.=       5050
TABLE INDX=        485
TABLE LEN.=        485
ROOT  SEG.=       3765
ITEM  REG.=         30
DATA INDEX=         30
DATA  LEN.=         30
          ------------
                 10075

CODE FILE STATUS: REPLACED

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

Figure C-4. Compiler Statistics for a Single-Segment Program

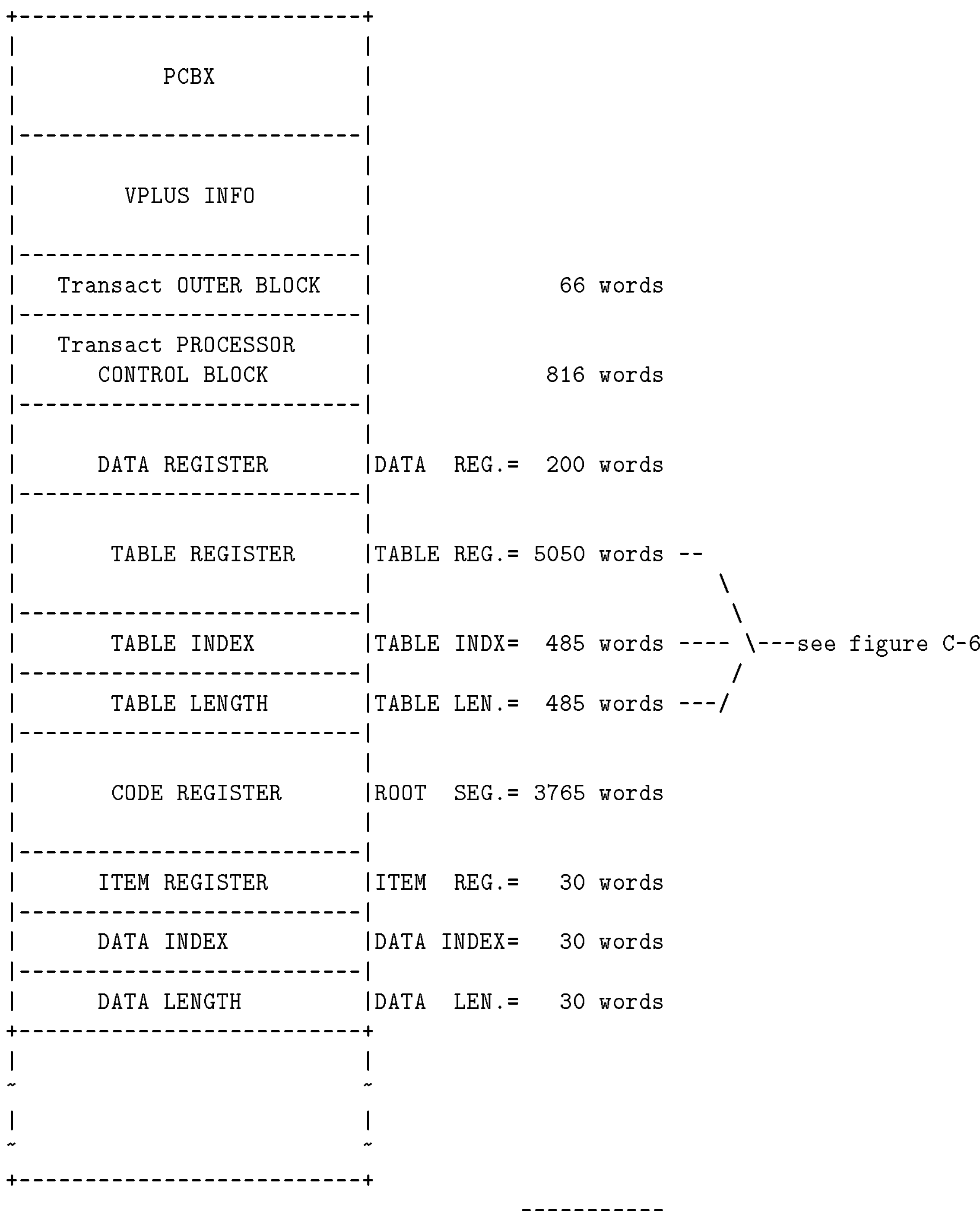


Figure C-5. Data Stack of a Single-Segment Program

```
                +-------------------------+
                |                         |
                |         BASES           |      BASE=   1,   10 words
                |                         |
                |-------------------------|
                |  VPLUS COMAREA; VPLUS,  |      FILE=  38,  544 words
                |   KSAM, MPE, AND DATA   |
                |     SET FILE INFO       |       SET=  12,  176 words
                |-------------------------|
                |       PROCEDURES        |      PROC=   0,    0 words
                |-------------------------|
                |        COMMANDS         |     $$CMD=  11,   65 words
                |-------------------------|
                |       SUBCOMMANDS       |      $CMD=   0,    0 words
                |-------------------------|
                |                         |
                |          ITEMS          |      ITEM=  82, 1047 words
                |                         |
                |-------------------------|
                |                         |
                |       TEXT STRINGS      |     STRNG= 195, 2192 words
                |                         |
                |-------------------------|
                |                         |
                |     CONTROL STRINGS     |     CNTRL= 116,  916 words
                |                         |
                |-------------------------|
                |                         |
                |       WORK SPACE        | WORK AREA=  30,  100 words
                |                         |
                |                         |             ^     ^
                +-------------------------+             |     |
                                                        |     |
TABLE INDEX and TABLE LENGTH entities--------------------+     |
                                                              |
               TABLE REGISTER entities------------------------+
```

Figure C-6. Table Register Entities of a Single-Segment Program

Multiple-Segment Programs

You can optimize your data stack requirements by segmenting your Transact program. The root segment and a current segment are always represented on the data stack. The savings in data stack space is approximately equal to the size of the segments not loaded. Although some processor time is required to overlay segments onto the data stack as they are required, the efficiency gained by decreasing the size of the data stack can be significant. Keeping applications functionally divided into segments minimizes segment switching.

The compiler listing for a multiple-segment version of the single-segment program discussed earlier is shown in Figure C-7. The program consists of four segments, each of which has compiler statistics in the following categories:

```
COMPILE TIME STATISTICS
RUN TIME STATISTICS
PARTIAL TABLE REG. SUMMARY
```

The `FINAL TABLE REG. SUMMARY` and the `RUN TIME STACK SUMMARY` reflect information for the largest segment, in this case segment 4. The following fields in the `RUN TIME STACK SUMMARY` are of special note:

```
SEG. TABLE=     Areas of the data stack used to keep track of where
XFER TABLE=     segments are located. The number of words required for
                SEG. TABLE= is version-dependent. XFER TABLE= contains
                2 words for each label defined with a DEFINE(ENTRY)
                statement.

ROOT SEG.=      The number of words that the code register requires
                for the root segment. Keep this segment as small
                as possible, since it is always memory-resident.

SCODE REG.=     The number of words that the code register requires
                for the largest segment.
```

Because the largest segment influences the number of words allocated for the data stack, try to make your segments as uniform in size as possible.

Figure C-8 and Figure C-9 show how the compiler statistics map to the run-time data stack.

```
SEGMENT 0 STATISTICS:
     STACK=      11210
     TABLE=       3138

*******RUN TIME STATISTICS******
     PCODE=         46
     SCODE=         46

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=    0,    0
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   54,  675
     STRNG=   28,  154
     CNTRL=   67,  303
          ------------
             188, 1686

COMPILED SEGMENT 0

SEGMENT 1 STATISTICS:
     STACK=      12683
     TABLE=       4624

*******RUN TIME STATISTICS******
     PCODE=        632
     SCODE=        586

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=    6,   87
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   54,  675
     STRNG=   42,  477
     CNTRL=   67,  303
          ------------
             208, 2096

COMPILED SEGMENT 1
```

Figure C-7. Compiler Statistics for a Multiple-Segment Program (1 of 3)

```
 SEGMENT 2 STATISTICS:
     STACK=      15623
     TABLE=       6419

*******RUN TIME STATISTICS******
     PCODE=       1676
     SCODE=       1044

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=    7,  103
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   61,  769
     STRNG=   76,  612
     CNTRL=   67,  303
          ------------
             250, 2341

COMPILED SEGMENT 2

SEGMENT 3 STATISTICS:
     STACK=      15644
     TABLE=       7392

*******RUN TIME STATISTICS******
     PCODE=       3123
     SCODE=       1447

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=   12,  176
      PROC=    0,    0
     $$CMD=    0,    0
      $CMD=    0,    0
      ITEM=   55,  685
     STRNG=   77,  869
     CNTRL=   68,  308
          ------------
             251, 2592

COMPILED SEGMENT 3
```

Figure C-7. Compiler Statistics for a Multiple-Segment Program (2 of 3)

```
SEGMENT 4 STATISTICS:
     STACK=      15738
     TABLE=       7134

*******RUN TIME STATISTICS******
     PCODE=       3773
     SCODE=        650

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   38,  544
       SET=   12,  176
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   74,  943
     STRNG=  110,  865
     CNTRL=  115,  911
          ------------
             361, 3514
COMPILED SEGMENT 4

****FINAL TABLE REG. SUMMARY****
WORK  AREA=   30,  100
          ------------
TABLE REG.=       3614
TABLE INDX=        391
TABLE LEN.=        391

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
SEG. TABLE=        128
TABLE REG.=       3614
TABLE INDX=        391
TABLE LEN.=        391
ROOT  SEG.=         46
XFER TABLE=          8
SCODE REG.=       1447
ITEM  REG.=         30
DATA INDEX=         30
DATA  LEN.=         30
          ------------
                  6315
CODE FILE STATUS: REPLACED
0 COMPILATION ERRORS
PROCESSOR TIME=00:01:41
```

Figure C-7. Compiler Statistics for a Multiple-Segment Program (3 of 3)

```
|--------------------------|
|          PCBX            |
|                          |
|--------------------------|
|                          |
|        VPLUS INFO        |
|--------------------------|
|  Transact OUTER BLOCK    |              66 words
|--------------------------|
|   Transact PROCESSOR     |
|      CONTROL BLOCK       |             816 words
|--------------------------|
|                          |
|      DATA REGISTER       |DATA  REG.=    200 words
|--------------------------|
|  DISC ADDRESS SEG. TABLE |SEG. TABLE=    128 words
|--------------------------|
|                          |
|     TABLE REGISTER       |TABLE REG.=   3614 words --\
|                          |                            \
|--------------------------|                             \
|       TABLE INDEX        |TABLE INDX=    391 words -----\--see Figure C-9
|--------------------------|                              /
|       TABLE LENGTH       |TABLE LEN.=    391 words ----/
|--------------------------|
|       CODE REGISTER      |
|      (Root Segment)      |ROOT  SEG.=     46 words
|--------------------------|
|      TRANSFER TABLE      |XFER TABLE=      8 words
|--------------------------|
|       CODE REGISTER      |
|  (SCODE - Overlay Area)  |SCODE REG.=   1447 words
|--------------------------|
|      ITEM REGISTER       |ITEM  REG.=     30 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=     30 words
|--------------------------|
|      DATA LENGTH         |DATA  LEN.=     30 words
|--------------------------|
~                          ~
|                          |
~                          ~
+--------------------------+
                                        ------------
              Approx. total data stack =  7197 words
```

Figure C-8. Data Stack of a Multiple-Segment Program

```
                    |-------------------------|
                    |                         |
                    |          BASES          |      BASE=   1,   10 words
                    |                         |
                    |-------------------------|
                    |  VPLUS COMAREA; VPLUS,  |      FILE=  38,  544 words
                    |   KSAM, MPE, AND DATA   |
                    |     SET FILE INFO       |       SET=  12,  176 words
                    |-------------------------|
                    |       PROCEDURES        |      PROC=   0,    0 words
                    |-------------------------|
                    |        COMMANDS         |     $$CMD=  11,   65 words
                    |-------------------------|
                    |       SUBCOMMANDS       |      $CMD=   0,    0 words
                    |-------------------------|
                    |                         |
                    |         ITEMS           |      ITEM=  74,  943 words
                    |                         |
                    |-------------------------|
                    |                         |
                    |      TEXT STRINGS       |     STRNG= 110,  865 words
                    |                         |
                    |-------------------------|
                    |                         |
                    |    CONTROL STRINGS      |     CNTRL= 115,  911 words
                    |                         |
                    |-------------------------|
                    |                         |
                    |      WORK SPACE         | WORK AREA=  30,  100 words
                    |                         |
                    |                         |             ^     ^
                    |-------------------------|             |     |
                                                            |     |
  TABLE INDEX and TABLE LENGTH entities---------------------+     |
                                                                  |
                 TABLE REGISTER entities--------------------------+
```

Figure C-9. Table Register Entities of a Multiple-Segment Program

Programs Using CALLs Without the SWAP Option

Splitting Transact programs into subprograms also decreases stack requirements.

Figure C-10 shows the compiler statistics for the program used for the earlier examples, restructured into a main program and four subprograms. The main program statistics appear on the first page and statistics for the subprograms appear on the subsequent four pages of the listing.

Figure C-11 shows the layout of the run-time data stack. Note that the top half of the stack, used by the main program, has the same components as the single-segment program data stack. The PROCESSOR PROC. VAR. area holds processor variables for calling subprograms; the size of this area is version-dependent. The next area is a second Transact PROCESSOR CONTROL BLOCK. The remaining areas are used by entities of the CALLed subprograms.

Figure C-12 portrays the entities in the TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components for the main program.

main program

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

*****COMPILE TIME STATISTICS****
     STACK=      11208
     TABLE=        962

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=         54

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=    2,   83
       SET=    0,    0
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=    1,    9
     STRNG=   10,   67
     CNTRL=    2,    5
          ------------
              27,  239

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    5,   50
          ------------
TABLE REG.=        289
TABLE INDX=         32
TABLE LEN.=         32

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
TABLE REG.=        289
TABLE INDX=         32
TABLE LEN.=         32
ROOT  SEG.=         54
ITEM  REG.=         25
DATA INDEX=         25
DATA  LEN.=         25
          ------------
                   682

CODE FILE STATUS: REPLACED
```

Figure C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (1 of 5)

subprogram 1

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

*****COMPILE TIME STATISTICS****
     STACK=      11208
     TABLE=       3262

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=        590

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=    8,  163
       SET=    6,   87
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   28,  348
     STRNG=   33,  413
     CNTRL=   25,  159
         -------------
             112, 1245

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    5,   50
         -------------
TABLE REG.=       1295
TABLE INDX=        117
TABLE LEN.=        117

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        100
TABLE REG.=       1295
TABLE INDX=        117
TABLE LEN.=        117
ROOT  SEG.=        590
ITEM  REG.=         20
DATA INDEX=         20
DATA  LEN.=         20
          ------------
                  2279

CODE FILE STATUS: REPLACED
```

Figure C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (2 of 5)

subprogram 2

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

*****COMPILE TIME STATISTICS****
     STACK=      11208
     TABLE=       4125

*******RUN TIME STATISTICS******
    PCODE=          0
    SCODE=       1101

***PARTIAL TABLE REG. SUMMARY***
    BASE=    1,   10
    FILE=   11,  190
     SET=    6,   87
    PROC=    0,    0
   $$CMD=   11,   65
    $CMD=    0,    0
    ITEM=   19,  240
   STRNG=   54,  468
   CNTRL=   13,   45
        ------------
            115, 1105

 ****FINAL TABLE REG. SUMMARY****
WORK  AREA=    8,   60
           ------------
TABLE REG.=       1165
TABLE INDX=        123
TABLE LEN.=        123

  *****RUN TIME STACK SUMMARY*****
DATA  REG.=        100
TABLE REG.=       1165
TABLE INDX=        123
TABLE LEN.=        123
ROOT  SEG.=       1101
ITEM  REG.=         20
DATA INDEX=         20
DATA  LEN.=         20
          ------------
                  2672

 CODE FILE STATUS: REPLACED
```

Figure C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (3 of 5)

subprogram 3

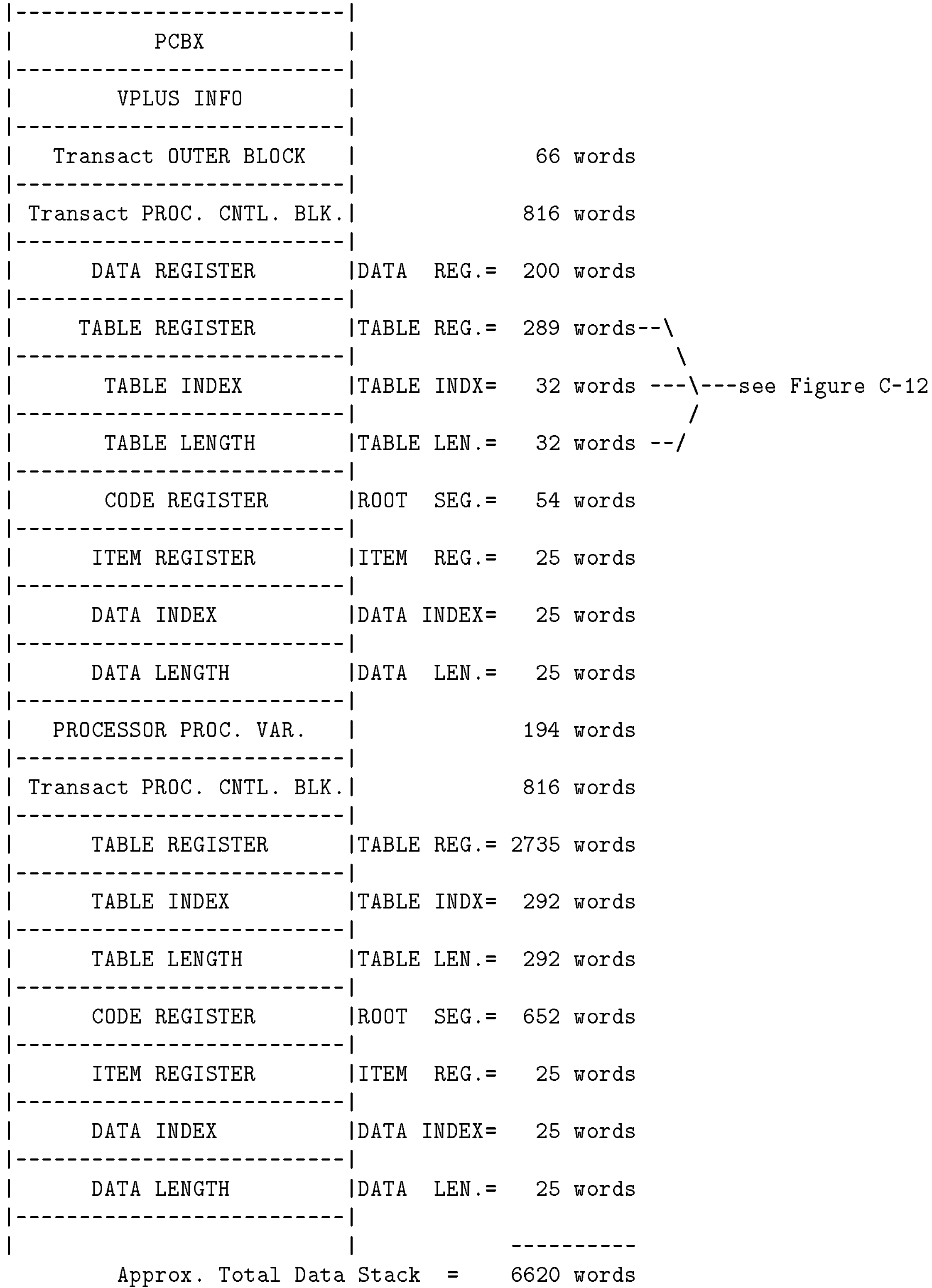
```
COMPILING WITH OPTIONS: CODE,DICT,STAT,ERRS

*****COMPILE TIME STATISTICS****
     STACK=      13640
     TABLE=       5622

*******RUN TIME STATISTICS******
     PCODE=          0
     SCODE=       1456

***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=   19,  310
       SET=   10,  146
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   20,  249
     STRNG=   66,  793
     CNTRL=   32,  120
          ------------
             159, 1693

****FINAL TABLE REG. SUMMARY****
WORK  AREA=    8,   60
          ------------
 TABLE REG.=       1753
 TABLE INDX=        167
 TABLE LEN.=        167

*****RUN TIME STACK SUMMARY*****
 DATA  REG.=        100
 TABLE REG.=       1753
 TABLE INDX=        167
 TABLE LEN.=        167
 ROOT  SEG.=       1456
 ITEM  REG.=         20
 DATA INDEX=         20
 DATA  LEN.=         20
           ------------
                   3703

 CODE FILE STATUS: REPLACED
```

Figure C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (4 of 5)

subprogram 4

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

 *****COMPILE TIME STATISTICS****
      STACK=      13640
      TABLE=       5178

  *******RUN TIME STATISTICS******
      PCODE=          0
      SCODE=        652

 ***PARTIAL TABLE REG. SUMMARY***
      BASE=    1,   10
      FILE=    2,   82
       SET=   10,  144
      PROC=    0,    0
     $$CMD=   11,   65
      $CMD=    0,    0
      ITEM=   57,  735
     STRNG=  103,  796
     CNTRL=   68,  703
          ------------
             252, 2535

****FINAL TABLE REG. SUMMARY****
WORK  AREA=   40,  200
          ------------
TABLE REG.=       2735
TABLE INDX=        292
TABLE LEN.=        292

*****RUN TIME STACK SUMMARY*****
DATA  REG.=        200
TABLE REG.=       2735
TABLE INDX=        292
TABLE LEN.=        292
ROOT  SEG.=        652
ITEM  REG.=         25
DATA INDEX=         25
DATA  LEN.=         25
          ------------
                  4246

CODE FILE STATUS: REPLACED
```

Figure C-10. Compiler Statistics for Program Using CALLs Without the SWAP Option (5 of 5)



```
|--------------------------|
|           PCBX           |
|--------------------------|
|        VPLUS INFO        |
|--------------------------|
|   Transact OUTER BLOCK   |             66 words
|--------------------------|
| Transact PROC. CNTL. BLK.|            816 words
|--------------------------|
|      DATA REGISTER       |DATA  REG.=  200 words
|--------------------------|
|     TABLE REGISTER       |TABLE REG.=  289 words--\
|--------------------------|                         \
|       TABLE INDEX        |TABLE INDX=   32 words ---\---see Figure C-12
|--------------------------|                          /
|       TABLE LENGTH       |TABLE LEN.=   32 words --/
|--------------------------|
|       CODE REGISTER      |ROOT  SEG.=   54 words
|--------------------------|
|      ITEM REGISTER       |ITEM  REG.=   25 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=   25 words
|--------------------------|
|      DATA LENGTH         |DATA  LEN.=   25 words
|--------------------------|
|   PROCESSOR PROC. VAR.   |            194 words
|--------------------------|
| Transact PROC. CNTL. BLK.|            816 words
|--------------------------|
|      TABLE REGISTER      |TABLE REG.= 2735 words
|--------------------------|
|      TABLE INDEX         |TABLE INDX=  292 words
|--------------------------|
|      TABLE LENGTH        |TABLE LEN.=  292 words
|--------------------------|
|      CODE REGISTER       |ROOT  SEG.=  652 words
|--------------------------|
|      ITEM REGISTER       |ITEM  REG.=   25 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=   25 words
|--------------------------|
|      DATA LENGTH         |DATA  LEN.=   25 words
|--------------------------|
|                          |           ----------
        Approx. Total Data Stack  =    6620 words
```

Figure C-11. Data Stack of Program Using CALLs Without the SWAP Option

```
                  |-------------------------|
                  |                         |
                  |          BASES          |      BASE=   1,   10 words
                  |                         |
                  |-------------------------|
                  |  VPLUS COMAREA; VPLUS,  |      FILE=   2    83 words
                  |   KSAM, MPE, AND DATA   |
                  |     SET FILE INFO       |       SET=   0,    0 words
                  |-------------------------|
                  |       PROCEDURES        |      PROC=   0,    0 words
                  |-------------------------|
                  |        COMMANDS         |    $$CMD=  11,   65 words
                  |-------------------------|
                  |       SUBCOMMANDS       |     $CMD=   0,    0 words
                  |-------------------------|
                  |                         |
                  |         ITEMS           |      ITEM=   1,    9 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |      TEXT STRINGS       |    STRNG=  10,   67 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |    CONTROL STRINGS      |    CNTRL=   2,    5 words
                  |                         |
                  |-------------------------|
                  |                         |
                  |      WORK SPACE         | WORK AREA=   5,   50 words
                  |                         |
                  |                         |              ^     ^
                  |-------------------------|              |     |
                                                           |     |
 TABLE INDEX and TABLE LENGTH entities--------------------+     |
                                                                 |
               TABLE REGISTER entities------------------------+
```

Figure C-12. Table Register Entities of Main Program Using CALLs Without the SWAP Option

Programs Using CALLs with the SWAP Option

If your main program is large, the SWAP option can reduce the amount of data stack space required. This option causes some of the main program's stack entities to be written out to a temporary file when a subprogram is called. The trade-off in this instance is the overhead required to create this file and restore its contents when control returns to the main program.

The compiler statistics provided for this program structure are the same as those provided when a program uses CALLs without the SWAP option. Refer back to Figure C-10 for compiler statistics produced when the earlier example was recoded to use the SWAP option with its CALLs.

When the main program is in control, the data stack looks like the top portion of the layout illustrated in Figure C-11. Components PCBX through PROCESSOR PROC. VAR. are present.

Figure C-13 illustrates how the data stack looks after subprogram 4 is called:

- Only a subset of the main program's TABLE REGISTER, TABLE INDEX, and TABLE LENGTH components are on the stack. The remainder of the entities have been placed in a temporary MPE file.
- The following components of the main program have also been placed in the temporary file: CODE REGISTER, ITEM INDEX, and DATA LENGTH.
- Two areas of the data stack are used for processor variables: PROCESSOR PROC. VAR. and SWAP PROC. VARIABLES. As in the case of CALLs without the SWAP option, the number of words in these areas is version dependent.

The entities in the main program's table register subsets are identified in Figure C-14. Note that the values for BASE=, FILE=, and SET= entities are represented in the compiler statistics for the main program in the PARTIAL TABLE REG. SUMMARY (refer to the first page of Figure C-10).

Figure C-15 illustrates the table components for the largest subprogram—subprogram 4.

```
|--------------------------|
|           PCBX           |
|--------------------------|
|        VPLUS INFO        |
|--------------------------|
|   Transact OUTER BLOCK   |             66 words
|--------------------------|
|    Transact PROCESSOR    |
|      CONTROL BLOCK       |            816 words
|--------------------------|
|      DATA REGISTER       |DATA  REG.=  200 words
|--------------------------|
| SUBSET OF TABLE REGISTER |             93 words--\
|--------------------------|                        \
| SUBSET OF TABLE INDEX    |              3 words ---\--see Figure C-14
|--------------------------|                         /
| SUBSET OF TABLE LENGTH   |              3 words --/
|--------------------------|
|   PROCESSOR PROC. VAR.   |            194 words
|--------------------------|
|   SWAP PROC. VARIABLES   |             67 words
|--------------------------|
|   Transact PROCESSOR     |            816 words
|      CONTROL BLOCK       |
|--------------------------|
|      TABLE REGISTER      |TABLE REG.= 2735 words--\
|--------------------------|                        \
|      TABLE INDEX         |TABLE INDX=  292 words ---\--see Figure C-15
|--------------------------|                         /
|      TABLE LENGTH        |TABLE LEN.=  292 words --/
|--------------------------|
|      CODE REGISTER       |ROOT  SEG.=  652 words
|--------------------------|
|      ITEM REGISTER       |ITEM  REG.=   25 words
|--------------------------|
|      DATA INDEX          |DATA INDEX=   25 words
|--------------------------|
|      DATA LENGTH         |DATA  LEN.=   25 words
|--------------------------|
|                          |           ----------
        Approx. Total Data Stack  =    6304 words
```

Figure C-13.
Data Stack of Program Using CALLs With the SWAP Option
(CALLed Program is on the Stack)

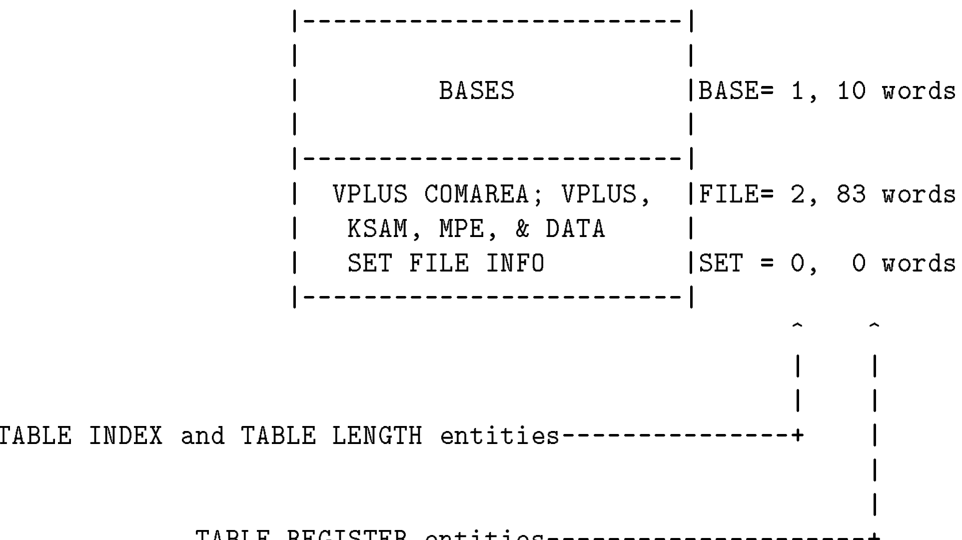


Figure C-14. Table Register Subsets for Main Program After CALLing Subprogram

```
                  |--------------------------|
                  |                          |
                  |          BASES           |      BASE=   1,    10 words
                  |                          |
                  |--------------------------|
                  |  VPLUS COMAREA; VPLUS,   |      FILE=   2     82 words
                  |   KSAM, MPE, AND DATA    |
                  |     SET FILE INFO        |       SET=  10,   144 words
                  |--------------------------|
                  |       PROCEDURES         |      PROC=   0,     0 words
                  |--------------------------|
                  |        COMMANDS          |    $$CMD=  11,    65 words
                  |--------------------------|
                  |       SUBCOMMANDS        |     $CMD=   0,     0 words
                  |--------------------------|
                  |                          |
                  |          ITEMS           |      ITEM=  57,   735 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |      TEXT STRINGS        |    STRNG= 103,   796 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |    CONTROL STRINGS       |    CNTRL=  68,   703 words
                  |                          |
                  |--------------------------|
                  |                          |
                  |       WORK SPACE         | WORK AREA=  40,   200 words
                  |                          |
                  |                          |             ^      ^
                  |--------------------------|             |      |
                                                           |      |
TABLE INDEX and TABLE LENGTH entities--------------------+      |
                                                                  |
               TABLE REGISTER entities-------------------------+
```

Figure C-15. Table Register Entities of Subprogram 4

Stack Usage Comparison

The following table summarizes the data stack requirements of the four program examples just examined. The values shown do not include the stack space required for the following components: PCBX, VPLUS, and subsystems such as SORT.

Table C-1. Example of Data Stack Requirements

Application Structure	Approximate Data Stack
Single-Segment Program	10957 words
Multiple-Segment Program	7197 words
Main Program CALLing Sub-Programs Without SWAP Option	6620 words
Main Program CALLing Sub-Programs With SWAP Option	6304 words

The main program in the final case is very small, so the savings in stack space are not as significant as they could be.

Processing Time Optimization

The following guidelines can help you improve the efficiency of your Transact p-code at run-time:

- Adjust the WORK= option of the SYSTEM statement to minimize the work space recoveries during execution. The number of work space recoveries can be determined by running test mode 101 or 102. Adjusting work space size may increase data stack requirements.
- Use DEFINE(INTRINSIC) to call system intrinsics whenever possible. This construct prevents the Transact processor from using LOADPROC dynamically to return the P-label of the intrinsic being called. Using DEFINE(INTRINSIC) reduces the overhead of loading the Transact program.
- Avoid calling many separate user-defined procedures from a Transact application. One LOADPROC is executed per procedure, contributing to processing overhead. If possible, combine all user-defined procedures into one procedure and identify the procedure to be executed with a control or index parameter.
- Avoid using the UNLOAD option of the PROC verb with frequently called procedures, since both LOADPROC and UNLOADPROC are called each time a procedure is called.
- Use the NOLOAD option of the PROC verb for infrequently called procedures such as error routines.
- Use UNLOAD to release table entries as appropriate, since the 255-entry/process limit of the Loader Segment Table (LST) is likely to be exceeded. A preferred approach is to combine user-defined procedures whenever possible.
- Avoid mixing character modes and block modes during a single application. This mixture requires considerable overhead in VPLUS whenever the switch from block mode to character mode occurs.
- Minimize the processing overhead required for opening and closing forms files by using only one forms file in any program or subprogram. Only one VPLUS forms file can be opened at a time.
- Avoid switching between segments to minimize the input/output overhead incurred in loading segment information into the data stack.

  Segments should conform as much as possible to the functional characteristics of the application. Commonly used routines should be grouped in the root segment (segment 0), since this segment is always memory-resident. However, this segment should be as small as possible.
- Minimize the number of calculations performed. If you need extensive numeric calculations, consider using subroutines in other languages and invoking them with the PROC statement.
- Use the MOVE verb whenever possible to transfer values between data items. The MOVE statement does no data type checks or conversions. The LET verb, however, performs time-consuming data type compatibility checks.
- Place frequently referenced data items on the top of the list register to minimize searches. For example, place them towards the end of the LIST verb statement. The list register is implemented as a linked list with list searches starting from the top of the list.

- Avoid using fragmented lists when accessing databases. Transact has to unscramble the list before database input/output operations are performed.
- Minimize internal sorting of large files.
- Follow these guidelines when using the LET verb:
  - It is most efficient to use single word integer types (I or J) for single +, -, =, or negation operations.
  - Use long reals (E or R) for single operations that include *, /, //, LN, LOG, SQRT, or exponentiation, as well as +, -, =, or negation operations.
  - Use packed decimal types (P) for all other operations.
  - Avoid mixing types within an operation.
- List the data sets in ascending data set order when using the LOCK option on the LOGTRAN verb.

D

Architected Call Interface (ACI)

Introduction

The Architected Call Interface (ACI) allows you to call existing Transact/iX subprograms from COBOL or Pascal.

ACI is provided as a single intrinsic call. It provides a means of invoking the desired Transact/iX system and then returning a status value that indicates the success or failure of the call. It also allows data to be passed by reference to the Transact/iX subprogram. These subprograms must reside in an executable library (XL) that has been provided with the RUN command.

Transact/iX subprograms must be compiled with the subprogram compiler option.

Syntax

`TL_CALL_TRANSACT`*(program_name, data_buffer, data_length, return_status)*

Parameters

program_name Character array by reference

Contains the name of the Transact/iX program to call (as specified in a SYSTEM statement). The program name can be uppercase or lowercase, but must be terminated with a blank. (A program name entered in lowercase will be automatically changed to uppercase prior to executing the call.) Because program names cannot be longer than six characters, the *return_status* parameter will return a non-zero value if the name is too long. It will also return a non-zero value if the program name cannot be found in the program's execution path.

data_buffer User defined structure by reference

Provides a data passing method to and from the Transact/iX subprogram and the calling program. This parameter is similar to the DATA= option in the CALL verb. The called program will start listing items at the beginning of this structure. Data can be passed down to the Transact/iX program by placing data in this structure prior to the intrinsic call. All items listed in the Transact/iX subprogram must be declared in this structure whether or not any data is to be shared. A series of appropriate LIST statements in the Transact/iX code will then allow you to interpret the data in this structure. Any modification of this data in the Transact/iX subprogram causes this structure to be directly modified in the calling program's data area.

Note

If the data buffer of the calling program is NOT defined exactly as the Transact/iX subprogram's data register, an error message, such as `DATA MEMORY PROTECTION TRAP`, is issued and the program terminates.

data_length

32-bit signed integer by value

The number of bytes in the *data_buffer* parameter, which is also the maximum data register size (in bytes) for the Transact/iX subprogram. This parameter takes precedence over any DATA= *data_length* specifications in the Transact/iX subprogram.

Note

When another language or third party package calls a Transact/iX subprogram, the data register space allocated to the subprogram is determined by the *data_length* parameter, not the DATA= *data_length* option of the SYSTEM statement in the called program. However, the *data_count* on the DATA= option is still observed.

The DATA= *data_length* option of the SYSTEM statement and the *data_length* parameter differ in how they are declared. The DATA= *data_length* is declared in 16-bit words; the *data_length* parameter is declared in bytes.

return-status

32-bit signed integer by reference

Is used to test the success or failure of the intrinsic call. This parameter is set to the following:

-2 — One or more errors occurred in the called program other than the following errors:

- Data entry errors for interactive programs.
- Errors processed by the subprogram using the STATUS option or the ERROR= option.

-1 — Abnormal end. A nonrecoverable error occurred while calling the Transact/iX subprogram.

0 — Successful execution of the call to a Transact/iX subprogram.

1 — The call failed because the *program_name* parameter contained a program name that was too long or not terminated with a blank.

2 — The call failed because the *program_name* parameter contained a program name that could not be found in the execution path of the program file. The Transact/iX subprogram was either not put in the user XL or the XL was not included in the RUN statement.

3 — Internal error. A problem arose from **hpgetprocplabel** or **hpmyprogram** intrinsic.

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 you call a Transact/iX subprogram, all the databases and files specified in the SYSTEM statement will be opened, regardless of whether or not they are accessed by the calling program.

When you call a Transact/iX subprogram, a new process is created for that subprogram. Because the Transact/iX subprogram and the calling program are two distinct processes, only the *data_buffer* is shared between the two processes. 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 can be 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;
```

```
begin
system_name := 'SYS1 ';

with data_buffer do
   begin
   x_item     := 'ABCDEFGH';
   i4_item    := 12345;
   i2_item    := 321;
   nine_item  := '111111';
   j4_item    := 2222;
   r4_item    := 99.12;
   packed_item[1] := 5;
   packed_item[2] := 5;
   packed_item[3] := 5;
   packed_item[4] := 5;
   packed_item[5] := 5;
   packed_item[6] := 12;   {C - sign bit}
   end;

tl_call_transact(system_name,
                 addr(data_buffer),
                 sizeof(data_buffer),
                 return_status);
if (return_status <> 0) then
   writeln ('Error calling TRANSACT/iX, Error Number:',return_status);

end.
```

Pascal Commands

This code can be compiled into a main program and executed with the following commands:

```
:pasxllk pastest
:run $oldpass;xl='userxl'
```

This runs the above Pascal application PASTEST, which calls the subprogram SYS1 that resides in USERXL.

COBOL Code

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBTEST.
AUTHOR.      HEWLETT-PACKARD.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SYSTEM-NAME          PIC  X(7).
01 RETURN-STATUS        PIC S9(5) COMP SYNC.
01 DATA-BUFFER.
   03 X-ITEM            PIC  X(8).
   03 I4-ITEM           PIC S9(5) COMP.
   03 I2-ITEM           PIC S9(4) COMP.
   03 NINE-ITEM         PIC  X(6).
   03 J4-ITEM           PIC S9(5) COMP.
   03 R4-ITEM           PIC S9(5) COMP.
   03 PACKED-ITEM       PIC S9(5) COMP-3.
   03 FILLER            PIC  X(2018).
01 DATA-LENGTH          PIC S9(5) COMP SYNC.

PROCEDURE DIVISION.
MAIN-PROGRAM.
   MOVE "SYS1   "     TO SYSTEM-NAME.
   MOVE 2048          TO DATA-LENGTH.

   MOVE "ABCDEFGH"    TO X-ITEM.
   MOVE 12345         TO I4-ITEM.
   MOVE 321           TO I2-ITEM.
   MOVE "111111"      TO NINE-ITEM.
   MOVE 2222          TO J4-ITEM.
   MOVE 55555         TO PACKED-ITEM.

   CALL "TL-CALL-TRANSACT" USING
                      SYSTEM-NAME,
                      @DATA-BUFFER,
                      \DATA-LENGTH\,
                      RETURN-STATUS.

   IF RETURN-STATUS IS NOT ZERO THEN
      DISPLAY "Error calling TRANSACT/iX, Error Number: "
              RETURN-STATUS.

   STOP RUN.
```

COBOL Commands

This code can be compiled into a main program and executed with the following commands:

```
:cob85xlk cobtest
:run $oldpass;xl='userxl'
```

This runs the above COBOL application COBTEST, which calls the Transact/iX subprogram SYS1 that resides in USERXL.

Pascal Code With Status

The following example shows how to include the status for the execution of the Transact/iX subprogram. This example uses a different data structure from the previous examples, but it can be compiled and executed in a similar way.

```
  program pastest2(output);
  $standard_level 'OS_FEATURES'$

type
   system_name_type = packed array[1..7] of char; {system name plus blank}
   data_record = packed record                    {Data register}
      tstatus    : integer;                       {I(9)  }
      x_item     : packed array[1..8] of char;    {X(8)  }
      i4_item    : integer;                       {I(9)  }
   end;

var
   data_buffer     : data_record;
   return_status   : integer;
   program_name    : program_name_type;

procedure tl_call_transact
   (
   var   system_name  : system_name_type;
         data_buffer  : localanyptr;
         data_length  : integer;
   var   return_status: integer
   ); external;

begin
system_name := 'SYS2 ';
```

```
with data_buffer do
   begin
   tstatus    := 0;
   x_item     := 'ABCDEFGH';
   i4_item    := 12345;
   tl_call_transact(system_name,
                    addr(data_buffer),
                    sizeof(data_buffer),
                    return_status);
   if (return_status <> 0) then
      writeln ('Error calling TRANSACT/iX, Error Number:',return_status);

   if (tstatus <> 0) then
      writeln ('Error during execution of TRANSACT/iX program:',tstatus);
   end;

end.
```

E

Native Language Support

Transact provides access to MPE native language support (NLS) at compile time and at run time. NLS is used to adapt programs to other languages by providing message catalogs, collating sequences, data formats, and numerical formats specific to a particular language. The default language is NATIVE-3000, which consists of the language attributes of the HP 3000 prior to NLS.

The SET(LANGUAGE) Statement

The SET(LANGUAGE) statement specifies the native language to be used by Transact:

```
SET(LANGUAGE) [language[,STATUS]];
```

The SET(LANGUAGE) statement allows the programmer to specify or change the native language at run time. The programmer can either specify a literal language name or number in quotes (which is checked at compile time) or the name of a data item which will contain the language number at run time. The data item must begin on a word boundary. Refer to the *Native Language Support Reference Manual* for a list of the names and numbers assigned to the available languages.

If STATUS is not specified and 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 either to the number of the current language and then resets the language number to 0 (NATIVE-3000), or to the language number of the calling program if SET(LANGUAGE) is issued in a called program. A compiler error results if the STATUS option is specified without *language*.

When you change languages using SET(LANGUAGE), any previously entered data is unchanged and remains in the format in which it was entered. Any new data is stored in the format appropriate to the specified language.

The RESET(LANGUAGE) Statement

The RESET(LANGUAGE) statement sets the STATUS register to the current language and then resets the language to zero (NATIVE-3000) or the calling program's language if issued in a called program. The STATUS option is not permitted with this statement.

Specifying the Language for the Compiler and Processor

Transact uses the native language message catalog for prompts and messages. You can set the language to be used by issuing the SETJCW command to set the NLS job control word NLUSERLANG to the desired language number, resulting in the messages and prompts being drawn from the catalog for that language. Refer to the *MPE Commands Reference Manual* and the *Native Language Support Reference Manual* for more information.

Alternatively, the job control word can be overridden by Transact, which switches to the language specific catalog when a SET(LANGUAGE) statement is encountered at run time.

If an additional Transact program is executed, or the same program is restarted by responding to the EXIT/RESTART prompt, the language ID is set to NATIVE-3000.

Called Programs

For called programs, the native language remains the same as that of the caller, and the calling program can override the language specification set globally with the NLUSERLANG job control word. For example, if the language on a system is set to NATIVE-3000 with the NLS job control word NLUSERLANG, but an Inform/V report is called by a Transact program that specifies French with the SET(LANGUAGE) statement, the report will appear in French.

A called program is free to change the native language, but when it returns, the language in effect at the time of the call is restored. When a called program is restarted, the language ID is set to the language passed to it by the calling program.

Numeric Input

For each language, when processing numeric data items for input, the characters used for the thousand's indicator and the decimal indicator are ignored—provided they are not one of the two field delimiter characters (`,` and `=`) defined for Transact for use with command sequences. If the character used for an indicator is one of the field delimiters, such as for NATIVE-3000, where the thousand's indicator of comma (`,`) is also a default field delimiter, then you must specify another delimiter with a SET(DELIMITER) statement. See Chapter 8 for more information on the SET(DELIMITER) statement.

Numeric Output

In unedited numeric output, the language defined decimal character is used instead of a period.

For edited numeric output, the descriptions of edit masks for the DISPLAY and FORMAT verbs in Chapter 8 describe completely any variations that are dependent on native language.

Date and Time

The date and time displayed by $TODAY, $DATELINE, and $TIME use the time formats specified by the current language. However, the LIST options related to date and time use only NATIVE-3000.

IF and MATCH Changes

IF and MATCH comparisons of alphabetic strings are performed using the collating sequence of the language in effect when the comparison is actually done. In other words, the collating sequence used is not necessarily determined by the language in effect when the match register was set.

The connectors (AND, OR), the logical relators (GT, LT, EQ, and so on), and the range indicator ("TO" in NATIVE-3000) used in responding to PROMPT(MATCH) at run time can change from language to language, depending on the native language message catalog.

Upshifting and Character Types

The upshift and character type tables used at any given time always reflect the current language. These tables are retrieved whenever the language is changed. Since any previously entered data is unchanged and remains in the format in which it was entered, type U data items are not "re-upshifted" to reflect different language-particular upshift requirements if the language changes. However, any new data is upshifted as required and stored in the format appropriate to the specified language.

Intrinsics That Support Native Languages

Transact passes the current native language to any subsystem (intrinsic) it calls that accepts a language ID. If a user calls a subsystem through the PROC verb, it is the user's responsibility to provide the language ID if desired and accepted by the subsystem.

Examples:

Here is an example of using NLUSERLANG JCW to invoke the French catalog for Trancomp and Transact

```
:SETJCW NLUSERLANG,7
:RUN TRANCOMP.PUB.SYS
       or
:RUN TRANSACT.PUB.SYS
```

Here is an example of using SET(LANGUAGE) to invoke the German catalog from within a Transact program:

```
SET(LANGUAGE) "GERMAN";
       or
SET(LANGUAGE) "8",STATUS;
       or
       :
DEFINE(ITEM) LANGUAGE X (20);
LIST LANGUAGE;
MOVE(LANGUAGE)="GERMAN";
```

```
SET(LANGUAGE) LANGUAGE;
        ⋮
```

To change language back to NATIVE-3000 or parent program language:

```
        ⋮
SET(LANGUAGE);
        or
RESET(LANGUAGE);
```

Index

Special characters

!, 5-14, 8-37, 8-66, 8-83, 8-186, 8-237
$, 8-36, 8-66
(, 8-37, 8-67
*, 8-37, 8-66
,, 8-37, 8-66
., 8-37, 8-66
:$, 5-4
=, 5-15
], 5-14, 8-113, 8-218
]], 5-14, 8-113, 8-218
^, 5-15, 8-36, 8-66

2

24 edit characters, 8-38, 8-67

3

32-bit integer arithmetic, 8-108

A

AA and aa edit characters, 8-38, 8-68
A and a edit characters, 8-38, 8-68
ABORT command, 11-15
absolute binary, 3-4
access, key, 4-4
access mode, 6-2
ACCOUNT option, LIST verb, 8-116
ACI (Architected Call Interface), D-1
alias items, 3-17
ALIAS option, DEFINE verb, 8-22
alignment, in Transact/iX, 9-17
ALIGN option, LIST verb, 8-116, 8-118
ALPHABETIC test value, 8-85, 8-188, 8-239
  -LOWER, 8-86, 8-188, 8-239
  -UPPER, 8-86, 8-188, 8-239
APPEND option
  GET verb, 8-77
  PUT verb, 8-181
  SET verb, 8-209
  UPDATE verb, 8-234
applications optimization, C-1
Architected Call Interface (ACI), D-1
ARGLNG parameter, PROC verb, 8-160
ARG parameter, PROC verb, 8-160
argument register, 4-4
arithmetic operations, 8-103
arithmetic traps in TRANDEBUG, 11-13
arrays
  items defined, 3-11
  manipulating, 8-104
  subscripting, 3-11
ASCII function, LET verb, 8-96
asterisk edit character, 8-37, 8-66
AUTOLOAD option
  description, 5-12
  RESET verb, 8-200
  SET verb, 8-215
AUTO modifier, LIST verb, 8-117
AUTOREAD option, GET verb, 8-77
AUTORPT command, 11-16

B

BANNER option, SYSTEM verb, 8-223
BASELNG parameter, PROC verb, 8-160
BASE option, SYSTEM verb, 8-223
BASE parameter, PROC verb, 8-160
batch processing, 9-11
binding data item attributes in Transact/iX, B-4
BLANKS option
  DATA verb, 8-14
  INPUT verb, 8-90
  PROMPT verb, 8-172
BREAK DELETE command, 11-18
BREAK LIST command, 11-20
BREAK SET command, 11-21
buffer record, defining, 6-15
built-in commands, 5-6
BYTE parameter, PROC verb, 8-160

C

calling intrinsics or SL routines, 8-158
calling Transact/iX subprograms from COBOL or Pascal, D-1
CALL verb, 8-2, C-20, C-28
caret symbol (^)
  as edit character, 8-36, 8-66
  as selection criterion, 5-15
CCTL option
  DISPLAY verb, 8-35
  FORMAT verb, 8-65

CENTER option
DISPLAY verb, 8-35
FORMAT verb, 8-66
chained access path, 8-152
CHAIN modifier
DELETE verb, 8-27
FIND verb, 8-51
GET verb, 8-72
OUTPUT verb, 8-145
REPLACE verb, 8-191
character mode, 5-11
CHAR function, MOVE verb, 8-132
CHCK compiler option, 9-8
CHECKNOT option
DATA verb, 8-14
PROMPT verb, 8-172
CHECK option
DATA verb, 8-14
PROMPT verb, 8-172
child items, 3-10
CLEAR option
GET verb, 8-77
PUT verb, 8-181
SET verb, 8-210
UPDATE verb, 8-234
CLOSE modifier, FILE verb, 8-47
CLOSE verb, 8-10
closing a database, 6-2
COBOL
calling Transact/iX subprograms, D-1
code, D-6
commands, D-7
data types, 3-4
subroutines with Transact/iX, 8-164
CODE compiler option, 9-8
COL function, MOVE verb, 8-133
COL option
DISPLAY verb, 8-35
FORMAT verb, 8-66
comma, 5-15
comma edit character, 8-37, 8-66
$$command, 5-4
: command, 11-14
command
built-in, 5-6
labels, 5-4
qualifiers, 5-6
sequences, 5-2
COMMAND argument, SET verb, 8-207
COMMAND modifier
RESET verb, 8-199
SET verb, 8-207
COMMAND processor command, 5-6
comments, 2-7
compilation, 9-7
compiled output control, 9-2
compiler
bypassing prompts, 9-10
differences, 9-32
error messages, 7-8
execution, 9-7, 9-8
listings, 9-31
options, 9-8
options in Transact/iX, 9-17
output destination, 9-12
TRANCODE, 9-12
TRANIN, 9-11
TRANLIST, 9-10, 9-12
TRANOUT, 9-12
Transact/iX, 9-16
TRANTEXT, 9-10, 9-11
compiler commands
!COPYRIGHT, 9-2
!ELSE, 9-2
!ENDIF, 9-3
!IF, 9-2
!INCLUDE, 9-2
!LIST, 9-2
!NOLIST, 9-2
!PAGE, 9-2
!SEGMENT, 9-2
!SET, 9-2
compiling Transact/iX programs, 9-16, 9-22
compound data items, 3-11
compound statements, 2-5
COMPUTE option, DEFINE verb, 8-22
conditional test
IF verb, 8-83
REPEAT verb, 8-186
WHILE verb, 8-237
connector, 5-8
CONTINUE command, 11-24
CONTROL modifier, FILE verb, 8-48
conversion, B-7
converting file formats , B-8
converting programs, B-1
!COPYRIGHT compiler command, 9-2
COUNT parameter, PROC verb, 8-160
$CPU edit characters, 8-36
CPU seconds used, 3-2
$CPU variable, 3-2, 8-35
critical item update, 8-196
Ctrl Y, 5-14
operation break, 5-14
user responses, 5-14
currency symbol ($) edit character, 8-36, 8-66
CURRENT modifier
DELETE verb, 8-27
FIND verb, 8-51
GET verb, 8-72

REPLACE verb, 8-192
CURRENT option
GET verb, 8-77
OUTPUT verb, 8-145
PUT verb, 8-181
CURSOR option
PUT verb, 8-181
CURSOR option, GET verb, 8-77, 8-210, 8-234

D

database
closing, 6-2
data dictionary, 3-9
locking, 6-3
opening, 6-1
opening mode, 9-13
DATA BREAK DELETE command, 11-25
DATA BREAK LIST command, 11-27
DATA BREAK REGISTER command, 11-28
DATA BREAK SET command, 11-30
data dictionaries, 3-9
data entry control characters, 5-14
data file migration, B-8
data items, 3-2
adding to data register, 4-3
adding to list register, 4-3
alias items, 3-17
array items, 3-11
child items, 3-10
compound, 3-11
listed multiple times, 4-3
parent items, 3-10
removing from list register, 4-3
sizes, 3-3
types, 3-3
DATA LOG command, 11-33
DATA option
CALL verb, 8-4
SYSTEM verb, 8-225
data register, 4-2
managing, 4-3
data specification, 3-9
data stack optimization, C-1
data storage
registers, 4-1
requirements, 4-3
data types, 3-3, 3-8
compatibility with databases, 3-9
compatibility with dictionaries, 3-9
compatibility with VPLUS, 3-8
data validation, 3-9
DATA verb, 4-7, 8-12
date and time variable, 3-2
$DATELINE edit characters, 8-36
$DATELINE variable, 3-2, 8-35
DATE option, LIST verb, 8-116
DBLOCK call, A-1
DBUNLOCK call, A-1
D, DD, and DDD edit characters, 8-39, 8-68
DECIMAL parameter, PROC verb, 8-160
DEFINE(ITEM) statement, 2-3
DEFINE verb, 8-19
DEFN command, 11-35
DEFN compiler option, 9-8
DELETE verb, 8-27, A-2
executing for a KSAM file, A-3
executing for TurboIMAGE data set, A-2
deleting a breakpoint, 11-18
DELIMITER modifier
RESET verb, 8-200
SET verb, 8-208
delimiters, 2-7, 5-15
blank, 2-7
comma, 2-7
equal sign, 2-7
parentheses, 2-7
semicolon, 2-7
with DOEND, 2-5
DEPTH option, SET verb, 8-215
DICT compiler option, 9-8
DIRECT modifier
DELETE verb, 8-27
FIND verb, 8-51
GET verb, 8-72
OUTPUT verb, 8-145
REPLACE verb, 8-192
DISPLAY BASE command, 11-36
DISPLAY CALLS command, 11-37
DISPLAY COMAREA command, 11-38
DISPLAY FILE command, 11-39
displaying contents of comarea, 11-38
DISPLAY INPUT command, 11-40
DISPLAY ITEM command, 11-41
DISPLAY KEY command, 11-43
DISPLAY MATCH command, 11-44
DISPLAY PERFORM command, 11-45
DISPLAY STATUS command, 11-46
DISPLAY STATUSDB command, 11-47
DISPLAY STATUSIN command, 11-48
DISPLAY UPDATE command, 11-49
DISPLAY verb, 8-34
DO and DOEND statements, 2-5
!DOMAIN System Dictionary command, 9-3
double buffering parameters, Transact/iX, 8-165
dynamic calls, 8-2
compiling programs for, 9-21
DYNAMIC_CALLS compiler option, 9-17
dynamic roll-back, 6-9

E

! edit character, 8-66
$ edit character, 8-66
(edit character, 8-67
* edit character, 8-66
, edit character, 8-66
. edit character, 8-66
^ edit character, 8-66
edit characters
  !, 8-37, 8-66
  $, 8-36, 8-66
  (, 8-37, 8-67
  *, 8-37, 8-66
  ,, 8-37, 8-66
  ., 8-37, 8-66
  ^, 8-36, 8-66, 8-68
  24, 8-38, 8-67
  AA and aa, 8-38, 8-68
  A and a, 8-38, 8-68
  D, DD, and DDD, 8-39, 8-68
  for $CPU, 8-36
  for $DATELINE, 8-36
  for $PAGE, 8-36
  for $TIME, 8-38
  H and HH, 8-38, 8-67
  M and MM, 8-38, 8-67, 8-68, 8-69
  M, MM, and nM, 8-39
  nM and nm, 8-39, 8-69
  nW and nw, 8-39, 8-69
  S and SS, 8-38, 8-68
  T, 8-38, 8-68
  YY and YYYY, 8-39, 8-69
  Z, 8-36, 8-66
  ZD, 8-39, 8-68
  ZH, 8-38, 8-67
  ZM, 8-38, 8-39, 8-67, 8-68
  ZS, 8-38, 8-68
EDIT command, 11-50
EDIT option
  DEFINE verb, 8-22
  DISPLAY verb, 8-37
  FORMAT verb, 8-66
!ELSE compiler command, 9-2
!ENDIF compiler command, 9-3
END option
  RESET verb, 8-200
  SET verb, 8-215
END verb, 8-44, 8-157
ENTRY modifier, DEFINE verb, 8-19
ENTRY option, DEFINE verb, 8-22
entry point labels, 9-5
environment variables, 9-21
equals sign, 5-15
error branching, 8-112
error handling, 7-1
  automatic, 7-2
  status register, 4-7
error messages
  compiler, 7-8
  looking up, 7-10
  searching catalogs, 7-10
  system errors, 7-10
  Transact processor, 7-9
  warnings, 7-10
ERROR option
  CLOSE verb, 8-10
  DELETE verb, 8-28
  FIND verb, 8-52
  GET verb, 8-76
  LET verb, 8-94
  OUTPUT verb, 8-146
  PATH verb, 8-152
  PUT verb, 8-180
  REPLACE verb, 8-192
  UPDATE verb, 8-233
  when taken, 7-4
errors
  database operation, 7-2, 7-6
  data entry, 7-2, 7-5
  file operation, 7-2, 7-6
  MPE/iX, 7-10
  MPE V, 7-10
  Transact, 7-10
ERRS compiler option, 9-8
exclamation point edit character, 8-37, 8-66
EXCLAMATION variable
  IF verb, 8-83
  REPEAT verb, 8-186
  WHILE verb, 8-237
executing Transact/iX programs, 9-16, 9-22
EXIT argument, SET verb, 8-207
exiting from LEVEL sequences, 8-113
EXIT OR RESTART message, 8-44
EXIT processor command, 5-6
EXIT verb, 8-46
EXPLAIN subsystem, 7-10
external procedure, PROC verb, 8-158

F

FEDIT option
  GET verb, 8-78
  PUT verb, 8-182
  SET verb, 8-210
  UPDATE verb, 8-234
FIELD command qualifier, 5-6
field delimiters, 5-15
FIELD option
  RESET verb, 8-200
  SET verb, 8-215

FIELD variable
IF verb, 8-84
REPEAT verb, 8-186
WHILE verb, 8-238
file format conversion, B-8
FILEID parameter, PROC verb, 8-160
file locking, 6-3
file names, reserved, 9-6
FILE option, SYSTEM verb, 8-225
FILE verb, 8-47
FIND verb, 8-51, A-4
executing for a KSAM file, A-6
executing for an MPE file, A-7
executing for a TurboIMAGE data set, A-4
FKEY option
GET verb, 8-78
PUT verb, 8-182
UPDATE verb, 8-234
floating point formats, B-2
FLOCK call, A-1
flowcharts, A-1
Fn option
GET verb, 8-78
PUT verb, 8-182
UPDATE verb, 8-234
FOPEN call, A-1
formatting parameters
DISPLAY verb, 8-35
FORMAT verb, 8-65
FORMAT verb, 8-64
FORM modifier
GET verb, 5-16, 8-72
PUT verb, 5-16, 8-177
SET verb, 8-208
UPDATE verb, 8-231
FORMSTORE option
RESET verb, 8-200
SET verb, 8-216
FREEZE option
GET verb, 8-78
PUT verb, 8-182
SET verb, 8-210
UPDATE verb, 8-235
FSTORESIZE option
description, 5-12
SYSTEM verb, 8-227
function keys, 5-16
FUNLOCK call, A-1

G

GET(FORM) verb, executing for a VPLUS form, A-12
getting information online, 7-10
GET verb, 8-72, A-9
executing for a KSAM file, A-10
executing for an MPE file, A-11
executing for a TurboIMAGE data set, A-9
GO TO verb, 8-82
GROUP option, LIST verb, 8-116

H

H and HH edit characters, 8-38, 8-67
HEAD option
DEFINE verb, 8-22
DISPLAY verb, 8-40
FORMAT verb, 8-69
SET verb, 8-217
HELP command, 11-51
home base, 3-2, 8-223
HOMEGROUP option, LIST verb, 8-116
$HOME variable, 3-2
HP3000_16 compiler option, 9-17

I

!IF compiler command, 9-2
IF verb, 8-83
!INCLUDE compiler command, 9-2
INFO= option in Transact/iX, 9-17
INFO= option, RUN command
compiler, 9-10
processor, 9-14
information messages,Transact processor, 7-9
Inform/V option, CALL verb, 8-5
INITIALIZE argument, SET verb, 8-207
INITIALIZE command, under MPE/iX, B-5
INITIALIZE processor command, 5-6
INIT option
DEFINE verb, 8-24
GET verb, 8-78
LIST verb, 8-116
PUT verb, 8-182
SET verb, 8-210
UPDATE verb, 8-235
INPUTLNG parameter, PROC verb, 8-161
INPUT parameter, PROC verb, 8-160
input register, 4-6
INPUT variable
IF verb, 8-84
REPEAT verb, 8-187
WHILE verb, 8-238
INPUT verb, 4-7, 8-90
integer number, 3-4
interpreting Transact programs, 9-13
INTRINSIC modifier, DEFINE verb, 8-19
intrinsics, calling, 8-158
invoking intrinsics or SL routines, 8-158
invoking other programs, 8-2
IPC (message) files, 6-21
item attribute resolution, B-4

ITEMLNG parameter, PROC verb, 8-161
ITEM modifier
DATA verb, 8-14
DEFINE verb, 8-19, 8-24, 8-25
ITEM parameter, PROC verb, 8-161
ITEM verb, 8-92

J

JOIN option
DISPLAY verb, 8-40
FORMAT verb, 8-69

K

KEYLNG parameter, PROC verb, 8-161
KEY modifier
DATA verb, 8-15
GET verb, 8-72
LIST verb, 8-117
PROMPT verb, 8-173
SET verb, 8-213
KEY parameter, PROC verb, 8-161
key register, 4-4
key value, 4-4
KSAM files, 6-15
CLOSE verb, 8-10
DELETE verb, 8-27
FIND verb, 8-51
LIST verb, 8-115
OUTPUT verb, 8-144
SYSTEM verb, 8-227
UPDATE verb, 8-230
KSAM option, SYSTEM verb, 8-227

L

LABEL BREAK SET command, 11-53
LABEL JUMP command, 11-55
labels, 8-157
command, 5-4
statement, 2-4
subcommand, 5-4
LANGUAGE modifier, SET verb, 8-213
language option, PROC verb, 8-163
LEADER option
DATA verb, 8-15
LIST verb, 8-118
PROMPT verb, 8-174
LEFT option
DISPLAY verb, 8-40
FORMAT verb, 8-69
SET verb, 8-217
LENGTH function, LET verb, 8-97
LET verb, exponentiation, 8-94
LEVEL modifier, END verb, 8-44, 8-113
LEVEL verb, 8-113
limitations, 6-13
LINE destination variable, LET verb, 8-93
LINE option
DISPLAY verb, 8-40
FORMAT verb, 8-69
LINE variable, 3-2
LINK command, 9-28
LINKEDIT command, 9-29
!LIST compiler command, 9-2
LIST compiler option, 9-8
list file, 9-8
listing breakpoints, 11-20
listing data breakpoints, 11-27
LIST option
DELETE verb, 8-28
FIND verb, 8-52
GET verb, 8-73
OUTPUT verb, 8-146
PATH verb, 8-152
PUT verb, 8-178
REPLACE verb, 8-192
SET verb, 8-210
UPDATE verb, 8-231
list register, 2-3, 4-2
managing, 4-3
list statement, 2-3
LIST verb, 4-7, 8-115
LN function, LET verb, 8-98
LNG option
DISPLAY verb, 8-40
FORMAT verb, 8-70
local form storage, 5-11, 5-13
LOC command, 11-56
locking, 6-9
across a transaction, 6-8
LOCK option, 6-7
optimized, 6-5
options available, 6-3
locking strategy with LOGTRAN, 6-10
LOCK option
DELETE verb, 8-30
FIND verb, 8-55
GET verb, 8-76
LOGTRAN verb, 8-124
OUTPUT verb, 8-148
PUT verb, 8-181
REPLACE verb, 8-194
UPDATE verb, 8-233
with database access verbs, 6-7
with file access verbs, 6-7
with LOGTRAN statement, 6-8
LOG command, 11-57
LOG function, LET verb, 8-99
logical connector, 5-8
logical value, 3-4

LOGTRAN verb, 8-122
locking strategy, 6-10
look-ahead loading of forms, 5-12
LOWER function, MOVE verb, 8-134

M

M and MM edit characters, 8-38, 8-67, 8-68, 8-69
MATCH modifier
DATA verb, 5-8, 8-15
LIST verb, 8-118
PROMPT verb, 5-8, 8-173
SET verb, 8-213
MATCH option, RESET verb, 8-201
MATCH prompt, 5-8, 5-15
match register, 4-5
match specification characters, 5-15
message files, 6-21
migration
checklist, B-11
examples, B-9
Transact/V data files to native mode Transact/iX, B-8
Transact/V programs to native mode Transact/iX, B-1
Transact/V source programs to native mode Transact/iX, B-7
M, MM, and nM edit characters, 8-39
mode
database access, 6-2
execution, 9-13
modifiers, 2-4
MODIFY INPUT command, 11-59
MODIFY ITEM command, 11-60
MODIFY KEY command, 11-61
MODIFY MATCH command, 11-62
MODIFY STATUS command, 11-64
MODIFY UPDATE command, 11-65
MOVE verb, 8-128
string functions, 8-131
MPE files, 6-15
automatic purging, 8-226
CLOSE verb, 8-10
LIST verb, 8-115
MPE/iX operating system, 9-16
MR (multiple RIN), 6-4
multiple-segment programs, C-14
data stack components, C-5
multiple systems in one file, B-6

N

naming conventions
data items, 3-2
subcommand labels, 5-4
user-entered passwords, 5-5
native language support, E-1
called programs, E-2
date and time, E-3
IF and MATCH changes, E-3
intrinsic calls, E-3
numeric input, E-2
numeric output, E-2
RESET(LANGUAGE)statement, E-1
SET(LANGUAGE) statement, 8-213
upshifting and character types, E-3
NEED option
DISPLAY verb, 8-40
FORMAT verb, 8-70
negative values in edit string, 8-37, 8-67
nesting level, 9-31
examples, 9-32
nM and nm edit characters, 8-39, 8-69
NMDEBUG command, 11-67
NOBANNER option, SET verb, 8-217
NOCOUNT option
DELETE verb, 8-30
OUTPUT verb, 8-148
REPLACE verb, 8-195
NOCRLF option
DISPLAY verb, 8-41
FORMAT verb, 8-70
NOECHO option
DATA verb, 8-14
INPUT verb, 8-90
PROMPT verb, 8-172
NOFIND option, GET verb, 8-76
NOHEAD option
DISPLAY verb, 8-41
FORMAT verb, 8-70
OUTPUT verb, 8-148
RESET verb, 8-201
SET verb, 8-217
!NOLIST compiler command, 9-2
NOLOAD option, PROC verb, 8-162
NOLOCK option
RESET verb, 8-201
SET verb, 8-217
NOLOOKAHEAD option
description, 5-12
RESET verb, 8-201
SET verb, 8-218
NOMATCH option
DELETE verb, 8-30
FIND verb, 8-55
GET verb, 8-76
OUTPUT verb, 8-148
REPLACE verb, 8-195
NOMSG option
CLOSE verb, 8-11
DELETE verb, 8-30

FIND verb, 8-55
GET verb, 8-76
LOGTRAN verb, 8-124
OUTPUT verb, 8-148
PATH verb, 8-152
PUT verb, 8-181
REPLACE verb, 8-195
UPDATE verb, 8-233
NOSIGN option
DISPLAY verb, 8-41
FORMAT verb, 8-70
!NOSYSDIC System Dictionary command, 9-3
NOTEST option, SYSTEM verb, 8-228
NOTRAP option, PROC verb, 8-162
NULL option, DATA verb, 8-14
null parameters in Transact/iX, 8-164
null subcommand, 5-4
numeric ASCII string, 3-4
numeric parameters, 10-4
NUMERIC test value, 8-85, 8-188, 8-239
nW and nw edit characters, 8-39, 8-69

O

OBJT compiler option, 9-9
OFFSET variable, LET verb, 8-93
opening a database, 6-1
OPEN modifier, FILE verb, 8-48
operation break, Ctrl Y, 5-14
operations, arithmetic, 8-103
OPT@ compiler option, 9-9, C-4
OPTE compiler option, 9-9, C-4
OPTH compiler option, 9-9, C-4
OPTI compiler option, 9-9, C-4
OPTIMIZE compiler option, 9-18
optimized locking, 6-5
optimizing applications, C-1
optimizing data stacks, C-1
optimizing processor time, C-33
optimizing programs, C-33
option-list, 2-5
OPTION modifier
RESET verb, 8-200
SET verb, 8-215
OPTION option
SYSTEM verb, 8-228
option variable intrinsics, 8-158
option variable procedures, Transact/iX, 8-164
optlock parameter
description, 6-3
SYSTEM verb, 8-224
OPT option, DEFINE verb, 8-24
OPTP compiler option, 9-10
OPTS compiler option, 9-10, C-4
order of evaluation
in conditionals, 8-86, 8-188, 8-240
OUTPUT verb, 8-144, A-14
executing for a KSAM file, A-15
executing for an MPE file, A-16
executing for a TurboIMAGE data set, A-14
overlays, 9-5

P

packed decimal, 3-4
packed decimal arithmetic, 8-110
PAGE BACK command, 11-68
!PAGE compiler command, 9-2
$PAGE edit characters, 8-36
PAGE FORWARD command, 11-69
PAGE JUMP command, 11-70
page number variable, 3-2
PAGE option
DISPLAY verb, 8-41
FORMAT verb, 8-70
$PAGE variable, 3-2, 8-35
PAGE variable, 3-2
LET verb, 8-93
PALIGN option, SET verb, 8-218
parent data items, 3-10
parenthesis edit character, 8-37, 8-67
PARM= RUN command option, 9-10
Pascal
calling Transact/iX subprograms, D-1
code, D-4
commands, D-5
passing control
to intrinsics or SL routines, 8-158
to other programs, 8-2
PASSWORD option, LIST verb, 8-117
passwords
commands and subcommands, 5-5
PATH modifier
DATA verb, 8-16
PROMPT verb, 8-174
PATH option, LIST verb, 8-118
PATH verb, 8-152, A-17
executing for a KSAM file, A-17
executing for a TurboIMAGE data set, A-17
p-code, 9-7
as input for Transact/iX, 9-16
PDEPTH option, SET verb, 8-218
PERFORM option
DELETE verb, 8-31
FIND verb, 8-55
OUTPUT verb, 8-148
REPLACE verb, 8-195
PERFORM verb, 8-157
period edit character, 8-37, 8-66
PLINE variable, 3-2
LET verb, 8-93
POSITION function, LET verb, 8-100

POSITION parameter, PROC verb, 8-161
precedence, rules of, 8-94
PRIMARY modifier
  DELETE verb, 8-27
  FIND verb, 8-51
  GET verb, 8-73
  OUTPUT verb, 8-145
  REPLACE verb, 8-192
PRINT command, 11-71
PRINT command qualifier, 5-6
PRINT option
  REPEAT verb, 8-187
  RESET verb, 8-201
  SET verb, 8-218
PRINT variable
  IF verb, 8-84
  WHILE verb, 8-238
PROCALIGNED_16/32/64 compiler options, 9-17
processing command sequences, 5-4
processor
  bypassing prompt, 9-14
  input and output destinations, 9-14
  redirecting VPLUS form output, 9-15
  test mode output, 9-15
  TRANDUMP, 9-15
  TRANIN, 9-14
  TRANLIST, 9-15
  TRANOUT, 9-15
  TRANSORT, 9-14, 9-21
  TRANVPLS, 9-15
processor command qualifiers
  FIELD, 5-6
  PRINT, 5-6
  REPEAT, 5-6
  SORT, 5-7
  TPRINT, 5-7
processor commands, 5-6
  COMMAND, 5-6
  EXIT, 5-6
  INITIALIZE, 5-6
  RESUME, 5-6
  TEST, 5-6
processor time optimization, C-33
PROCINTRINSIC compiler option, 9-18
PROCTIME option, LIST verb, 8-117
PROC verb, 8-158
program
  compilation, 9-7
  optimization, C-33
  overlays, 9-5
  segmentation, 9-5
prompting for data, 8-12
PROMPT option, SET verb, 8-218
PROMPT verb, 4-7, 8-171
PROPER function, MOVE verb, 8-135
PUT(FORM) verb, executing on a VPLUS form, A-19
PUT verb, 8-177, A-18
  executing for a KSAM or MPE file, A-18
  executing for a TurboIMAGE data set, A-18
PWIDTH option, SET verb, 8-218

R

RCHAIN modifier
  DELETE verb, 8-28
  FIND verb, 8-51
  GET verb, 8-73
  OUTPUT verb, 8-145
  REPLACE verb, 8-192
READ modifier, FILE verb, 8-48
real arithmetic, 64-bit, 8-109
real numbers, 3-4
RECNO option
  DELETE verb, 8-31
  FIND verb, 8-55
  GET verb, 8-76
  OUTPUT verb, 8-148
  PUT verb, 8-181
  REPLACE verb, 8-195
registers, 4-1
  argument register, 4-3
  data register, 4-2
  example, 4-9
  input register, 4-6
  in segmented programs, 9-6
  key register, 4-3
  list register, 4-2
  match register, 4-5
  setting values to, 8-12
  status register, 4-7
  update register, 4-6
  verb modifier summary, 4-7
  write-only, 4-3
relational operators, 5-8
REPEAT command, 11-73
REPEAT command qualifier, 5-6
REPEAT option, SET verb, 8-218
REPEAT variable
  IF verb, 8-84
  REPEAT verb, 8-187
  WHILE verb, 8-238
REPEAT verb, 8-186
REPLACE verb, 8-190, A-21
  executing for a KSAM file, A-22
  executing for an MPE file, A-23
  executing for a TurboIMAGE data set, A-21
Report/V option, CALL verb, 8-6
reserved file names, 9-6
reserved system variables, 3-2

reserved words, 2-8
RESET(LANGUAGE) statement, E-1
RESET verb, 8-199
responses, user
!, 8-83, 8-186, 8-237
], 8-218
]], 8-218
RESTART, 8-44
RESUME processor command, 5-6
RETURN verb, 8-157, 8-205
RIGHT option
DATA verb, 8-14
DISPLAY verb, 8-41
FORMAT verb, 8-70
PROMPT verb, 8-172
SET verb, 8-219
rounding, 8-107
ROW option
DISPLAY verb, 8-41
FORMAT verb, 8-70
RSERIAL modifier
DELETE verb, 8-28
FIND verb, 8-52
GET verb, 8-73
OUTPUT verb, 8-145
REPLACE verb, 8-192
RSOM file, 9-16
rules of precedence, 8-94
RUN command, 9-7, 9-10, 9-13
running Transact, 9-1, 9-13
RUN progname command, 9-30
run-time binding of data items, B-4
run-time environment, TRANDEBUG, 11-12
run-time features
not supported by Transact/iX, B-4
supported by Transact/iX, B-4
run-time stack, C-2
RUN TRAN.PUB.SYS command, 9-23

S

S and SS edit characters, 8-38, 8-68
SCAN option
DATA verb, 8-15
LIST verb, 8-118
PROMPT verb, 8-174
!SCOPE System Dictionary command, 9-3
!SEGMENT compiler command, 9-2
segmented programs, 9-5
compiler command, 9-2
selection criteria
MATCH prompt, 5-15
match register, 4-5
run time, 5-7
semicolon, with DOEND, 2-5
SEQUENCE modifier, END verb, 8-44
SERIAL modifier
DELETE verb, 8-28
FIND verb, 8-52
GET verb, 8-73
OUTPUT verb, 8-145
REPLACE verb, 8-192
SESSION option, LIST verb, 8-117
!SET compiler command, 9-2
SET(FORM) verb, executing for a VPLUS form, A-24
SET(LANGUAGE) statement, E-1
SETLNG parameter, PROC verb, 8-161
SET modifier
DATA verb, 8-16
PROMPT verb, 8-174
SET parameter, PROC verb, 8-161
setting a breakpoint, 11-21
at a data item, 11-30
at a data item value, 11-30
at a label, 11-53
at a register, 11-28
setting values to registers, 8-12
SET(UPDATE), parent and child values, B-6
SET verb, 8-207, A-24
SHOW option, DATA verb, 8-16
SIGNON option, SYSTEM verb, 8-228
SINGLE option
DELETE verb, 8-31
FIND verb, 8-55
OUTPUT verb, 8-149
REPLACE verb, 8-195
SIZE option, CALL verb, 8-4
SIZE parameter, PROC verb, 8-161
SL routines, calling, 8-158
SOPT option
DELETE verb, 8-31
FIND verb, 8-55
OUTPUT verb, 8-149
REPLACE verb, 8-195
SORT command, 11-74
SORT command qualifier, 5-7
SORT modifier, FILE verb, 8-48
SORT option
FIND verb, 8-56
OUTPUT verb, 8-149
RESET verb, 8-201
SET verb, 8-219
SYSTEM verb, 8-228
SORT variable
IF verb, 8-84
REPEAT verb, 8-187
WHILE verb, 8-238
source code formatting, 2-6
source program migration, B-7
SPACE function, MOVE verb, 8-137

SPACE option
DISPLAY verb, 8-41
FORMAT verb, 8-70
special characters, 5-14
special characters as selection criteria, 5-15
specifying language for the compiler and processor, E-2
SQRT function, LET verb, 8-101
STACK modifier
RESET verb, 8-202
SET verb, 8-221
STAT compiler option, 9-10, C-6
statement labels, 2-4
statements, 2-4
compound, 2-5
formatting, 2-6
static calls, 8-2
compiling programs for, 9-19
STATUS option
CALL verb, 8-5
CLOSE verb, 8-11
database and file operation verbs, 7-6
data entry verbs, 7-5
DATA verb, 8-14
DELETE verb, 8-32
FIND verb, 8-56
GET verb, 8-76, 8-78
INPUT verb, 8-90
LOGTRAN verb, 8-125
OUTPUT verb, 8-149
PATH verb, 8-155
PROMPT verb, 8-172
PUT verb, 8-180
REPLACE verb, 8-195
UPDATE verb, 8-233
STATUS parameter, PROC verb, 8-161
status register, 4-7
testing with IF, 7-3
STATUS variable, 3-2
IF verb, 8-84
LET verb, 8-94
REPEAT verb, 8-187
WHILE verb, 8-238
$STDINX, 9-11
$STDLIST, 9-12
STEP command, 11-77
storage registers, 4-1
streamed batch job, 9-11
STRING function, MOVE verb, 8-138
string functions, MOVE verb, 8-131
$subcommand, 5-4
subcommand labels, 5-4
SUBPROGRAM compiler option, 9-18
SUPPRESS option
RESET verb, 8-201
SET verb, 8-219
SWAP option, CALL verb, 8-5, C-28
synonym, 8-24, 8-27, 8-51, 8-73, 8-145, 8-192
syntax options, defined, 8-1
!SYSDIC System Dictionary command, 9-3
System Dictionary, 3-9
System Dictionary commands, 9-3
!DOMAIN, 9-3
!NOSYSDIC, 9-3
!SCOPE, 9-3
!SYSDIC, 9-3
!VERSION, 9-3
!VERSIONSTATUS, 9-3
system error messages, 7-10
system errors, causes of, 7-10
SYSTEM NAME prompt, 9-13
SYSTEM statement, 2-3
access mode, 6-2
system variables, 3-2
SYSTEM verb, 2-3, 8-223
WORKFILE option, 8-229

T

TABLE modifier, DISPLAY verb, 8-34
TABLE option, SET verb, 8-219
target, 2-5
TDBIGINIT file, TRANDEBUG, 11-11
T edit character, 8-38, 8-68
TERMID option, LIST verb, 8-117
terminating TRANDEBUG, 11-15
TEST command processor, 5-6
TEST command test facility, 10-1
test modes, 10-4
output, 9-15
under MPE/iX, B-4
$TIME edit characters, 8-38
TIME option, LIST verb, 8-117
time out for terminal input, 8-48
TIMER option, LIST verb, 8-117
$TIME variable, 3-2, 8-35
time variable, 3-2
TITLE option
DISPLAY verb, 8-41
FORMAT verb, 8-70
TLINE variable, 3-2
LET verb, 8-94
$TODAY edit characters, 8-39
$TODAY variable, 3-2, 8-35
TPRINT command, 11-79
TPRINT command qualifier, 5-7
TPRINT option
RESET verb, 8-201
SET verb, 8-219
TRACE command, 11-81
TRAILER option

DATA verb, 8-15
LIST verb, 8-118
PROMPT verb, 8-174
TRANCODE, 9-12
TRANDBMODE, 9-21
TRANDEBUG, 9-21, 11-1
accessing MPE/iX command interpreter, 11-14
alternative debug entry points, 11-12
arithmetic traps, 11-13
compatibility, 11-3
compiling with TRANDEBUG, 11-4
continuing program execution, 11-8
Ctrl-Y, 11-8
debugging VPLUS applications, 11-11
deleting a breakpoint, 11-18
disabling, 11-12
displaying contents of input register, 11-40
displaying data items, 11-9
displaying information about a database, 11-36
displaying information about an item, 11-35
displaying information about specific MPE/KSAM files, 11-39
displaying source code, 11-5
displaying the CALL stack, 11-37
displaying values of items in data register, 11-41
ending a session, 11-5
features and benefits, 11-1
listing breakpoints, 11-20
listing data breakpoints, 11-27
logging commands, 11-33
modifying data items, 11-9
redirecting VPLUS input and output, 11-11
repeating last command, 11-16
run-time environment, 11-12
setting a breakpoint, 11-21, 11-30
setting a breakpoint at a label, 11-53
setting a breakpoint at a register, 11-28
source code window, 11-5
starting a session, 11-5
startup file, 11-11
stepping through program, 11-9
terminating execution, 11-15
tutorial, 11-4
TRANDEBUG commands
: , 11-14
ABORT, 11-15
AUTORPT, 11-16
BREAK DELETE, 11-18
BREAK LIST, 11-20
BREAK SET, 11-21
CONTINUE, 11-24
DATA BREAK DELETE, 11-25
DATA BREAK LIST, 11-27
DATA BREAK REGISTER, 11-28
DATA BREAK SET, 11-30
DATA LOG, 11-33
DEFN, 11-35
DISPLAY BASE, 11-36
DISPLAY CALLS, 11-37
DISPLAY COMAREA, 11-38
DISPLAY FILE, 11-39
DISPLAY INPUT, 11-40
DISPLAY ITEM, 11-41
DISPLAY KEY, 11-43
DISPLAY MATCH, 11-44
DISPLAY PERFORM, 11-45
DISPLAY STATUS, 11-46
DISPLAY STATUSDB, 11-47
DISPLAY STATUSIN, 11-48
DISPLAY UPDATE, 11-49
EDIT, 11-50
HELP, 11-51
LABEL BREAK SET, 11-53
LABEL JUMP, 11-55
LOC, 11-56
LOG, 11-57
MODIFY INPUT, 11-59
MODIFY ITEM, 11-60
MODIFY KEY, 11-61
MODIFY MATCH, 11-62
MODIFY STATUS, 11-64
MODIFY UPDATE, 11-65
NMDEBUG, 11-67
PAGE BACK, 11-68
PAGE FORWARD, 11-69
PAGE JUMP, 11-70
PRINT, 11-71
REPEAT, 11-73
SORT, 11-74
STEP, 11-77
TPRINT, 11-79
TRACE, 11-81
USE, 11-83
VERSION, 11-85
WINDOW LENGTH, 11-86
WINDOW OFF, 11-87
WINDOW ON, 11-88
TRANDEBUG compiler option, 9-18
TRANDUMP, 9-15
TRANIN file designator, 9-11, 9-14
TRANLIST file designator, 9-12, 9-15
TRANOUT file designator, 9-2, 9-12, 9-15
Transact
error handling, 7-1
interpreting programs, 9-13
test facility, 10-1
transaction logging, 8-122

Transaction Manager (XM), 6-9
Transact/iX
alignment, 9-17
binding data item attributes, B-4
calling subprograms from COBOL, D-1
calling subprograms from Pascal, D-1
compiler options, 9-17
compiling programs, 9-16, 9-22
double buffering parameters, 8-165
executing programs, 9-16, 9-22
features, B-2
INITIALIZE command, B-5
migrating to, B-1
null parameters, 8-164
option variable procedures, 8-164
test modes not supported, B-4
TRANCOMP options used, 9-19
unsupported run-time features, B-4
Transact processor
error messages, 7-9
information messages, 7-9
Transact/V
features, B-4
migrating from, B-1
TRANSORT, 9-14, 9-21
TRANTEXT, 9-11
TRANVPLS, 9-15
TRANXL command, 9-25
TRANXLGO command, 9-27
TRANXLLK command, 9-26
TRUNCATE option
DISPLAY verb, 8-41
FORMAT verb, 8-70
truncation, 8-107
TurboIMAGE
dynamic roll-back, 6-9
Transaction Manager, 6-9
TYPE parameter, PROC verb, 8-162

U

UNLOAD option, PROC verb, 8-162
UPDATE(FORM) verb, executing for a VPLUS form, A-28
UPDATE modifier
DATA verb, 8-17
FILE verb, 8-48
LIST verb, 8-118
PROMPT verb, 8-175
SET verb, 8-222
UPDATE option
REPLACE verb, 8-196
RESET verb, 8-202
update register, 4-6
parent and child values, B-6
UPDATE verb, 8-230, A-26
executing for a KSAM file, A-26
executing for an MPE file, A-27
executing for a TurboIMAGE data set, A-26
uppercase alphanumeric string, 3-4
UPPER function, MOVE verb, 8-140
upshift, 4-6
USE command, 11-83
USER option, LIST verb, 8-117
user responses
!, 5-14, 8-83, 8-186, 8-237
], 5-14, 8-218
]], 5-14, 8-218

V

VALUE function, LET verb, 8-102
VCOM parameter, PROC verb, 8-162
verbs, 2-4
CALL, 8-2
CLOSE, 8-10
DATA, 4-7, 8-12
DEFINE, 8-19
DELETE, 8-27
DISPLAY, 8-34
END, 8-44
EXIT, 8-46
FILE, 8-47
FIND, 8-51
FORMAT, 8-64
GET, 8-72
GO TO, 8-82
IF, 8-83
INPUT, 4-7, 8-90
ITEM, 8-92
LET, 8-93
LEVEL, 8-113
LIST, 4-7, 8-115
LOGTRAN, 8-122
MOVE, 8-128
OUTPUT, 8-144
PATH, 8-152
PERFORM, 8-157
PROC, 8-158
PROMPT, 4-7, 8-171
PUT, 8-177
REPEAT, 8-186
REPLACE, 8-190
RESET, 8-199
RETURN, 8-205
SET, 8-207
SYSTEM, 2-3, 8-223
UPDATE, 8-230
WHILE, 8-237
VERSION command, 11-85
!VERSIONSTATUS System Dictionary command, 9-3

!VERSION System Dictionary command, 9-3
VPLS option
  RESET verb, 8-202
  SET verb, 8-219
  SYSTEM verb, 8-228
VPLUS
  closing forms file, 8-10
  forms, 5-16
  GET(FORM), 5-16, 8-72
  local form storage, 5-11
  PUT(FORM), 5-16, 8-177
  SET(FORM), 8-208
  special keys, 5-16
  SYSTEM verb, 8-228
  TRANVPLS file, 9-15
  UPDATE(FORM), 8-231
  VCLOSETERM, 8-220
  VOPENTERM, 8-220
VPLUS interface, 5-11

W

WAIT option
  PUT verb, 8-182
  UPDATE verb, 8-235
warning messages, 7-8
WHILE verb, 8-237
WIDTH option, SET verb, 8-220
WINDOW LENGTH command, 11-86
WINDOW OFF command, 11-87
WINDOW ON command, 11-88
WINDOW option
  GET verb, 8-78
  PUT verb, 8-183
  SET verb, 8-211
  UPDATE verb, 8-235
WORKFILE option
  FIND verb, 8-57
  SYSTEM verb, 8-229
WORK option, SYSTEM verb, 8-229
WRITE modifier, FILE verb, 8-49
write-only registers, 4-3

X

XERR compiler option, 9-10
XREF compiler option, 9-10

Y

YY and YYYY edit characters, 8-39, 8-69

Z

ZD edit characters, 8-39, 8-68
Z edit character, 8-36, 8-66
ZERO[E]S option
  DISPLAY verb, 8-41
  FORMAT verb, 8-70
  SET verb, 8-220
ZH edit characters, 8-38, 8-67
ZM edit characters, 8-38, 8-39, 8-67, 8-68
zoned decimal, 3-4
ZS edit characters, 8-38, 8-68