INTOUCH® 4GL
A Guide to the INTOUCH Language

This document contains PROPRIETARY and CONFIDENTIAL information. Such information may not be disclosed for any purpose without written permission from:

Touch Technologies, Inc.
9988 Hibert Street, Suite 310
San Diego, CA 92131

1-800-525-2527 or 1-619-566-3603

® VAX is a registered trademark of Digital Equipment Corporation.
® INTOUCH is a registered trademark of Touch Technologies, Inc.

NOTICE

Touch Technologies, Inc. (TTI) has prepared this publication for use by TTI personnel, licensees, and customers. This information is protected by copyright. No part of this document may be photocopied, reproduced or translated to another language without prior written consent of Touch Technologies Incorporated.

TTI believes the information described in this publication is accurate and reliable; much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material.

The information contained herein is subject to change without notice and should not be construed as a commitment by Touch Technologies, Inc.

The following are trademarks of Touch Technologies, Inc., and may be used only to describe products of Touch Technologies, Inc.:

DYNAMIC TAPE ACCELERATOR INTOUCH CleanDisk

DYNAMIC LOAD BALANCER PLUS REMOTE DEVICE FACILITY

The following are trademarks of Digital Equipment Corporation, and may be used only to describe products of Digital Equipment Corporation:

DBMS DCL DECNET RDB RMS VAX OpenVMS

Last Revision date: 20-Oct-1995

Copyright ©1992, 1995 Touch Technologies, Inc.

Contents

PREFACE xxi

CHAPTER 1 WORKING IN THE INTOUCH ENVIRONMENT 1–1

1.1 GETTING STARTED 1–1

1.2 ACCESSING THE VMS SYSTEM UTILITIES 1–2

1.3 RECALLING PREVIOUS COMMANDS 1–3

1.4 GETTING HELP WHEN IN INTOUCH 1–3

1.5 USING INTOUCH–SOME EXAMPLES 1–4
1.5.1 Writing an INTOUCH program 1–5
1.5.2 Running an INTOUCH program 1–5
1.5.3 Interrupting and Changing an INTOUCH program 1–6

1.6 USING EXIT AND BACK 1–8

1.7 MESSAGES IN INTOUCH 1–10

1.8 USING THE TAB KEY FEATURES 1–10
1.8.1 Tab Key Indenting 1–10
1.8.2 Command Completion 1–11
1.8.3 Spelling Correction 1–11
1.8.4 Expanding Abbreviations 1–12

1.9 OTHER INTOUCH FEATURES 1–12

Contents

CHAPTER 2 INTOUCH COMMANDS 2–1

2.1 PRESENTATION OF COMMAND AND STATEMENT INFORMATION 2–1

2.2 ABOUT INTOUCH COMMANDS 2–2

2.3 INTOUCH COMMANDS 2–3
2.3.1 NEW 2–3
2.3.2 SAVE 2–4
2.3.3 REPLACE 2–5
2.3.4 ABORT 2–5
2.3.5 RUN and RNH 2–7
2.3.6 OLD 2–8
2.3.7 LIST 2–9
2.3.8 LIST ERROR 2–10
2.3.9 EDIT 2–10
2.3.10 RENUMBER 2–11
2.3.11 FRAME ON | OFF 2–12
2.3.12 REFRESH 2–12
2.3.13 IDE 2–12
2.3.14 COMPILE 2–13
2.3.14.1 Compiled Images • 2–14
2.3.14.2 Running Compiled Images • 2–14
2.3.15 Developing INTOUCH Programs 2–15
2.3.16 GO 2–15
2.3.17 UNSAVE 2–16

CHAPTER 3 PROGRAM ELEMENTS AND DATA IN INTOUCH 3–1

3.1 INTOUCH PROGRAM ELEMENTS 3–1
3.1.1 Continuing Program Lines 3–3
3.1.2 Labels 3–4
3.1.3 Storing INTOUCH Programs 3–4

3.2 DATA IN INTOUCH 3–4
3.2.1 Data Types 3–4
3.2.1.1 Integers • 3–5
3.2.1.2 Real Numbers • 3–5
3.2.1.3 String Data • 3–6

3.2.2 BOOLEAN 3–6
3.2.3 Expressions 3–6
3.2.4 Constants 3–7
3.2.4.1 Numeric constants • 3–7
3.2.4.2 E Notation • 3–7
3.2.4.3 String Constants • 3–8
3.2.5 Variables 3–9
3.2.5.1 Arrays • 3–10
3.2.5.2 Substrings • 3–11
3.2.6 Structure References 3–11
3.2.7 Multiple Occurrence Fields 3–13
3.2.8 Compound Expressions 3–14
3.2.8.1 Numeric Expressions • 3–14
3.2.8.2 String Expressions • 3–15
3.2.8.3 Conditional Expressions • 3–15
3.2.9 Order of Evaluation 3–18

CHAPTER 4 WRITING AND DEBUGGING INTOUCH PROGRAMS 4–1

4.1 COMMENTS 4–1
4.1.1 REM 4–1
4.1.2 ! comment_text 4–2
4.1.3 ~ Used to Border Part of a Program 4–3

4.2 PROGRAM, END, STOP AND HALT STATEMENTS 4–4
4.2.1 PROGRAM 4–4
4.2.2 END 4–4
4.2.3 STOP 4–5
4.2.4 HALT 4–6
4.2.5 RANDOMIZE 4–7

4.3 DEBUG FACILITIES 4–8

4.4 DEBUG ON/OFF 4–8
4.4.1 DEBUG ON 4–8
4.4.2 DEBUG OFF 4–9

4.5 MONITORING PROGRAM EXECUTION 4–9

4.6 THE TRACE AND STATISTICS FEATURES 4–10

4.6.1 TRACE ON 4–10
4.6.2 TRACE OFF 4–12
4.6.3 STATS ON 4–12
4.6.4 STATS OFF 4–13
4.6.5 LIST STATS 4–13

4.7 BREAK AND STEP 4–14
4.7.1 BREAK 4–14
4.7.2 STEP 4–15

4.8 GO–CONTINUING PROGRAM EXECUTION 4–16
4.8.1 GO 4–16

CHAPTER 5 INPUTTING AND STORING TERMINAL DATA 5–1

5.1 INPUT STATEMENT 5–1
5.1.1 Types of INPUT Statements 5–2
5.1.2 INPUT Styles 5–2

5.2 INPUT OPTIONS 5–3

5.3 INPUT COMMANDS AND KEYSTROKES AVAILABLE TO THE USER 5–4
5.3.1 Editing Input 5–4

5.4 INPUTTING STRINGS 5–5

5.5 INPUTTING MULTIPLE VARIABLES 5–5
5.5.1 Simple Input Statement 5–5
5.5.2 LINE INPUT Statement 5–6

5.6 INPUT DEFAULT PROMPT AND TEXT 5–6

5.7 PROMPT OPTION 5–7

5.8 AT OPTION 5–7

5.9 ATTRIBUTES OPTION 5–8

5.10 LENGTH OPTION 5–9

5.11 DEFAULT OPTION 5–9
5.11.1 Pre-positioning to a DEFAULT Menu Item 5–10

5.12 ERASE OPTION 5–11

5.13 VALID OPTION 5–11

5.14 TIMEOUT OPTION 5–12

5.15 AREA OPTION 5–12

5.16 SCREEN OPTION 5–13
5.16.1 Screen Format Characters 5–14
5.16.2 Screen Format Options 5–16
5.16.3 The User's Response to the SCREEN Option 5–19

5.17 MENU OPTION 5–20
5.17.1 Items 5–20
5.17.1.1 Descriptions • 5–21
5.17.1.2 Data Value • 5–21
5.17.2 Submenu Indicator 5–21
5.17.3 Menu DIRECTIVES: 5–21
5.17.4 User Interface 5–27

5.18 KEY INPUT 5–28

CHAPTER 6 PRINTING AND DISPLAYING DATA 6–1

6.1 PRINT 6–1
6.1.1 Printing Multiple Expressions 6–2
6.1.2 TAB Function 6–3
6.1.3 AT Option 6–4

6.1.4 ERASE Option 6–4
6.1.5 WIDE Option 6–5
6.1.6 Printing Numbers 6–5
6.1.7 Positioning the Cursor for the Next Print Statement 6–6
6.1.8 Printing Attributes–Highlighting Options 6–6
6.1.9 USING Option 6–7
6.1.10 Fields 6–7
6.1.10.1 String Format Characters • 6–8
6.1.10.2 Numeric Format Characters • 6–8
6.1.11 Directives 6–13
6.1.11.1 {UCASE}? • 6–14
6.1.11.2 {LCASE}? • 6–14
6.1.11.3 {HYPHEN} • 6–14
6.1.11.4 {DATE [argument]}? • 6–14
6.1.11.5 {ROTATE n}? • 6–15
6.1.11.6 {TIME}? • 6–15
6.1.11.7 {ZIPCODE}? • 6–16

6.2 USING THE INTOUCH SCREEN 6–16

6.3 MESSAGE 6–16

6.4 DELAY 6–18

6.5 CLEAR 6–19

6.6 OPTION COMMUNICATE ON | OFF 6–22

CHAPTER 7 PROGRAM DIRECTIVES 7–1

7.1 COMPILE DIRECTIVES 7–1
7.1.1 %MESSAGE 7–1
7.1.2 %MESSAGE ERROR 7–1
7.1.3 %INCLUDE 7–2
7.1.4 %INCLUDE CONDITIONAL 7–2
7.1.5 %DEBUG 7–3

CHAPTER 8 ASSIGNING VARIABLES 8–1

8.1 DECLARE 8–1
8.1.1 DECLARE STRUCTURE 8–2
8.1.2 DECLARE PREFIX 8–3

8.2 OPTION STATEMENTS 8–5
8.2.1 OPTION REQUIRE DECLARE 8–5
8.2.2 OPTION BASE 8–5

8.3 LET 8–6

8.4 LSET, RSET AND CSET 8–7
8.4.1 LSET 8–8
8.4.2 RSET 8–8
8.4.3 CSET 8–9

8.5 LSET, RSET, CSET FILL 8–10

8.6 DATA, READ, RESTORE STATEMENTS 8–10
8.6.1 READ 8–10
8.6.2 RESTORE 8–12

CHAPTER 9 SET AND ASK STATEMENTS 9–1

9.1 SET AUTOEXIT 9–1

9.2 SET BACK ON | OFF 9–2
9.2.1 SET BACK ON 9–2
9.2.2 SET BACK OFF 9–2

9.3 SET ERROR ON | OFF 9–3
9.3.1 SET ERROR ON 9–3
9.3.2 SET ERROR OFF 9–4

9.4 ASK ERRORS 9–4

Contents

9.5 SET EXIT ON | OFF 9–5
9.5.1 SET EXIT ON 9–5
9.5.2 SET EXIT OFF 9–6

9.6 SET HELP ON | OFF 9–6
9.6.1 SET HELP ON 9–6
9.6.2 SET HELP OFF 9–7

9.7 ASK KEYSTROKES 9–7

9.8 ASK | SET MARGIN 9–8
9.8.1 ASK MARGIN 9–8
9.8.2 SET MARGIN 9–8

9.9 ASK | SET MESSAGELINE 9–9
9.9.1 ASK MESSAGELINE 9–9
9.9.2 SET MESSAGELINE 9–9

9.10 ASK PAGESIZE 9–10

9.11 SET PORT ON | OFF 9–10
9.11.1 SET PORT ON 9–10
9.11.2 SET PORT OFF 9–11

9.12 ASK RESPONSES 9–12

9.13 SET SCROLL 9–12

9.14 ASK | SET SEED 9–13

9.15 ASK | SET SYSTEM 9–14
9.15.1 ASK SYSTEM: COMMENT 9–14
9.15.2 SET SYSTEM: COMMENT 9–15
9.15.3 ASK SYSTEM: DIRECTORY 9–15
9.15.4 SET SYSTEM: DIRECTORY 9–16
9.15.5 ASK SYSTEM, LOGICAL: VALUE 9–16
9.15.6 SET SYSTEM, LOGICAL: VALUE 9–17

9.15.7 ASK SYSTEM: MODE 9–17
9.15.8 ASK SYSTEM: PARAMETER 9–18
9.15.9 ASK SYSTEM, PASS: SUCCESS 9–18
9.15.10 ASK SYSTEM: PROCESS 9–19
9.15.11 SET SYSTEM: PROCESS 9–19
9.15.12 ASK SYSTEM: RIGHTS 9–20
9.15.13 ASK SYSTEM, SYMBOL: VALUE 9–20
9.15.14 SET SYSTEM, SYMBOL: VALUE 9–21
9.15.15 ASK SYSTEM: USER 9–21

9.16 ASK | SET WINDOW 9–22
9.16.1 ASK WINDOW AREA 9–22
9.16.2 SET WINDOW AREA 9–22
9.16.3 ASK WINDOW: COLUMN 9–23
9.16.4 SET WINDOW: COLUMN 9–23
9.16.5 ASK | SET WINDOW: CURRENT 9–24
9.16.6 ASK WINDOW: DATA 9–25
9.16.7 SET WINDOW: DATA 9–25
9.16.8 ASK | SET WINDOW: KEYMAP 9–26
9.16.9 SET WINDOW KEYSTROKE: VALUE 9–27
9.16.10 ASK WINDOW: ROW 9–28
9.16.11 SET WINDOW: ROW 9–29
9.16.12 ASK WINDOW: TYPEAHEAD 9–29
9.16.13 SET WINDOW: TYPEAHEAD 9–30

9.17 ASK | SET ZONEWIDTH 9–30
9.17.1 ASK ZONEWIDTH 9–30
9.17.2 SET ZONEWIDTH 9–31

CHAPTER 10 DEFINING ARRAYS 10–1

10.1 DIMENSIONING ARRAYS 10–1
10.1.1 DIM 10–1
10.1.2 REDIM 10–3
10.1.3 OPTION BASE 10–4

Contents

CHAPTER 11 LOOPS, CONDITIONALS, CHAINING AND BRANCHING 11–1

11.1 FOR/NEXT LOOP 11–1
11.1.1 The STEP Option 11–3
11.1.2 Nesting Loops 11–4
11.1.3 EXIT FOR 11–4
11.1.4 REPEAT FOR 11–5
11.1.5 ITERATE FOR 11–6

11.2 DO LOOP 11–7
11.2.1 WHILE and UNTIL Options 11–8
11.2.2 EXIT DO 11–10
11.2.3 REPEAT DO 11–11
11.2.4 ITERATE DO 11–12

11.3 EXECUTE 11–13

11.4 CONDITIONALS 11–14

11.5 IF/THEN ... END IF 11–14
11.5.1 ELSE Option 11–16
11.5.2 ELSEIF Option 11–17

11.6 SELECT CASE/END SELECT 11–18
11.6.1 CASE expr, expr, expr... 11–19
11.6.2 CASE ELSE Option 11–20
11.6.3 CASE IS Option 11–21

11.7 CHAIN PROGRAMS 11–21

11.8 PASS COMMANDS TO THE OPERATING SYSTEM 11–22
11.8.1 PASS [NOWAIT: | NORETURN:] 11–22

11.9 BRANCHING 11–24
11.9.1 Branching and Code Blocks 11–24

11.10 GOTO 11–24

11.10.1 ON...GOTO...[ELSE] 11–25
11.10.1.1 ELSE Option • 11–26

11.11 GOSUB/RETURN 11–27
11.11.1 GOSUB 11–27
11.11.2 ON...GOSUB 11–28
11.11.2.1 ELSE Option • 11–30

11.12 DISPATCH 11–30

11.13 ROUTINE/END ROUTINE 11–31
11.13.1 EXIT ROUTINE 11–31
11.13.2 REPEAT ROUTINE 11–32

CHAPTER 12 EXCEPTION HANDLING 12–1

12.1 CAUSE EXCEPTION 12–1

12.2 WHEN EXCEPTION IN 12–2

12.3 WHEN EXCEPTION USE 12–3

12.4 HANDLER ROUTINE ACTIONS 12–5
12.4.1 RETRY 12–5
12.4.2 CONTINUE 12–6
12.4.3 RESUME 12–7
12.4.4 EXIT HANDLER 12–8

CHAPTER 13 CALLING ROUTINES WRITTEN IN OTHER LANGUAGES 13–1

13.1 LIBRARY 13–1

13.2 CALL 13–2
13.2.1 Optional Arguments 13–2
13.2.2 Passing Mechanisms 13–3

CHAPTER 14 DATA STRUCTURE STATEMENTS 14–1

14.1 GENERAL INFORMATION 14–1

14.2 OPEN STRUCTURE 14–3
14.2.1 ACCESS Option 14–4
14.2.2 LOCK Option 14–4
14.2.3 DATAFILE Option 14–4

14.3 CLOSE STRUCTURE STRUC_NAME 14–4

14.4 ADD STRUCTURE 14–5
14.4.1 CANCEL ADD 14–6
14.4.2 EXIT ADD 14–7

14.5 DELETE STRUCTURE 14–8

14.6 LOCK | UNLOCK STRUCTURE 14–8

14.7 EXTRACT STRUCTURE 14–9
14.7.1 INCLUDE Statement 14–11
14.7.2 EXCLUDE Statement 14–12
14.7.3 SORT Statement 14–12
14.7.4 FOR EACH/NEXT 14–14
14.7.5 EXTRACT STRUCTURE: KEY 14–15
14.7.6 EXTRACT STRUCTURE, FIELD: PARTIAL KEY 14–16
14.7.7 CANCEL EXTRACT 14–17
14.7.8 EXIT EXTRACT 14–18
14.7.9 REEXTRACT STRUCTURE 14–18
14.7.10 EXTRACT STRUCTURE: APPEND 14–19
14.7.11 EXTRACT STRUCTURE, SET, USING 14–20
14.7.12 EXTRACT STRUCTURE, SET, FIELD: KEY 14–21

14.8 ASK STRUCTURE 14–21
14.8.1 ASK STRUCTURE FIELD: item 14–22
14.8.1.1 FIELD Expressions • 14–23
14.8.1.2 Item: ACCESS • 14–24
14.8.1.3 Item: APPLICATION • 14–25
14.8.1.4 Item: ATTRIBUTES • 14–25
14.8.1.5 Item: CHANGEABLE • 14–25
14.8.1.6 Item: CLASSIFICATION • 14–26

14.8.1.7 Item: DATATYPE • 14–26
14.8.1.8 Item: DESCRIPTION • 14–26
14.8.1.9 Item: HEADING • 14–26
14.8.1.10 Item: HELP • 14–27
14.8.1.11 Item: KEYED • 14–27
14.8.1.12 Item: LENGTH • 14–27
14.8.1.13 Item: NAME • 14–27
14.8.1.14 Item: NULL • 14–27
14.8.1.15 Item: NUMBER • 14–28
14.8.1.16 Item: OPTIMIZED • 14–28
14.8.1.17 Item: POSITION • 14–28
14.8.1.18 Item: PRINTMASK • 14–28
14.8.1.19 Item: PROMPT • 14–29
14.8.1.20 Item: SCREENMASK • 14–29
14.8.1.21 Item: VRULES • 14–29

14.8.2 ASK STRUCTURE: CURRENT — 14–29
14.8.3 ASK STRUCTURE: FIELDS — 14–30
14.8.4 ASK STRUCTURE: KEYS — 14–31
14.8.5 ASK STRUCTURE: CAPABILITY — 14–31
14.8.6 ASK STRUCTURE: EXTRACTED — 14–32
14.8.7 ASK STRUCTURE: ID — 14–33
14.8.8 ASK STRUCTURE: POINTER — 14–33
14.8.9 ASK STRUCTURE: RECORD — 14–34
14.8.10 ASK STRUCTURE: RECORDSIZE — 14–35
14.8.11 ASK STRUCTURE: ACCESS — 14–35
14.8.12 ASK \|SET STRUCTURE: TIMEOUT — 14–36
14.8.13 ASK \| SET STRUCTURE #string_expr . . . — 14–36
14.8.14 ASK STRUCTURE: ENGINE — 14–37

14.9 SET STRUCTURE 14–37
14.9.1 SET STRUCTURE: CURRENT — 14–37
14.9.2 SET STRUCTURE, FIELD: KEY — 14–39
14.9.3 SET STRUCTURE, FIELD: PARTIAL KEY — 14–39
14.9.4 SET STRUCTURE: ID — 14–40
14.9.5 SET STRUCTURE: POINTER — 14–41
14.9.6 SET STRUCTURE: RECORD — 14–41
14.9.7 SET STRUCTURE: EXTRACTED 0 — 14–42
14.9.8 SET STRUCTURE, SET, USING: EXPRESSION 'owner' — 14–42
14.9.9 SET STRUCTURE, SET: EXPRESSION 'owner' — 14–43
14.9.10 SET STRUCTURE, SET, USING, FIELD: KEY — 14–43
14.9.11 SET STRUCTURE, SET, USING — 14–43

Contents

CHAPTER 15 FILE HANDLING 15–1

15.1 OPEN #CHNL_NUM: NAME ... 15–1
15.1.1 OPEN Options 15–2
15.1.1.1 ACCESS Option • 15–2
15.1.1.2 UNFORMATTED Option • 15–3
15.1.1.3 OPTIMIZE OFF Option • 15–3
15.1.1.4 UNIQUE Option • 15–3

15.2 CLOSE #CHNL_NUM 15–4

15.3 PRINT #CHNL_NUM 15–4

15.4 INPUT #CHNL_NUM: VAR, VAR... 15–5
15.4.1 Inputting Multiple Variables 15–6

15.5 LINE INPUT #CHNL_NUM: STR_VAR, STR_VAR... 15–6
15.5.1 EOF Option 15–7
15.5.2 Multiple Variables 15–8

15.6 ASK #CHNL_NUM: 15–8
15.6.1 ASK Options 15–9

15.7 SET #CHNL_NUM: EXPR 15–10
15.7.1 SET #chnl_num: Options 15–11

15.8 DELETING FILES 15–12
15.8.1 KILL 15–12

CHAPTER 16 CREATING STRUCTURES, FIELD DEFINITIONS WITH SETUP 16–1

16.1 SETUP–OVERVIEW 16–1
16.1.1 Entering SETUP 16–2

16.2 SETUP PROMPTS 16–3

16.2.1 Structure Name 16–3
16.2.2 Database engine 16–3
16.2.3 Dataset 16–4
16.2.4 Data dictionary 16–5

16.3 SETUP PROCEDURES 16–7

16.4 DEFINE PROCEDURE 16–8
16.4.1 Field name 16–8
16.4.2 Classification 16–9
16.4.3 Description 16–12
16.4.4 Data type 16–12
16.4.5 First position 16–13
16.4.6 Field length 16–13
16.4.7 Number of occurrences 16–14
16.4.8 Semantics 16–14
16.4.9 Prompt text 16–15
16.4.10 Report heading 16–15
16.4.11 Print mask 16–15
16.4.12 Screen mask 16–16
16.4.13 Help text 16–16
16.4.14 Access rules 16–16
16.4.15 Validation rules 16–17
16.4.16 Field Definitions and Key Fields — General Information 16–17

16.5 SHOW PROCEDURE 16–18
16.5.1 Brief Field Display 16–19
16.5.2 Full Field Display 16–20
16.5.3 List Fields Display 16–20
16.5.4 Structure Display 16–21

16.6 MODIFY STRUCTURE PROCEDURE 16–21
16.6.1 Modify Structure General Information 16–22
16.6.2 Modify Structure Security Information 16–23

16.7 CREATE PROCEDURE 16–24

16.8 DEFINING COBOL RECORD FIELDS 16–27
16.8.1 Define Example 16–27

Contents

CHAPTER 17 INTOUCH RUNNING UNDER VMS 17–1

17.1 FILE SPECIFICATIONS 17–1

APPENDIX A BUILT-IN FUNCTIONS A–1

A.1 SYSTEM FUNCTIONS A–1

A.2 OTHER FUNCTIONS A–3

APPENDIX B RESERVED WORDS B–1

APPENDIX C INTOUCH'S ERROR AND EXCEPTION MESSAGES C–1

C.1 COMPILE ERRORS C–1

C.2 EXCEPTIONS C–4

APPENDIX D ASCII CHARACTER SET D–1

APPENDIX E CREATING LIBRARIES FOR USE WITH INTOUCH E–1

APPENDIX F INTOUCH DATABASE INTERFACES F–1

F.1 SUPPORTED DATABASE ENGINES F–1

INDEX

EXAMPLES

1–1	The INTOUCH Screen Frame	1–1
1–2	Running a VMS System Utility Program	1–2
1–3	The INTOUCH HELP Facility	1–3
1–4	Writing a Sample Program	1–5
1–5	INTOUCH Program Error Message	1–5
1–6	Running an INTOUCH Program	1–6
1–7	Interrupting an INTOUCH Program	1–7
1–8	Listing Program Lines	1–7
1–9	Resuming INTOUCH Program Execution	1–8
3–1	Executing an INTOUCH program	3–2
3–2	Structures	3–12
4–1	Debug TRACE ON Example	4–11
4–2	Listing Statistics in DEBUG System	4–14
4–3	Using BREAK in DEBUG System	4–15
4–4	Using STEP in DEBUG System	4–16
4–5	Using GO in DEBUG System	4–17
16–1	To Use SETUP	16–2
16–2	SETUP Prompts	16–2
16–3	SETUP Procedures	16–7
16–4	Field Definition Input Screen	16–8
16–5	DEFINE Screen showing field class generated defaults	16–11
16–6	DEFINE Screen showing SETUP generated defaults	16–11
16–7	SHOW - Brief Field Display Example	16–19
16–8	SHOW - Full Field Display Example	16–20
16–9	SHOW - Fields List Display Example	16–20
16–10	SHOW - Structure Display Example	16–21
16–11	CREATE Procedure RMS Input Screen	16–25

FIGURES

16–1	Character Positions	16–13

TABLES

3–1	E Notation	3–7
9–1	Supported Keystroke Concepts	9–28
14–1	Structure Types	14–32
A–1	CONVERT$ Function - Supported Data Types	A–5
A–2	DATE$ Function - Integer Values	A–7

Contents

A–3	DAYS Function - Integer Values	A–8
A–4	EDIT$ Function - Operation Values	A–10
A–5	FORMAT$ Function - Date Arguments	A–15
A–6	FULLTIME$ Function - Integer Values	A–16
D–1	ASCII Character Set	D–1

Preface

INTOUCH 4GL is a next generation database language for VAX and Alpha AXP computers. It includes a fourth-generation database language and an advanced development environment.

The INTOUCH language:

- is concise and powerful.
- promotes structured programming.
- is easy to use–even for computer novices.

INTOUCH includes a transparent interface to the RMS file management system as well as other database engines. The interface includes:

- simple structure statements for complex data manipulations.
- numerous extensions which allow the technical programmer to develop advanced applications.

The INTOUCH environment is designed for developing complex applications rapidly. The interactive environment:

- allows immediate response and fast development.
- allows you to interrupt program execution, change code and then continue execution.
- has high level debugging facilities for pinpointing program bugs quickly and accurately.
- returns errors immediately–like an interpreter–while code is optimized by the compiler.

Development cycle time is the time it takes to edit, compile, link, run and debug code. INTOUCH reduces this time by providing an interactive and fast response environment. INTOUCH offers features that cut cycle time significantly–up to 80%.

You will find INTOUCH an exciting and powerful programming tool.

About this Manual

This manual is one of two reference manuals describing the INTOUCH system. Both manuals are designed to provide the information you need in a manner that is concise and easy to use. The manuals are:

- *A Guide to the Language*
- *Specifications*

A Guide to the Language describes the INTOUCH system and how to develop programs with it. The Guide explains the development cycle and INTOUCH's features for speeding up this cycle. It describes the INTOUCH environment, program elements and the INTOUCH language and debugging system.

INTOUCH's **Specifications** contain a brief description of the INTOUCH system, statements, commands, etc., and some background on how the system operates.

Purpose

The purpose of this Guide is to present the information you will need to develop programs with INTOUCH. The intent is to provide you, the user, with simple and concise explanations of the INTOUCH system, features, and language. This manual is designed to provide a basic and thorough explanation of each element. This manual can also be used as a reference guide by advanced users.

Intended Audience

This Guide is written for both experienced programmers and those self-learners who have had some exposure to computers and programming languages. If you have little programming experience, or are using this manual to teach yourself, please start with chapter one.

Conventions Used in this Manual

User Conventions

In some places in this manual, the format of statements and commands is represented abstractly. For example:

```
Format:     [LET] var = expr
```

Certain conventions are used to represent values which you (the user) must supply. These conventions are always shown in *lower-case*. These are the user conventions:

array	Array element.
block of code	An INTOUCH statement or a series of INTOUCH statements.
chnl	An I/O channel associated with a file.
chnl_num	An I/O channel associated with a file.
cond_expr	Conditional expression.
col	Column. Used in the TAB or AT option to indicate the column to print at.
const	Constant value.
expr	Expression. May be a variable, constant, array element, structure reference, etc. or any combination of the previous separated by operators.
field_expr	Field name expression.
filename	File name. (Described briefly below and in the Command Language and DCL User's Guide of the OpenVMS documentation set).
file_spec	File specification. (See the Command Language and DCL User's Guide of the OpenVMS documentation set).
func	Function.
handl	Exception handler. Pertaining to an exception handling routine.
handler	The block of code in an exception handler which is executed when an exception is generated.

int	Integer value.
int_expr	Integer expression.
int_var	Integer variable.
label	Alphanumeric statement label.
libr	Pertaining to libraries.
line_num	Program line number.
name	Name. Indicates the declaration of a name or the name of a program unit, such as a subprogram.
num	Numeric value. Num indicates that either a real or integer numeric value may be used.
num_expr	Numeric expression. Num_expr indicates that either a real or integer numeric value may be used.
num_var	Numeric variable.
param	An expression which is passed to a function, subprogram or other program unit. A parameter can be any valid INTOUCH expression.
protected block	The block of code protected by an exception handling routine.
real	Real number. Indicates that only a real number may be used.
row	Row. Used in the AT option to specify the row to print at.
str	String. Str indicates that only a string value may be used.
str_expr	String expression.
struc	An INTOUCH structure.
struc_name	Structure name.
struc_expr	Structure expression.
sub	An INTOUCH subprogram.
target	The target point of a branch statement. The target can be either a program line number, an alphanumeric label, or a ROUTINE statement.
uncond_expr	Unconditional expression. Uncond_expr indicates that only an unconditional expression can be used.
var	Variable. May be a simple variable, array element or a structure reference.

Program Examples

In the program examples, keywords in INTOUCH statements and commands are in upper-case letters and information you (the user) supply is in lower-case. For example:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    ASK STRUCTURE cl, FIELD id: CHANGEABLE z
    ASK STRUCTURE cl, FIELD city: CHANGEABLE z1
    PRINT z
    PRINT z1
    CLOSE STRUCTURE cl
20  END
```

The double quote (") can be used in place of the single quote (') as long as they are paired and not specified.

Other Conventions

Other conventions used in this manual are:

[ ] Optional portions of a statement are enclosed in brackets.

```
INPUT [PROMPT str_expr] : var
```

{ | } Braces combined with a vertical bar indicate that one of the elements in the braces **must be chosen.** The elements are separated by the vertical bar.

```
EXIT {FOR | DO}
```

[|] Brackets combined with a vertical bar indicates that one of the options in the brackets can be used. The elements are separated by the vertical bar. Only one of the elements can be used.

```
DO [WHILE expr | UNTIL expr]
```

. . . An ellipsis indicates that the language element can be repeated. An ellipsis following an option enclosed in braces indicates that you can repeat the format of the whole option.

```
INPUT [PROMPT str_expr] : var, var...
```

dashes Three sets of dashes indicate a block of code.

```
DO
     ---
     ---  block of code
     ---
LOOP
```

. A vertical ellipsis indicates the continuation of a body of program code which is not a defined block of code.
.
.

1 Working in the INTOUCH Environment

1.1 Getting Started

It is easy to start INTOUCH, simply type the word **INTOUCH** in upper or lower-case letters, at the system prompt (usually the $). This will put you into the INTOUCH environment and the INTOUCH screen frame will be displayed.

Example 1–1: The INTOUCH Screen Frame

```
INTOUCH 4.2                    Program: NONAME
The Next Generation Language   Status :




INTOUCH

          INTOUCH Copyright (c) 1984-1995 Touch Technologies, Inc.
EXIT = Exit                                          \ = Back  HELP = Help
```

The frame consists of the two highlighted bars across the top of the screen and the three highlighted bars across the bottom. Inside the frame, is the highlighted INTOUCH prompt **INTOUCH**.

When you are in the INTOUCH environment you can create, edit and run INTOUCH programs, access files, and use VMS system utilities. You can also change the INTOUCH environment by using the INTOUCH commands and debugging system.

1.2 Accessing the VMS System Utilities

You can access the VMS system utilities from within the INTOUCH environment.

To run a VMS system utility program, enter **$ utility_name** at the INTOUCH prompt. INTOUCH will run the utility and return you to the the INTOUCH environment–the INTOUCH prompt. Here is an example:

Example 1–2: Running a VMS System Utility Program

```
INTOUCH 4.2                    Program: NONAME
The Next Generation Language   Status :


INTOUCH

$ show user
      VAX/VMS User Processes at 30-APR-1995 11:42:20.29
    Total number of users = 5,  number of processes = 16

 Username      Node     Interactive  Subprocess   Batch
 BATCH_CTL      TTI             -         1          1
 HTTP_SERVER     JR             -         1          1
 TESTER         FAST            5         1
 TESTER          TTI            1
 MRF_HANDLER    FAST            -         -          1
 UUCP_DAEMON     TTI            -         1          3
INTOUCH


EXIT = Exit                                          \ = Back  HELP = Help
```

To get into the VMS system environment and still remain in INTOUCH, you can enter **$$** at the INTOUCH prompt. You will be placed at the DCL prompt and the screen will appear as:

```
Passed to VMS...  Press  RETURN  when done.  USER:[TESTER]
$$
```

You can run any of the VMS utilities at this point. The **$$** reminds you that you are still in INTOUCH. When you want to return to the INTOUCH environment, simply press the Return key and the INTOUCH frame (if it is on) and the INTOUCH prompt will be displayed.

1.3 Recalling Previous Commands

The INTOUCH environment handles command recall. That is, you can access previous commands entered at the INTOUCH prompt. You can recall the last 50 previously entered commands or statements by using the UP and DOWN arrow keys. (You can also use Ctrl/B or the Prev key for UP and Ctrl/V or the Next key for DOWN.)

You can also recall a previous command or statement by entering part of the command and pressing the Find or PF2 key.

1.4 Getting Help when in INTOUCH

You can get help when at the INTOUCH prompt by entering **HELP** and pressing the Return key. The screen will clear and you will enter INTOUCH's HELP Facility.

Example 1–3: The INTOUCH HELP Facility

```
HELP

  INTOUCH is The Next Generation Language for VAX computers.  This
  help file contains information on INTOUCH.

  o     For detailed and basic level help, use Language

  o     To exit help, press the RETURN key repeatedly

  o     To redisplay a list of available topics, type a ?

  o     For information about the advantages of INTOUCH and the
        INTOUCH manuals, type Features at the topic prompt below


  Additional information available:

  Features   HELP       Language   Tables

Topic?
```

If you want information about one of the listed *topics*–Features, HELP, Language, Tables–enter the topic at the "Topic?" prompt and press Return. Some topics have subtopics. You can select a subtopic by entering the subtopic at the "TOPIC Subtopic?" prompt.

When you want to exit out of the HELP Facility, press the Return key continuously until you are back at the INTOUCH prompt.

INTOUCH has an extensive library of HELP text on many topics. When in the HELP Facility, you can type a question mark (?) at any HELP topic prompt and get a list of the available topics and sub-topics.

1.5 Using INTOUCH–Some Examples

Once you enter the INTOUCH environment, all of the INTOUCH features are available to you. For example, **PRINT** can be executed from the INTOUCH environment and it can also be a statement in an INTOUCH program. Enter **print 'hello'** in upper or lower-case at the INTOUCH prompt and you will see:

```
INTOUCH
print 'hello'
hello
INTOUCH
```

and if you enter **print date$**, you will see:

```
INTOUCH
print date$
19950430
INTOUCH
```

Spaces and Quotes

INTOUCH is space sensitive. As you try out the examples in this manual, be sure to type spaces where they are shown.

Either pairs of single quotes ('xxx') or pairs of double quotes ("xxx") can be used both in INTOUCH immediate mode (e.g. SAVE 'x.int' or SAVE "x.int") and within an INTOUCH program for string values (e.g. print 'hello' or print "hello").

1.5.1 Writing an INTOUCH program

Here is a sample program for you to try. Enter each line exactly as shown. At the end of each line, press the RETURN key.

Example 1–4: Writing a Sample Program

```
10  input 'Please enter your name': name$
20  print 'Hello, '; name$
30  end
```

If you make an error such as misspelling the word **print**:

Example 1–5: INTOUCH Program Error Message

```
10  input 'Please enter your name' : name$
20  printt 'Hello, '; name$

20  printt 'Hello, '; name$
..1
Unrecognized statement

                          Unrecognized statement
EXIT = Exit                                        \ = Back  HELP = Help
```

INTOUCH will display an *error message*. At this point, you can either retype the line, or you can correct the line. To correct this error, use the UP arrow key to recall the line, use the left arrow key to move over to the second **t**, use the delete key to delete the second **t** and then press the Return key.

Now that you have typed in the sample program, you are ready to run it.

1.5.2 Running an INTOUCH program

To run or execute the sample INTOUCH program, you can enter **RUN** or **RNH**. RUN will print a program header and RNH will not.

When you run the program, you will be asked to enter your name. Enter your name and press Return.

Example 1–6: Running an INTOUCH Program

```
10  input 'Please enter your name': name$
20  print 'Hello, '; name$
30  end
run
NONAME        30-APR-1995 14:51
Please enter your name? Sally
Hello, Sally

INTOUCH

rnh
Please enter your name? Sam
Hello, Sam

INTOUCH
```

1.5.3 Interrupting and Changing an INTOUCH program

You can interrupt an INTOUCH program while it is running, make immediate changes and continue execution. This can be done because INTOUCH compiles each statement as it is entered. And, as you have seen previously, it returns any compile-time errors immediately.

The following shows how to run the sample program, iterrupt it, make a change and continue execution.

Interrupting the Program

To interrupt an INTOUCH program, enter Ctrl/C while the program is running.

To interrupt the sample program, RUN the program and at the "Please enter your name?" prompt, enter Ctrl/C. INTOUCH will interrupt program execution and tell you where it stopped.

Example 1–7: Interrupting an INTOUCH Program

```
INTOUCH 4.2                    Program: NONAME
The Next Generation Language   Status :  BREAK at 10

INTOUCH

run
NONAME         30-APR-1995 15:33
 Cancel nter your name?

BREAK at 10

INTOUCH
```

BREAK at 10 tells you that the program was stopped when it was executing line 10.

You are now at the INTOUCH prompt and can make changes.

Listing Program Lines

Before making changes, you can list the current program lines by entering the word **LIST**. For example:

Example 1–8: Listing Program Lines

```
INTOUCH

list
NONAME         30-APR-1995 16:11
10  input 'Please enter your name': name$
20  print 'Hello, '; name$
30  end

INTOUCH
```

Changing the Sample Program

You can change the sample program by entering the following two lines:

```
10 name$ = 'Sunny'

25 print day$
```

```
10  name$ = 'Sunny'
25  print day$
```

Resuming Program Execution

You have changed the sample program and are now ready to resume execution. You want the program to start running at line 10. To continue program execution, enter **GOTO 10** and press Return.

Example 1–9: Resuming INTOUCH Program Execution

```
INTOUCH
10  name$ = 'Sunny'
25  print day$
goto 10
Hello, Sunny
Sunday
INTOUCH
```

INTOUCH started program execution at line 10 and ran all the program lines including line 25, the line you added.

As you can see, INTOUCH makes it easy to add and change program code.

1.6 Using EXIT and Back

When you want to get out of the INTOUCH environment, enter **EXIT** and you will be returned to the system prompt (the $ prompt).

When you EXIT out of INTOUCH, you lose the environment you have set up. Any unsaved changes you have made to the environment are lost. When you re-enter INTOUCH, you will begin again with the default INTOUCH environment.

Summary Message Line

When you are in INTOUCH and at the INTOUCH prompt or in an INTOUCH program, utility or routine, you will see a summary message line at the bottom of the screen frame.

```
INTOUCH

EXIT = Exit                                          \ = Back  HELP = Help
```

The summary message reminds you that, while you are in INTOUCH:

— you can enter HELP to get help.

— you can enter EXIT to exit.

— you can use the \ (backslash) to backup to a previous prompt or menu.

You also have access to these three commands from within your own INTOUCH programs. Here is an example of how they are used in a program:

```
10  DO
      INPUT 'Please enter your name': name$
      IF  _HELP  THEN  MESSAGE 'Type in your name'
      IF  _HELP  or  _BACK  THEN  REPEAT DO
      IF  _EXIT  THEN  EXIT DO
      PRINT 'Hello '; name$
    END DO
20  END

RNH
Please enter your name? help
Please enter your name? \
Please enter your name? exit

               Type in your name
```

When in an INTOUCH program or routine, you can also press the Help key to get help and press Ctrl/Z to exit.

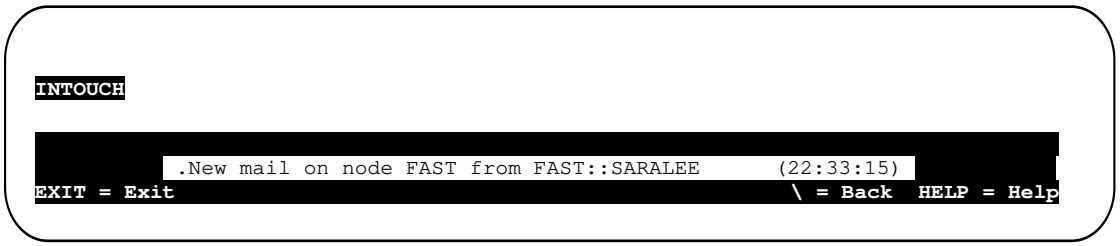
Summary

You can enter EXIT or press CTRL/Z from within a program to do one of the following:

- Exit a procedure within the program.
- Exit the program and return to the INTOUCH prompt.
- EXIT the program and return to the system prompt if you are running a compiled program.

1.7 Messages in INTOUCH

INTOUCH displays system, program and procedure messages within the bottom screen frame. For example:



If the screen frame is not on, messages will be displayed on the bottom line of the screen. These are INTOUCH's default message lines. If you want messages to appear elsewhere on the screen you can change the default line (see Section 9.9, ASK | SET MESSAGELINE).

1.8 Using the Tab Key Features

When in INTOUCH, the Tab key can be used to perform various operations that cut down the time it takes to write program code. As you get familiar with INTOUCH and the INTOUCH language, you will find that the Tab key features can be very helpful.

1.8.1 Tab Key Indenting

When you use the Tab key at the INTOUCH prompt or while in the INTOUCH editor, you will find that Tab indents in the following ways:

- INTOUCH indents a line according to statement type. Typically, the Tab pressed once at the beginning of the line indents the line 8 spaces.
- A subsequent Tab on the same line indents an additional 2 spaces.
- After the indentation has reached column 16, a subsequent Tab is 8 spaces.
- If the previous line typed is a block construct (i.e. ROUTINE, FOR, DO, EXTRACT, IF, etc.), INTOUCH automatically adjusts the first Tab of the line to indent 2 additional spaces.

1.8.2 Command Completion

When entering an INTOUCH command, you can enter part of the command, and then press the Tab key. INTOUCH will then supply the full name of the command. If the command is ambiguous, a selection menu of all valid choices will appear.

In the following example, INTOUCH completes **so** to form the command **sort**:

```
20  so Tab              <--- your entry
20  sort                <--- INTOUCH supplies
```

In this example, **in** is ambiguous, so INTOUCH provides a menu of choices:

```
                                          +--Choices---+
                                          |  INCLUDE   |
                                          |  INF       |
                                          |  INPUT     |
                                          +------------+
30      in
```

1.8.3 Spelling Correction

The Tab key can be used to correct misspelled words in INTOUCH. Just position the cursor on the misspelled word and press the Tab key. The word will be replaced with its correct spelling. In the example below, the misspelled command **SET STRCTRE** is changed *instantly* to the correct command spelling when the Tab key is pressed.

```
50  set strctre Tab
50  set structure
```

1.8.4 Expanding Abbreviations

INTOUCH uses abbreviations for a number of statements. You can use the [Tab] key with any of the specific abbreviations to expand the appreviations. To use this feature, enter an abbreviation and then press the [Tab] key. The current abbreviations that can be used are:

os.	OPEN STRUCTURE
es.	EXTRACT STRUCTURE
ee.	END EXTRACT
in.	INCLUDE
ex.	EXCLUDE
fe.	FOR EACH
e.	EDIT
l.	LIST
p.	PRINT
pu.	PRINT USING
ss.	SET STRUCTURE
pa.	PRINT AT
li.	LINE INPUT
pc.	PRINT #
oc.	OPEN #
pe.	PRINT _EXTRACTED
lm.	LINE INPUT MENU
ls.	LINE INPUT SCREEN

1.9 Other INTOUCH Features

You can enter INTOUCH program code at the INTOUCH prompt. You can also use the **INTOUCH editor** to create and change program code. Refer to Section 2.3.9, EDIT for information on the INTOUCH editor.

The appendix sections of this Guide provide information on INTOUCH's built-in fuctions, error and exception messages and other general topics. You will want to get familiar with the built-in functions as they will allow you to easily manipulate data.

Now that you have an idea of how to use and work in the INTOUCH environment, the remainder of this Guide will provide the information you need to create and use INTOUCH programs.

2 INTOUCH Commands

2.1 Presentation of Command and Statement Information

Part of this chapter and several other chapters in this Guide describe the commands and statements in the INTOUCH language. The purpose of this section is to explain how the command and statement information is presented.

Statements that can only be used in connection with one another, such as IF THEN or the SELECT CASE statements, are described together. Each command and statement description includes the following information:

FORMAT:

```
INPUT [options] var [, var...]                                    (example 1)

EXTRACT STRUCTURE struc_name: KEY field = expr1 [TO expr2]    (example 2)
          ---
          ---  block of code
          ---
END EXTRACT
```

The FORMAT contains the command or statement format. This includes the required keywords and the optional elements, if there are any.

Keywords are words that must be typed exactly as shown in the format. Keywords are shown in UPPER-CASE, e.g. EXTRACT STRUCTURE, KEY, TO, END EXTRACT. The elements in lower-case represent information that must be provided by the user in order to use the statement. For example, "struc_name" (the structure name) must be provided by the user in this statement. "field" and "expr1" information must also be provided.

Keywords and lower-case elements in brackets are optional. Multiple options are separated with commas.

INTOUCH is space sensitive, so spaces must be included where they are shown in the format. Either the single quote or double quote can be used as long as they are paired.

For information on elements (struc_name, expr1, etc.) and their representations, bracket usage and other conventions used in this Guide, you can refer to Conventions Used in this Manual.

EXAMPLE:

```
10  INPUT 'Enter full name', TIMEOUT 30, ELAPSED x%: name$
    PRINT name$
    PRINT 'Elapsed time: '; x%
20  END

RNH
Enter full name? TTI Tester
TTI Tester
Elapsed time:  13
```

The EXAMPLE shows how the command or statement is used. Wherever possible, the example is a full program rather than a program fragment.

You are encouraged to type in the INTOUCH examples and try running them.

PURPOSE:

In some cases, PURPOSE information is provided for usage clarity.

DESCRIPTION:

The DESCRIPTION is a detailed description of the command or statement which tells about the command or statement, explains how it works and other specific information.

2.2 About INTOUCH Commands

INTOUCH *COMMANDS* cause INTOUCH to take some action. You can use the commands at the INTOUCH prompt. For instance, in the following example, the command **RUN** executes the entered program.

```
10  INPUT 'Your name, please': name$
20  PRINT 'Hello, '; name$
30  END

RUN
NONAME       21-APR-1995 17:40
Your name, please? Tester
Hello, Tester

INTOUCH
```

2.3 INTOUCH Commands

The following sections describe and explain how to use many of the INTOUCH commands.

2.3.1 NEW

FORMAT:

```
NEW ['workspace_name']
```

EXAMPLE:

```
INTOUCH 4.2                    Program: print_name
The Next Generation Language   Status :


INTOUCH

NEW 'print_name'

INTOUCH


EXIT = Exit                                          \ = Back  HELP = Help
```

DESCRIPTION:

NEW clears current memory and assigns a name to the current workspace. The workspace name:

— can be up to, but not longer than, 19 characters.

— can consist of letters, numbers, underscores and hyphens.

Whenever you begin writing a new program, type NEW followed by the name of the new program. If you do not enter a workspace name, INTOUCH uses the default name "NONAME".

When INTOUCH executes the NEW command, it clears current memory and assigns the name you specify or NONAME, if you do not specific a name, to the current workspace. The workspace name appears to the right of the "Program:" heading in the upper right corner of the screen.

Note

When you use the NEW command, *current memory* is cleared and any programs in memory are lost. Therefore, if you want to save what is in current memory, you need to do a SAVE or REPLACE before you use the NEW command. INTOUCH

writes all active output buffers, closes all files and releases all storage.

2.3.2 SAVE

FORMAT:

```
SAVE ['file_spec']
```

EXAMPLE:

```
SAVE 'print_name'
INTOUCH
```

DESCRIPTION:

After a *NEW* program has been created, the **SAVE** command is used to save it for later use.

The SAVE command is used to save original files. An original file is one which has a unique file specification (i.e., there are no other files saved under that name). When INTOUCH saves a program, it is ordered sequentially by line number. INTOUCH saves programs as text files.

SAVE, with no file specification, saves your file under the current workspace name with an extension of **.INT**. You can include a complete file specification with the SAVE command. For example:

```
SAVE 'testdisk:[programs]print_name'
```

If a file specification is included, INTOUCH tries to save the file under the specification. (See the *Command Language and DCL User's Guide* Volume 2–Section 3, of the OpenVMS Version 5.5 documentation set for a description of file specifications.)

If you do not give a complete file specification, or if you do not give a directory, INTOUCH saves the program in your current (default) directory.

For the SAVE command, the file specification must be unique. If the file specification is not unique (if a file with the same name has already been saved), an exception will result. (Exceptions are discussed in Appendix C, INTOUCH's Error and Exception Messages.)

After you have saved a file once, you use the REPLACE command to save the copy or changed version.

2.3.3 REPLACE

FORMAT:

```
REPLACE ['file_spec']
```

EXAMPLE:

```
REPLACE 'print_name'
INTOUCH
```

DESCRIPTION:

After you make changes to a program, use **REPLACE** to save the new version.

The REPLACE command replaces the last version of your program with the version in current memory. If an earlier version of the file cannot be found, INTOUCH creates a new file with the specification given. *file_spec* is the specification for the file being replaced. If no file specification is included, INTOUCH uses the current workspace name and the default **.INT**.

If an earlier version does exist, INTOUCH saves the copy in current memory and increments the version number by 1. Previous versions of the file remain unchanged. The program in current memory is not changed when the REPLACE command is executed.

2.3.4 ABORT

FORMAT:

```
ABORT [int_expr]
```

EXAMPLE:

```
INTOUCH
ABORT

                          INTOUCH Finished  1-MAY-1995 17:16
FAST$
```

DESCRIPTION:

ABORT aborts INTOUCH *without* saving your latest changes.

This statement passes the ABORT status to the operating system. In this way, INTOUCH applications can force a command procedure to abort.

You can also give ABORT an OPTIONAL status value, *int_expr*, to pass to the operating system. *int_expr* is an exit status code in decimal. This value will be passed to DCL as the reason why INTOUCH aborted.

```
INTOUCH
ABORT 200

                          INTOUCH Finished  1-MAY-1995 17:16
%SYSTEM-W-GSDFULL, global section descriptor table is full
$
```

"ABORT 200" produces the same result as:

```
$ exit %d200
%SYSTEM-W-GSDFULL, global section descriptor table is full
$
```

2.3.5 RUN and RNH

FORMAT:

```
{RUN | RNH} ['file_spec']
```

EXAMPLE:

```
10  INPUT 'Your name, please': name$
20  PRINT 'Hello, '; name$
30  END

RUN
NONAME       21-APR-1995 17:40
Your name, please? Tester
Hello, Tester

INTOUCH
```

DESCRIPTION:

Use the **RUN** or **RNH** command to execute a program after it has been written.

The RUN command executes programs. RUN with no file specification (as in the above example) runs the current program in memory.

When INTOUCH executes the RUN command, INTOUCH displays a header with the program name, current date and time. INTOUCH then executes the program in memory.

You can give a file specification with the RUN command. If a file specification is provided, INTOUCH searches for the file, loads it, and then runs it. If no extension is given, INTOUCH will use the default extension **.INT**.

```
INTOUCH

RUN 'testdisk:[programs]print_name'
NONAME       21-APR-1995 17:40
Your name, please? Tester
Hello, Tester

INTOUCH
```

RNH runs the specified program without displaying the program header. Other than suppressing the header, RNH operates the same as the RUN command.

2.3.6 OLD

FORMAT:

```
OLD ['file_spec']
```

```
INTOUCH

OLD 'print_name'

INTOUCH

LIST
PRINT_NAME     1-MAY-1995 20:32
10  INPUT 'Your name, please': name$
20  PRINT 'Hello, '; name$
30  END

INTOUCH
```

DESCRIPTION:

The **OLD** command loads a program you have saved.

OLD searches for the file specified and loads it into current memory. *file_spec* is the specification for the file you are loading. If no extension is specified, INTOUCH uses the default extension **.INT**. If no *file_spec* is provided with the OLD command, INTOUCH tries to load the file carrying the current workspace name and an extension of **.INT**.

When INTOUCH loads a file, it assigns the file name (minus the extension) to the current workspace. The file name appears at the "Program:" heading in the upper right corner of the screen.

Note

When the OLD command is executed, any file previously in memory is lost.

The contents of any variables in the new program are cleared (set to zero for numeric variables and a null string for string variables).

If INTOUCH cannot find the file specified, an exception is generated and the file in current memory does not change.

2.3.7 LIST

FORMAT:

```
{LIST | LNH} [label | line_num | routine {- | ,} ...]
```

```
INTOUCH

OLD 'print_name'

INTOUCH

LIST
PRINT_NAME     1-MAY-1995 20:32
10  INPUT 'Your name, please': name$
20  PRINT 'Hello, '; name$
30  END

INTOUCH
```

DESCRIPTION:

Use the **LIST** command to display all or part of your program source code.

LIST displays lines from the program in current memory. The listing includes a header with the program name, current date and time. The **LNH** command suppresses the header. LIST (or LNH) by itself lists the entire program in current memory. It lists the program sorted sequentially by line number. If no program is in current memory, nothing will be listed.

LNH lists the program without displaying the program header. Other than suppressing the header, LNH operates the same as the LIST command.

Specific sections of a program can be listed by referencing the program's line numbers, labels and routines. You can also list a range of line numbers (low to high) and combinations of line numbers, labels and routines. For example:

LIST 500, 630, 710	lists lines 500, 630 and 710
LIST 210-230, 400	lists lines 210 thru 230 and line 400
LIST do_input	lists the lines of code under the label "do_input"
LIST date_routine	lists the lines in the "date_routine" routine
LIST 210-230, do_input	lists a line range and the label lines
LIST 21000	lists from the line number to the end of the program

If a line is specified which does not exist, nothing is listed. INTOUCH displays program lines in the order requested. Therefore, lines can be listed out of sequential order.

2.3.8 LIST ERROR

FORMAT:

```
[LIST | LNH] ERROR [:label | line_num [- | ,] label | line_num...]
```

EXAMPLE:

```
LIST ERROR
    or
LNH ERROR
```

DESCRIPTION:

LIST ERROR lists compile-time errors to the terminal. LIST ERROR displays the line number and the source code of all lines with errors. **LNH ERROR** lists the errors without the program header, date and time.

All available options for the LIST command are available with LIST ERROR.

2.3.9 EDIT

FORMAT:

```
EDIT line_num | label | routine [[- | ,] line_num | label | routine...]
```

EXAMPLE:

```
EDIT 410, 510-570, DO_TOTALS
```

PURPOSE:

EDIT allows you to make changes to your program using a full-screen text editor.

DESCRIPTION:

EDIT transfers control to an editor. The program or a specified part of it is placed in the editor. You can then modify the program using the editor. When the editor is exited, INTOUCH returns to the INTOUCH prompt.

The INTOUCH environment is integrated into the TPU editor with EDT keypad and INTOUCH extended features. The INTOUCH editor is automatically activated when you enter **EDIT**.

If you do not want to use the INTOUCH editor, you will need to define the logical, **INTOUCH_EDITOR**, to *FALSE* and the **EDIT** symbol to the editor of your choice.

```
$ DEFINE INTOUCH_EDITOR FALSE
$ DEFINE EDIT == "EDIT/EDT"
```

EDIT by itself, puts the entire program in current memory. The program must be in current memory to use the EDIT command. Specific sections of a file can be edited by referencing the program's line numbers, labels and routines.

All available options for the LIST command are available to the EDIT command.

2.3.10 RENUMBER

FORMAT:

```
RENUMBER
```

EXAMPLE:

```
3  DIM name$(5)
5  FOR i = 1 TO 5
7    INPUT 'Your name, please': name$(i)
8    PRINT 'Hello, '; name$(i)
9  NEXT i
10 END

RENUMBER

Pass 1, scanning source statements
Pass 2, changing line numbers
Pass 3, rebuilding source code
INTOUCH

LNH
1000  DIM name$(5)
1010  FOR i = 1 TO 5
1020    INPUT 'Your name, please': name$(i)
1030    PRINT 'Hello, '; name$(i)
1040  NEXT i
1050 END
```

PURPOSE:

Use **RENUMBER** to renumber your program when you run out of room between line numbers.

DESCRIPTION:

RENUMBER operates on the program in current memory. RENUMBER renumbers or resequences the program in current memory. It takes the first line number and renumbers it 1000. Subsequent line numbers are incremented by ten and all references to line numbers are changed to match.

2.3.11 FRAME ON | OFF

FORMAT:

```
FRAME {ON | OFF}
```

EXAMPLE:

```
FRAME ON
FRAME OFF
```

DESCRIPTION:

FRAME OFF clears the screen and turns the INTOUCH frame off. The INTOUCH frame consists of the highlighted bars across the top and bottom of the screen and the information they contain.

FRAME ON clears the screen and turns the INTOUCH frame on.

2.3.12 REFRESH

FORMAT:

```
REFRESH
```

EXAMPLE:

```
REFRESH
```

DESCRIPTION:

REFRESH repaints the INTOUCH screen. INTOUCH clears the screen and repaints the INTOUCH frame and prompt. This includes any other information that was on the screen.

2.3.13 IDE

FORMAT:

```
IDE
```

EXAMPLE:

```
INTOUCH

IDE

               Version 4.2
   INTOUCH Copyright (c) 1984-1995 Touch Technologies, Inc.

       USA: (800) 525-2527     (619) 566-3603

INTOUCH
```

DESCRIPTION:

IDE causes INTOUCH to display its copyright and version number. Use the version number when contacting Touch Technologies, Inc. with suggestions or problems.

2.3.14 COMPILE

FORMAT:

```
COMPILE
```

EXAMPLE:

```
COMPILE
```

PURPOSE:

Use **COMPILE** to compile your finished INTOUCH programs. Compiled INTOUCH programs can be run from outside the INTOUCH environment.

DESCRIPTION:

COMPILE converts the program in current memory into an INTOUCH **RUN** file. This RUN file can then be executed from outside the INTOUCH environment. COMPILE *only* works on the program in current memory.

When INTOUCH executes the COMPILE command, it compiles the source code and saves the compiled copy with the current workspace name and the extension **.RUN** (i.e. PRINT_NAME.RUN). If a compiled version of the file already exists, INTOUCH compiles the new version, saves it and increments the version number by 1.

You can run compiled programs by typing INTOUCH followed by the filename at the system prompt. For example:

```
$ INTOUCH print_name
```

The above command starts INTOUCH and runs the compiled version of the PRINT_NAME program. When execution is complete, INTOUCH will be exited and you will return to the system prompt.

2.3.14.1 Compiled Images

INTOUCH procedures and programs can also be compiled into fast-starting memory images. These images are activated within two seconds, regardless of the size of the INTOUCH procedure or program. The drawback of using compiled images is that they are very large. The minimum size is approximately 500 disk blocks. Image sizes increase in 256 block increments.

INTOUCH images are compiled from outside the INTOUCH environment. To compile into an image, use the following command:

```
    $ INTOUCH/COMPILE filename
Example:
    $ INTOUCH/COMPILE print_name
```

Compiled image files have a file extension of **.INT_IMG** for VAX systems and **.INT_IMG_AXP** for Alpha AXP systems.

```
    $ INTOUCH/COMPILE CUSTOMER
will create:
        CUSTOMER.INT_IMG            <--- on VAX systems
        CUSTOMER.INT_IMG_AXP        <--- on AXP systems
```

2.3.14.2 Running Compiled Images

You can run INTOUCH compiled images from outside of the INTOUCH environment. To run an image, use the following command format:

```
    $ INTOUCH/IMAGE filename
Example:
    $ INTOUCH/IMAGE customer
```

To set up a DCL symbol to run an INTOUCH compiled image, use the following command format:

```
    $ symbolname :== '$INTOUCH_IMAGE filename/IMAGE'
Example:
    $ MAINTAIN :== '$INTOUCH_IMAGE TTI_RUN:MAINTAIN/IMAGE'
```

2.3.15 Developing INTOUCH Programs

The following command entered at the system prompt:

```
    $ INTOUCH/DEVELOP filename.INT
Example:
    $ INTOUCH/DEVELOP sample_print.int
```

puts you into INTOUCH, OLD's in *filename.INT* and automatically replaces the file when you enter EXIT to exit INTOUCH.

This command streamlines the file change procedures by performing these steps for you:

```
$ INTOUCH               gets into the INTOUCH environment
OLD 'filename'          brings the file into the workspace
REPLACE 'filename'      after you make your changes and enter
                        EXIT, INTOUCH adds "1" to the version
                        number and replaces the file
```

2.3.16 GO

FORMAT:

```
GO
```

EXAMPLE:

```
GO
```

DESCRIPTION:

Use **GO** to continue a program after it has been interrupted.

GO resumes program execution after it has been interrupted. Once execution has stopped, you can enter immediate mode and debug commands, change code, etc. GO lets you resume execution even after changing code. If a HALT or BREAK statement was used, execution resumes at the first statement after the HALT or BREAK.

```
10  DEBUG ON
20  FOR i = 1 TO 6
30  IF  i = 4  THEN  BREAK
40  NEXT i
50  END

INTOUCH

RNH
BREAK at 30

INTOUCH

PRINT SQR(i)
 2

INTOUCH

35  PRINT i;
GO
 4  5  6
INTOUCH
```

2.3.17 UNSAVE

FORMAT:

```
UNSAVE ['file_spec']
```

EXAMPLE:

```
UNSAVE 'print_name'
```

DESCRIPTION:

UNSAVE deletes or unsaves the latest version of the file specified. The *file_spec* is the specification for the file. If no *file_spec* is given, INTOUCH uses the current workspace name and the default extension **.INT**. When INTOUCH executes the UNSAVE command, it searches for the file specified and then deletes that file. The source program will still remain in current memory.

3 Program Elements and Data in INTOUCH

This chapter describes the basic elements that make up an INTOUCH program. It also describes the types of data used with INTOUCH programs and how the data is processed.

3.1 INTOUCH Program Elements

A program is a series of instructions. These instructions describe how to manipulate data to produce a desired result. You determine what data the program will manipulate, how the program will manipulate it and what the results of these manipulations will be.

When you create a program, you must use instructions that INTOUCH understands. The INTOUCH language provides these instructions. The language consists of statements. These statements are something like the words in this manual. You put them together in a meaningful order and INTOUCH executes the program you write. Here is an example of an INTOUCH program:

```
10  INPUT 'Please enter your name': name$
20  PRINT 'Hello, '; name$
    PRINT 'Today is '; day$
    PRINT name$; ', have a good '; day$
30  END
```

The **INPUT** statement tells INTOUCH to ask for a name. The **PRINT** statements tell INTOUCH to print the information. **END** tells INTOUCH it has reached the end of the program.

With INTOUCH, when you execute this program, it will look like this:

Example 3–1: Executing an INTOUCH program

```
INTOUCH

10  INPUT 'Please enter your name': name$
20  PRINT 'Hello, '; name$
    PRINT 'Today is '; day$
    PRINT name$; ', have a good '; day$
30  END

INTOUCH

RUN
NONAME         2-MAY-1995 10:45
Please enter your name? Julian
Hello, Julian
Today is Tuesday
Julian, have a good Tuesday

INTOUCH
```

RUN is a command. RUN tells INTOUCH to perform the instructions you have given. You will need to use commands as well as statements to develop your programs.

When you run the above program, you will notice that the day is not asked for. **DAY$** is a reserved word that INTOUCH uses to store the day in. There are several other reserved words that INTOUCH uses. You can refer to Appendix B, Reserved Words to see a list of the reserved words.

INTOUCH programs are modular in structure. Every program can be divided into program units. The main unit is the main body of the program. This unit begins with the first program line and ends with the END statement.

```
First program line --   10  INPUT 'Please enter your name': name$
                        20  PRINT 'Hello, '; name$
                            PRINT 'Today is '; day$
                            PRINT name$; ', have a good '; day$
     END statement --   30  END
```

Each program unit is made up of program lines. A program line consists of one or more statements. In the above example, line 20 has three statements. If you include more than one statement in a line, you must separate the statements with a line terminator (a carriage return). You can terminate a line from the keyboard by pressing the Return or Enter key. For example:

```
10  INPUT 'Please enter your name': name$<Return>
20  PRINT 'Hello, '; name$<Return>
    PRINT 'Today is '; day$<Enter>
    PRINT name$; ', have a good '; day$<Return>
30  END<Return>
```

When INTOUCH executes a program, it starts at the lowest line number and executes all the statements in that program line. Then it goes to the next sequential line number and executes all the statements in that line and so on.

Multiple statements can be placed on a single line, delimited by the \ (backslash). CHR$(10) also signals the same as a \. A \ terminating the line implies an "&".

```
10  FOR z1 = 4. to 7. STEP .5 \ PRINT z1 \ NEXT z1
20  END

RNH
 4
 4.5
 5
 5.5
 6
 6.5
 7
```

Note

Putting multiple statements on a single line is not recommended. It makes changing code difficult and the logic flow is not easy to follow.

INTOUCH accepts line numbers in the range of **1** to **999999**. Line numbers must be integers.

INTOUCH allows programs with NO line numbers execpt, there must be a line **1** at the beginning of the program. For example:

```
1  PROGRAM print_name
       -------
       -------
       -------
   END
```

3.1.1 Continuing Program Lines

You can continue a statement on the next physical line by using an **ampersand** (&). This allows you to continue a statement that is too long to fit on one line.

To continue a statement, place the ampersand at the end of the continued line. You can also place an optional ampersand at the beginning of the next line. For example:

```
10  INPUT 'Please enter your name': name$
20  PRINT 'Hello, '; name$
    PRINT 'Today is '; day$
    PRINT name$; &
          ', have a good '; &      ! required ampersand
          & day$                   ! optional ampersand
30  END

RNH
Please enter your name? Julian
Hello, Julian
Today is Tuesday
Julian, have a good Tuesday
```

Note

In the above example, the comment text follows the line continuation character (&). Whenever you comment a continued line, the exclamation mark *must* come AFTER the ampersand.

You can continue statements between words; however, you cannot break up a word.

3.1.2 Labels

A **label** is a way to identify a place in a program. Labels must:

begin with a letter
can be up to 31 characters long
can contain letters, digits and underscores

To use a label, put the label name on a program line and separate it from the following statement with a colon. The label name can also be put on a line by itself and followed by multiple statements. Both options are shown in the example below.

```
10  ask_input:  INPUT 'Please enter your name': name$
20  do_print:
    PRINT 'Hello, '; name$
    PRINT 'Today is '; day$
    PRINT name$; ', have a good '; day$
30  END
```

3.1.3 Storing INTOUCH Programs

INTOUCH programs have a default extension of **.INT** so, you do not have to use a file extension with your file name when using the INTOUCH commands (SAVE, UNSAVE, OLD, etc.). It is recommended that you use this extension for all of your INTOUCH programs.

INTOUCH source programs are saved as **text files**. You can print the text files with the TYPE and PRINT commands, and edit them with any text editor.

3.2 Data in INTOUCH

Programs manipulate data. The data can be numeric or textual, but the data must be presented according to the rules of the INTOUCH language.

3.2.1 Data Types

INTOUCH accepts three basic types of data:

1 integer numbers

2 real numbers

3 and string (textual) data

The following sections describe each of the three types of data.

3.2.1.1 Integers

An **integer** number is a whole number with no fractional part. The following are examples of integers:

```
+5%        894%        -369004%
```

Integers can be positive or negative. Integer constants can end with a trailing percent sign (%). About integers:

- Names for integer variables, arrays or functions must be designated by a trailing percent sign (%) or declared with a DECLARE statement.
- A negative integer constant is designated by a leading minus sign.
- A positive integer constant can be designated by a leading plus sign or by the absence of a sign.
- Integers have a size of 32 bits.
- INTOUCH allows integer numbers anywhere in the range of -2147483648 to +2147483647.

3.2.1.2 Real Numbers

A **real** number can have a fractional part. The following are real constants:

```
5.4              894.0              -369004
```

Real numbers can be positive or negative. Any number that is not designated as an integer or a string is treated as a real number.

A real number without a decimal point is called an **ambiguous constant**. About real numbers:

- A variable, array or function whose name has no trailing % or $ is treated as a real number unless it is declared.
- A negative real constant has a leading minus sign.
- A positive real constant can have a leading plus sign or no sign.
- Ambiguous constants are treated as real numbers with a fractional part of 0. (5 = 5.0)
- Real numbers have a size of 64 bits, a precision 15 decimal digits and correspond to the VAX double floating point number.
- Real numbers can be anywhere in the range of -1.7E38 to +1.7E38.
- Real numbers have a precision of .29E-38 to 1.7E38.

3.2.1.3 String Data

String data consists of text. Text can be any characters. All of the following are valid string constants:

```
'Hello, Fred.'
'Account number (##-####)'
'65'
```

About string data:

- String constants must be enclosed in single or double matching quotes. String variables must be designated by a trailing dollar sign ($) or declared with a DECLARE statement.
- A string can contain up to 65535 characters.
- String variables are dynamic in length.

3.2.2 BOOLEAN

Boolean variables represent either a TRUE or FALSE) condition. These variables can be expressed with a trailing "?". For example:

```
10  done? = FALSE
    DO
      INPUT 'Ready': ans$
      IF  _EXIT  THEN  done? = TRUE
      IF  _BACK  THEN  done? = TRUE
    LOOP UNTIL done?
20  END
```

3.2.3 Expressions

Expressions can be:

- constants
- simple variables
- array elements. Array elements are a type of variable.
- structure references. Structure references are a feature of the INTOUCH data structure system.
- substrings.
- compound expressions. Compound expressions are any combination of the above operands separated by operators.

3.2.4 Constants

A **constant** is a value that does not change during a program execution. Constants can be any of the three data types: integer numeric, real numeric or string.

3.2.4.1 Numeric constants

Integer numeric constants are written with digits, with no decimal fraction. An integer constant can end with a percent sign. Examples:

```
235             412%
```

Real numeric constants are written with a sign, digits, and a decimal point. For example, 5.38 is a real numeric constant.

3.2.4.2 E Notation

E notation is a shorthand way of representing very large or very small numbers. Real numeric constants can be written in E notation. INTOUCH displays any real value of greater than 12 decimal digits in E notation. The format of E notation is:

```
+/- number E +/- n
```

Table 3–1 E Notation

Notation	Description
+/-	is the sign of the number (positive or negative). If no sign is given, the number is assumed to be positive.
number	is the number on which exponentiation is performed.
E	exponentiation means "times 10 to the power of".
+/-	is the sign of the exponent (positive or negative). If no sign is given, the exponent is assumed to be positive.
n	is the exponent represented as one or more integer digits.

If you wanted to represent 1000 in E notation, you would use:

```
+1E3
```

This means "take 1 and multiply it by 10 to the 3rd power" which is 1*(10^3) = 1000. If you wanted to represent .01, you would use a negative exponent.

```
1E-2 = 1*(10^-2) = .01
```

3.2.4.3 String Constants

String constants must be enclosed in quotes. The quotes are called **string delimiters**. They tell INTOUCH where the string begins and ends. The quotes are not considered part of the string. Everything within the quotes *is* considered part of the string. Note that the following strings are different:

```
10  PRINT 'Account number (##-####)'
20  PRINT '    Account number     (##-####)'
30  END

RNH
Account number (##-####)
    Account number     (##-####)
```

An empty string is called a **null string**. A null string is indicated by a pair of quotes with nothing between them ("").

String delimiters can be single quotes (') or double quotes ("). However, the quotes must match and be paired.

Quotes can be included as part of a string by using them within the string delimiters. For example:

```
"Message 'Time to go home!'"
```

The string part of this is:

```
 Message 'Time to go home!'
```

The delimiter quotes can be used within a string, by repeating them twice. For instance:

```
"Message ""Time to go home!"""
```

The string part of this is

```
 Message "Time to go home!"
```

3.2.5 Variables

Variables are a way of storing values that can change in the program during execution. A variable names a storage location. The *value* of the variable is stored in this location. Here are two types of variables:

- Simple variables such as: AVERAGE and NAME$
- Array elements such as: AVERAGE(3) and NAME$(2,12)

Two other constructs are used in ways similar to variables—you can use them in expressions and assign values to them:

- Substrings such as: NAME$[7:12]
- Structure references such as: SALES(AVERAGE) and CLIENT(PHONE)

Variables are represented by a name consisting of a letter or a series of letters, numbers and underscore characters. Variables:

- must start with a letter.
- can be followed by a proper data type designation (% for integers and $ for strings).
- can be declared explicitly.

Some examples of valid variables are:

```
TODAY$          X%
Z               INDEX
```

Variables can be integer numeric, real numeric or string. Integer variables can only contain integer values. If a real numeric value is assigned to an integer variable, INTOUCH rounds the real number and uses the resulting integer.

String variable names can have a trailing dollar sign ($). If numeric data is assigned to a string variable, an exception occurs.

```
TODAY$          X$
LAST_NAME$      ANSWER$
```

A variable that is not followed by a $ or a % is treated as a real numeric variable unless it is declared. If you assign string data to a real numeric variable, an error will result. If you assign integer data to a real numeric variable, INTOUCH treats it as a real number with a fractional part of 0. The following are examples of real numeric variables:

```
INDEX           Z
COUNTER         AMOUNT
```

INTOUCH uses the last assigned value when a variable is referenced.

3.2.5.1 Arrays

An **array** is a variable that consists of a group of elements. Each element represents a storage location. A different value can be assigned to each element. Here is an example of an array:

```
QUANTITY(1)
QUANTITY(2)
QUANTITY(3)
QUANTITY(4)
QUANTITY(5)
```

To indicate which element you want to access, you use numbers after the array name called *subscripts*. The subscripts are enclosed in parentheses. Example: AMOUNT(3,2)

Subscripts must be given as integers. If a real numeric subscript is given, it will be rounded and the integer portion used. Arrays can contain real numeric, integer or string values. String and integer arrays are designated by placing the appropriate symbols after the array name and before the subscript. For example:

```
A$(5,10)        A%(5,10)
```

About arrays:

- Arrays must be defined with the DIM statement and can be redefined with the REDIM statement.
- Arrays can have up to 32 dimensions.
- An array can have as many elements as memory allows.
- Integer and string arrays must be designated by the appropriate symbol (% for integers, $ for strings) or else they must be declared.
- Subscripts must be integers. Real subscripts are rounded.
- Multiple subscripts must be separated by commas.
- The default low bound for an array is 1 or the number indicated with the OPTION BASE statement. (See Section 10.1.3)
- You can indicate the low bound explicitly with a TO clause:
  DIM COSTS(5 TO 10)

3.2.5.2 Substrings

Substrings are a way to refer to a part of a string. The format of a substring is:

```
str_var [begin : end]
```

str_var is the name of a string variable. *begin* is the position in the string where your substring begins. *end* is the position at which the substring ends. For example, here is a string called ALPHABET$:

```
10  alphabet$ = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
20  PRINT alphabet$[9:14]
30  END

RNH
IJKLMN
```

The substring ALPHABET$[9:14] tells INTOUCH to begin at the ninth character and count to the 14th character. Everything between and including these two positions makes up the substring. Therefore, the substring is "IJKLMN".

begin and *end* are integers. If you give real numbers for these positions, INTOUCH rounds them and uses the remaining integers. You can manipulate a substring like any other expression. You can store data in the substring. And, you can use substrings to change the value of a string. For example:

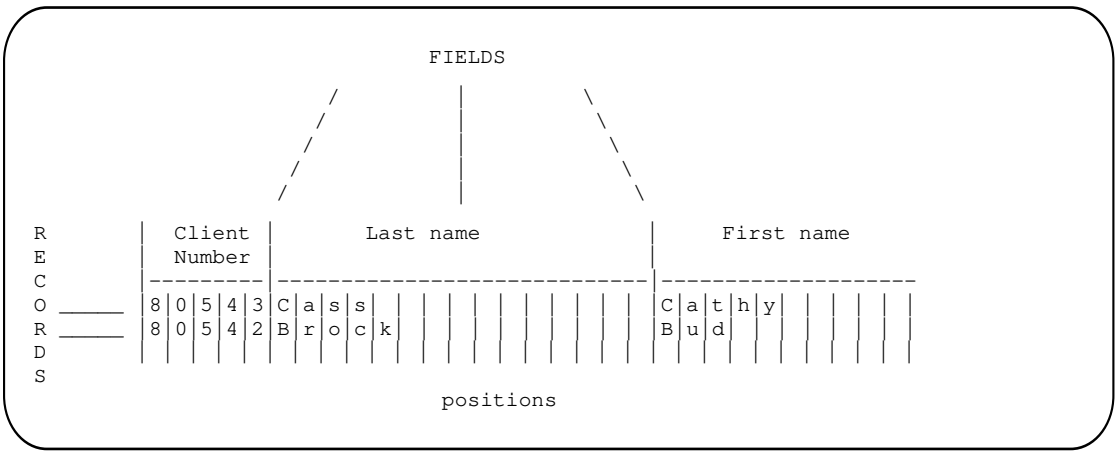
```
10  LET a$ = 'Your tests are in.'
20  PRINT a$
30  LET a$[6:10] = 'results'
40  PRINT a$
50  END

RNH
Your tests are in.
Your results are in.
```

3.2.6 Structure References

Another type of variable is a **structure reference**. INTOUCH includes a transparent interface to several record management systems, including the VMS file management system. One of the major features of INTOUCH is its ability to perform database operations as a part of the language. INTOUCH's data structure statements allow you to manipulate stored data from within your own programs. (See Chapter 14, Data Structure Statements for informaton on the INTOUCH data structure statements.)

INTOUCH stores data in **structures**. Structures look something like this:

Example 3–2: Structures



Each structure is made up of **records** and **fields**. In the CLIENT structure above, we have a record for each customer.

Each record consists of fields. For example, our customer records might contain a field for the customer's ID number, last name, first name, address, phone number, company name, etc.. Each of these pieces of data is stored in its own field–the name field, address field, phone number field, etc.. These fields appear as columns in the example shown above.

For information on creating structures and defining fields, see Chapter 16, Creating Structures, Field Definitions with SETUP.

You can reference the field data in structures by using structure references. To reference a field, indicate the structure name and the expression of the field whose contents you want to access.

```
struc_name(field_expr)
```

struc_name is the name associated with the structure. *field_expr* is the name of a field in the structure. When you reference a field, INTOUCH searches the current record for this field and reads its contents. Some examples of structure references are:

```
CLIENT(PHONE)          CATALOG(PART_NUM)
```

The *field_expr* can be either a string or numeric expression.

You can use a string constant to specify the field name. If you give the field name as a string constant, you need not enclose it in quotes. INTOUCH will use the string constant as the field name:

```
            PRINT CL(LAST)
                      /
 the field is specified by its field name
```

If you specify the field as an expression, you will need to precede the expression with a pound sign (#). The pound sign tells INTOUCH that the following characters are an expression, not the field name. If you do not include the pound sign, INTOUCH will interpret the characters as a field name. Here are two examples:

```
            PRINT CL(#FIELDNAME$)
                       /
 the field is specified by the variable FIELDNAME$

            PRINT CL(#FIELDNUM)
                      /
 the field is specified by the variable FIELDNUM
```

See Section 14.8.1.1, FIELD Expressions for an example of a program that uses field expressions.

3.2.7 Multiple Occurrence Fields

Fields with multiple occurrences (single dimension array) are supported.

When defining a field with multiple occurrences, the length of the field must be the length of a single occurrence. Individual occurrences of a field are accessed by including the occurrence number in the field expression.

```
10  OPEN STRUCTURE cust: NAME 'tti_run:customer', ACCESS INPUT
20  EXTRACT STRUCTURE cust
      PRINT cust(address#1)
      PRINT cust(address#2)
    END EXTRACT
30  END

RNH
10010 Sunset Cliffs Blvd.

1122 Monroe Ave.
PO Box 8765
11A SE. Hwy A1A
PO Box 11A-A
2111 Brawley Blvd.
        .
        .
        .
999 World Vista Avenue

#2 Bougainvillea Blvd.
```

3.2.8 Compound Expressions

You can use **compound expressions** in your programs. Compound expressions consist of operators and operands. There are three types of compound expressions:

Numeric expressions
String expressions
Conditional expressions

3.2.8.1 Numeric Expressions

Numeric expressions consist of numeric (integer or real) variables, constants or expressions separated by arithmetic operators. The arithmetic operators are +, -, *, /, and ^.

```
                    Constants    Variables

+          Add
                    4%+2%        Z + TWO16
-          Subtract
                    4%-2%        Z - TWO16
/          Divide
                    4%/2%        Z / TWO16
*          Multiply
                    4%*2%        Z * TWO16
^          Raise to a power
                    4%^2%        Z ^ TWO16
```

You can combine any number of these operators in an expression.

```
4 + Z ^ TWO16                Z * TWO16 / 2
```

You cannot generally use two arithmetic operators next to each other. However, you can use a + or - sign to indicate a positive or negative number. For example:

```
TOTAL * -2   =   TOTAL * (-2)
TOTAL / +2   =   TOTAL / (+2)
```

If all the values in an arithmetic expression are of the same data type, the result of the expression will be that data type. For example, if an expression consists only of integer numbers, it will yield an integer result. If an expression consists only of real numbers, the result will be a real number. If an expression consists of integers and real numbers, the result will be a real number. If the target of a real calculation is an integer (a% = 1.5 + 2.8) the result is rounded before it is assigned to the target.

3.2.8.2 String Expressions

String expressions are strings concatenated (joined). String expressions can be joined by a plus sign (+) or by an ampersand (&). INTOUCH evaluates this type of string expression by concatenating the strings. For example:

```
10  z$ = 'MO' + 'TH' & 'ER'
20  PRINT z$
30  END

RNH
MOTHER
```

In the above example, INTOUCH joins the strings separated by a plus sign and an ampersand, and assigns their value to Z$. You can include string constants, variables, functions, etc. in your string expressions. For example:

```
10  LET last$ = ' is it.'
20  PRINT 'This' + last$
30  END

RNH
This is it.
```

3.2.8.3 Conditional Expressions

Conditional expressions are expressions which yield a **TRUE** (1) or **FALSE** (0) value. Conditional expressions are created by using either relational or logical operators. When INTOUCH evaluates a conditional expression, it returns a value of either TRUE or FALSE. If the expression is TRUE, INTOUCH returns the integer 1. If the expression is FALSE, INTOUCH returns the integer 0.

Conditional Numeric Expressions

Relational operators are similar to those used in algebra. The relational operators are:

=	equals	X=Y	X is equal to Y
<	less than	X<Y	X is less than Y
>	greater than	X>Y	X is greater than Y
<=	less than or equal to	X<=Y	X is less than or equal to Y
>=	greater than or equal to	X>=Y	X is greater than or equal to Y
<>	not equal to	X<>Y	X is not equal to Y

X and Y can be any unconditional or conditional expression.

Performing Relational Operations on Strings

When you perform relational operations on strings, INTOUCH determines which string occurs first in the the ASCII collating sequence and returns **TRUE** or **FALSE**. For instance, when you perform relational operations on two strings, INTOUCH checks the ASCII values for each character in each string. INTOUCH compares the strings character by character–using these ASCII values–and determines where there is a difference in the values.

When INTOUCH finds a character that differs, it compares the two and determines which one is less (which has a smaller ASCII code number). INTOUCH then returns a TRUE or FALSE value depending on the relational expression. For example:

```
10  a$ = 'TEXT'
    b$ = 'TEST'
    MESSAGE$ = 'Strings are equal'
20  IF  a$ < b$  THEN  message$ = a$ + ' is less than ' + b$
    IF  b$ < a$  THEN  message$ = b$ + ' is less than ' + a$
30  PRINT message$
40  END

RNH
TEST is less than TEXT
```

INTOUCH compares the two strings. They are identical up to the third character. The ASCII value of S is 53. The ASCII value of X is 58. Therefore INTOUCH prints "TEST is less than TEXT".

Logical Operators

The logical operators are:

NOT	NOT X	**TRUE** if X is false and **FALSE** if X is true.
AND	X AND Y	**TRUE** if X and Y are true.
OR	X OR Y	**TRUE** if X or Y is true.
XOR	X XOR Y	**TRUE** if X is true, or if Y is true but **FALSE** if both X and Y are true.
EQV	X EQV Y	**TRUE** if X and Y are true, or **TRUE** if X and Y are false, but **FALSE** otherwise.
IMP	X IMP Y	**TRUE** if X is true and Y is false.

X and Y can be any expressions. Logical operators are usually used on integers or expressions which yield an integer result such as conditional expressions. Logical operators will always yield an integer result. If a logical operator is used on a real number, the real number is rounded and the resulting integer is used.

Logical expressions always return an integer value. If the integer value is a 1, the expression is TRUE. If the integer value is a 0, the expression is FALSE. (NOT 0 is equal to -1 and is TRUE. NOT 1 is equal to -2 and is FALSE.)

VALUE	TRUE	FALSE
0		X
1	X	
NOT 0 (-1)	X	
NOT 1 (-2)		X

- If a logical operator is used on a real expression, the expression is rounded and the resulting integer value is operated upon.
- Integers are represented as a signed 32-bit quantity.

Bit Manipulation

Logical operators can be used to do **bit manipulation**. Computers represent values in a binary code, using ones and zeros. INTOUCH integer values are represented as a 32-bit binary longword. A bit which is set to 1 is considered on. A bit which is set to 0 is off. The value of the word is equal to the value of all the bits which are on, added together. For example:

```
0 0 0 1 0 1 1 1  =  16 + 4 + 2 + 1 = 23
```

The last bit has a value of 1. The second to the last bit has a value of 2. The third bit has a value of 4, the fourth a value of 8, the fifth bit has a value of 16, and so on. Each bit has a value double that of the previous one.

```
       0   0   0   0   0   0   0   0
-----------------------------------------
     128  64  32  16  8   4   2   1
```

Bits can be manipulated and tested using logical operators. The logical operators work on bits. They compare each position in each word according to the particular rules of the logical operator. For instance, here is the **AND** operator used on two values:

```
10  LET a% = 23%       ! 00010111
20  LET b% = 37%       ! 00100101
30  LET c% = (a% AND b%)
40  PRINT c%
50  END

RNH
 5
```

When INTOUCH executes this program, it compares the two values. It sets a bit in the result to 1 (on), only if both the bits at a given position are on (1). The value of the resultant word is 5.

```
A%     0 0 0 1 0 1 1 1 = 23
B%     0 0 1 0 0 1 0 1 = 37
       ---------------
C%     0 0 0 0 0 1 0 1 =  5
```

3.2.9 Order of Evaluation

When INTOUCH evaluates an expression, it evaluates it in a specific order. INTOUCH evaluates expressions from left to right.

```
1+Z+4                 equals          (1+Z)+4
1+Z-4                 equals          (1+Z)-4

3*4/QUANTITY          equals          (3*4)/QUANTITY
12/QUANTITY*3         equals          (12/QUANTITY)*3
```

The following priorities take precedence over the left to right evaluation rule:

1 INTOUCH always evaluates expressions in parentheses first. Parentheses, ( ), can be used to change the order of any of the following operations. If parentheses are nested, INTOUCH evaluates them from the inside out. For example:

```
Z%-(X% / (Y% + AMOUNT))
```

INTOUCH evaluates the expression Y% + AMOUNT first. Next, it divides the X% by AMOUNT to determine that result. Finally, it subtracts the entire sum from Z%.

2 INTOUCH performs functions second.

3 INTOUCH performs exponentiation.

4 INTOUCH performs multiplication and division.

5 INTOUCH performs addition and subtraction.

6 INTOUCH performs relational operations from left to right. (The relational operators are: =, <, >, <=, >= and <>.) The only exception is the assignment of the result. The result is always assigned last.

7 INTOUCH performs logical operations in the following order:

NOT
AND
OR
XOR
IMP
EQV

4 Writing and Debugging INTOUCH Programs

This chapter describes some statements that are used in basically all INTOUCH programs. It also describes the INTOUCH debug facilities and how to use them.

4.1 Comments

You might want to include comments in programs. Comments are not executable statements. They are simply included in source code for informational purposes. They are seen when a program is listed or printed out. However, INTOUCH will ignore them when it executes a program.

There are two types of comments allowed in INTOUCH: REM comments and exclamation points (!). The following example shows each of these statements in use:

```
10  DIM name$(10)                                  ! Setup array
20  REM Main logic
    FOR i = 1 TO 10                                ! Begin the loop
      INPUT 'Please enter your name': name$(i)      ! Ask for a name
      IF  _EXIT  THEN  EXIT FOR                    ! End if they want
      PRINT 'Hello, '; name$(i)                    ! Print hello
    NEXT i                                         ! End the loop
30  END

RNH
Please enter your name? Mary
Hello, Mary
Please enter your name? exit
```

4.1.1 REM

FORMAT:

```
REM comment_text
```

EXAMPLE:

```
10  REM Get a name
    INPUT 'Please enter your name': name$
20  PRINT 'Hello, '; name$
30  END

RNH
Please enter your name? Lucy
Hello, Lucy
```

PURPOSE:

Use **REM** to put remarks on program lines. You can use these remarks to clarify your code.

DESCRIPTION:

When a program is listed on the screen or printed out, the REM lines are displayed exactly as they were written in the source code.

A REM statement must be placed at the beginning of a program line. INTOUCH ignores the REM statement and everything after it up until the end of the line and it continues execution at the next executable line. REM can be used anywhere in a multiple line statement. For example:

```
10  INPUT 'Please enter your name': name$
    REM Print hello
    PRINT 'Hello, '; name$
20  END

RNH
Please enter your name? Lucy
Hello, Tom
```

When the above program is run, INTOUCH executes the INPUT statement, ignores the REM statement, and then executes the PRINT and END statements.

4.1.2 ! comment_text

FORMAT:

```
! comment_text
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$    ! Ask for a name
    PRINT 'Hello, '; name$                   ! Say hello
20  END

RNH
Please enter your name? Mike
Hello, Mike
```

PURPOSE:

Use the **!** to put comments in a program.

DESCRIPTION:

You can use comments to clarify parts of your program as shown in the example above.

When the program is listed or printed, the "!" line is displayed as it was written in the source code.

When INTOUCH executes the above program, it executes the INPUT statement, ignores the "!" and the comment text following it and continues execution at the PRINT statement. The "!" does not have to be placed at the beginning of a physical line. It can be used anywhere on a program line.

The Exclamation Point with Line Continuation

The exclamation point can be used after an ampersand to document continued lines. When a line is continued with an ampersand, any comments must **follow** the ampersand. For example:

```
10  INPUT a$
    IF  a$ = ''  THEN  PRINT a$; &     ! Here is the trailing
        ' is OK.'                      ! comment text
20 END
```

4.1.3 ~ Used to Border Part of a Program

EXAMPLE:

```
10  DIM name$(10)                                  ! Setup array
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
20  REM Main logic
    FOR i = 1 TO 10                                ! Begin the loop
      INPUT 'Please enter your name': name$(i)     ! Ask for a name
      IF  _EXIT  THEN  EXIT FOR                    ! End if they want
      PRINT 'Hello, '; name$(i)                    ! Print hello
    NEXT i                                         ! End the loop
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
30  END
```

PURPOSE:

Use the **TILDE** to highlight and/or border parts of your program source code.

DESCRIPTION:

The tilde can be used to clarify or highlight code. Tildes appear when the program is listed or printed but INTOUCH ignores them when it executes the program. Tildes within quotes ('~~') are treated as string constants.

4.2 PROGRAM, END, STOP and HALT Statements

4.2.1 PROGRAM

FORMAT:

```
PROGRAM prog_name
```

EXAMPLE:

```
10  PROGRAM DISPLAY_NAME
20  INPUT 'Please enter your name': name$
    PRINT 'Hello, '; name$
30  END
```

PURPOSE:

Use the **PROGRAM** statement to name your program.

DESCRIPTION:

PROGRAM is used to name programs. *prog_name* is the program name. The program name must meet the specifications for variable names. It must:

- start with a letter.
- can consist of letters, numbers, and/or underscore characters (_).
- can be up to, but not longer than 39 characters.

4.2.2 END

FORMAT:

```
END
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
20  PRINT 'Hello, '; name$
30  END

RNH
Please enter your name? John
Hello, John
```

PURPOSE:

Use the **END** statement to end program execution. If you have *EXTERNAL* subprograms or functions, use END to mark the end of the main program unit.

DESCRIPTION:

The END statement marks the end of a program. When INTOUCH executes the END statement, it:

- Writes all active output buffers and closes all files in the current program.
- Exits the current program and returns to the INTOUCH prompt if in the environment.

The END statement does not have to be the last physical line in a program. If you have subprograms, functions, etc., they can physically follow the END statement. However, END marks the end of the source code for the main program unit.

4.2.3 STOP

FORMAT:

```
STOP
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    INPUT 'How old are you': age
    IF  age < 1  THEN
      PRINT 'Not a valid age'
      STOP
    END IF
20  PRINT name$; ' is'; age
30  END

RNH
Please enter your name? Ted
How old are you? 38
Ted is 38
```

PURPOSE:

Use **STOP** to terminate program execution where you do not want to mark the physical end of your program.

DESCRIPTION:

STOP behaves exactly as the END statement does. However, STOP does not mark the end of a program.

STOP ends program execution and returns control to the INTOUCH environment. For instance, the example program would run as follows:

```
RNH
Please enter your name? Ted
How old are you? .5
Not a valid age

INTOUCH
```

4.2.4 HALT

FORMAT:

```
HALT
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    INPUT 'How old are you': age
    IF  age < 1  THEN
      PRINT 'Not a valid age'
      HALT
    END IF
20  PRINT name$; ' is'; age
30  END

RNH
Please enter your name? Tex
How old are you? 0
Not a valid age
Halt at 10.4

INTOUCH
```

PURPOSE:

Use **HALT** if you want to interrupt program execution, check values, and then continue execution.

DESCRIPTION:

HALT interrupts program execution, but it does not close any files, nor does it write the active output buffer. HALT interrupts program execution and returns to the INTOUCH prompt. Execution can be continued with the GO command.

The EXAMPLE program would run as follows:

```
RNH
Please enter your name? Tex
How old are you? 0
Not a valid age
Halt at 10.4

INTOUCH

let age = 34

INTOUCH

go
Tex is 34
```

4.2.5 RANDOMIZE

FORMAT:

```
RANDOMIZE
```

EXAMPLE:

```
10  RANDOMIZE
20  x = RND
    PRINT x
30  END

RNH
 .244013674718

INTOUCH

RNH
 .524856061388
```

PURPOSE:

RANDOMIZE gives the RND function a new starting point. This ensures that you will get a different random number sequence each time your program executes.

DESCRIPTION:

INTOUCH uses a list of random numbers when the RND function is used. If no RANDOMIZE statement is used, INTOUCH starts at the same place in this list each time the program executes. This means the same series of random numbers is returned each time the program executes.

RANDOMIZE tells INTOUCH to pick a random starting point in the list. This ensures that a different series of random numbers is returned each time the program executes. (See RND for information on the RND function.)

4.3 Debug Facilities

INTOUCH detects and announces run-time and compile-time errors. Sometimes errors occur which don't prevent execution, but cause a program to execute incorrectly. INTOUCH provides a high-level DEBUG system for detecting these more subtle errors.

DEBUG ON enables INTOUCH's Debug System. DEBUG OFF disables the system. When DEBUG is enabled, a DEBUG flag will appear at the status message in the upper right corner of the INTOUCH screen. When DEBUG is disabled, the flag disappears from the screen. DEBUG ON turns on the frame if it was off.

The related function for the INTOUCH Debug System is _DEBUG. See _DEBUG for information on the _DEBUG system function.

Some DEBUG features automatically switch DEBUG ON or OFF when they are executed. Others require that DEBUG be enabled. (See DEBUG ON.)

Note

Using the debugging facilities impacts the INTOUCH system's performance. The system has been designed for optimal performance in a normal production environment. Therefore, DEBUG will slow INTOUCH response time.

4.4 DEBUG ON/OFF

4.4.1 DEBUG ON

FORMAT:

```
DEBUG ON
```

EXAMPLE:

```
DEBUG ON
1050 DEBUG ON
```

PURPOSE:

Use **DEBUG ON** to enable INTOUCH's debug system. The DEBUG system helps you find problems in the way your program runs. You must enable the system in order to use its features.

DESCRIPTION:

When DEBUG is enabled, all of INTOUCH's DEBUG features are available. DEBUG remains enabled until a DEBUG OFF command or statement is executed or until a DEBUG feature is executed which disables it.

DEBUG ON can be issued in immediate mode or as a statement in a program. If DEBUG ON is used in a program, INTOUCH enables DEBUG when it encounters the DEBUG ON statement. After you turn DEBUG ON, you must use RUN or RNH to start the program before you can use the STEP statement.

4.4.2 DEBUG OFF

FORMAT:

```
DEBUG OFF
```

EXAMPLE:

```
DEBUG OFF
1050 DEBUG OFF
```

PURPOSE:

Use **DEBUG OFF** to disable INTOUCH's DEBUG system. You should turn DEBUG OFF when you have finished correcting your program.

DESCRIPTION:

DEBUG OFF can be issued in immediate mode or as a statement in a program. If DEBUG OFF is used in a program, INTOUCH disables DEBUG when it encounters the DEBUG OFF statement.

DEBUG will remain disabled until a DEBUG ON statement is executed, or until a DEBUG feature is executed which enables it.

4.5 Monitoring Program Execution

DEBUG's TRACE and STATISTICS features monitor your program's execution. These features take effect only during program execution.

TRACE traces your program line by line as it executes. TRACE lists the number of each program line as it is executed. STATISTICS records information on your program's execution. It records the time each program line takes to execute and the number of times the line is executed. The word "STATISTICS" can be abbreviated to "STATS" (STATS ON, STATS OFF, LIST STATS). This abbreviation will be used frequently in this Guide.

4.6 The TRACE and STATISTICS Features

Both the TRACE and STATISTICS features operate on program lines. A number of program lines can be attached to one line number.

```
10  PRINT a$
    PRINT b$
    PRINT c$
```

The TRACE feature notes these program lines by line number. For instance:

```
10  PRINT a$ --- 10
```

If more than one program line is attached to a line number, the program lines are denoted by a period and the number of the sub-line.

```
10  PRINT a$ --- 10
    PRINT b$ --- 10.1
    PRINT c$ --- 10.2
```

In the example above, the first line is noted as 10. The program line following the first line is 10.1. The next line is 10.2 and so on. Labels are noted in the same way. For example:

```
10  show: PRINT a$ --- SHOW
    PRINT b$       --- SHOW.1
    PRINT c$       --- SHOW.2
```

4.6.1 TRACE ON

FORMAT:

```
TRACE ON
```

EXAMPLE:

```
TRACE ON
1050 TRACE ON
```

PURPOSE:

Use **TRACE ON** to turn INTOUCH's trace feature on. The trace feature lists the number of each program line as it is executed. Use TRACE to see if your statements are being executed in the correct order and the correct number of times.

DESCRIPTION:

To use TRACE, you must have the frame on.

TRACE ON enables INTOUCH's trace feature. The trace feature lists each program line as it is executed. TRACE ON can be executed as a command or as a statement. If TRACE ON is executed as a command, DEBUG is automatically switched on. If TRACE ON is executed as a statement, and DEBUG is off, INTOUCH ignores the TRACE ON. The following program is executed with the trace feature on:

```
10  DIM name$(5)
    MAIN: FOR i = 1 TO 5
            INPUT 'Please enter your name': name$(i)
            IF  _EXIT  THEN  EXIT FOR
            PRINT 'Hello, '; name$(i); '!'
          NEXT i
20  END
```

Example 4–1: Debug TRACE ON Example

```
INTOUCH 4.2                    Program: NONAME                              DEBUG
The Next Generation Language   Status :   RUN


INTOUCH

TRACE ON

INTOUCH

RNH
Please enter your name? John
Hello, John!
Please enter your name?
                   [10]   [MAIN]   [MAIN.1]   [MAIN.2]   [MAIN.3]   [MAIN.4]   [MAIN.1]

EXIT = Exit                                                  \ = Back  HELP = Help
```

The trace remains in effect until a TRACE OFF command or statement is executed. When program execution stops, the trace stops. If program execution continues and the TRACE ON command is still in effect, the trace continues.

4.6.2 TRACE OFF

FORMAT:

```
TRACE OFF
```

EXAMPLE:

```
TRACE OFF
1050 TRACE OFF
```

PURPOSE:

Use **TRACE OFF** to turn the trace feature off.

DESCRIPTION:

TRACE OFF disables the trace feature. INTOUCH stops listing program line numbers when TRACE OFF is executed. If TRACE OFF is executed as a statement in a program and DEBUG is off, INTOUCH ignores the statement.

4.6.3 STATS ON

FORMAT:

```
STATS ON
```

EXAMPLE:

```
STATS ON
1020 STATS ON
```

PURPOSE:

Use **STATS ON** to turn on the statistics feature, which stores the execution time and count for each program line. Use STATS to tell if your statements are being executed the correct number of times and which parts of your program are taking the most time. STATS is especially useful for speeding up your program's execution time.

DESCRIPTION:

STATS ON enables INTOUCH's statistics feature. INTOUCH begins recording statistics when program execution begins. The statistics feature remains enabled until the STATS OFF statement is executed.

STATS ON can be executed in immediate mode or in a program. If STATS ON is executed in immediate mode, DEBUG is automatically switched on. If STATS ON is executed in a program, and DEBUG is off, INTOUCH ignores the statement. When STATS ON is executed, any statistics previously recorded are lost.

4.6.4 STATS OFF

FORMAT:

```
STATS OFF
```

EXAMPLE:

```
STATS OFF
1020 STATS OFF
```

PURPOSE:

Use **STATS OFF** to turn off the statistics feature.

DESCRIPTION:

STATS OFF turns off INTOUCH's statistics feature. STATS OFF can be executed in immediate mode or in a program. If STATS OFF is executed in a program and DEBUG is off, INTOUCH ignores the statement. STATS OFF leaves DEBUG on.

4.6.5 LIST STATS

FORMAT:

```
LIST STATS [:label | line_num [- | ,] label | line_num...]
```

EXAMPLE:

```
LIST STATS
```

PURPOSE:

Use **LIST STATS** to display the statistics recorded by the STATISTICS feature.

DESCRIPTION:

LIST STATS lists each program line along with the number of times the line was executed and the execution time of each line.

Example 4–2: Listing Statistics in DEBUG System

```
INTOUCH 4.2                    Program: NONAME                    STATS    DEBUG
The Next Generation Language   Status :   RUN

INTOUCH

LIST STATS
NONAME          4-MAY-1995 00:54
          1       0.00        10  DIM name$(5)
          1       0.00            MAIN: FOR i = 1 TO 5
          5      16.64                    INPUT 'Please enter your name': name$(i)
          5       0.00                    IF  _EXIT  THEN  EXIT FOR
          5       0.00                    PRINT 'Hello, '; name$(i); '!'
          5       0.00                  NEXT i
          1       0.00        20  END

INTOUCH


EXIT = Exit                                              \ = Back  HELP = Help
```

The far-left column lists the number of times each statement was executed. The next column gives the time each statement took to execute. The time is given in seconds and fractions of a second. (0.01 means the program line was executed in one-one hundredth of a second.) The last column lists the program itself. STATS must be ON for LIST STATS to be executed.

All the options available with the "LIST" statement are also available with LIST STATS. (See Section 2.3.7, LIST for listing information.)

4.7 BREAK and STEP

4.7.1 BREAK

FORMAT:

```
BREAK
```

EXAMPLE:

```
1050 BREAK
```

Use **BREAK** to stop program execution when DEBUG is ON. For instance, you might use BREAK to stop the program if a variable is assigned a wrong value.

DESCRIPTION:

The BREAK statement can be used anywhere in a program. The BREAK statement will not take effect unless DEBUG is turned on. If DEBUG is off, INTOUCH ignores any BREAK statements.

When INTOUCH executes a BREAK statement, it interrupts program execution and prints a BREAK message. The BREAK message tells what line the break occurred in. Program execution can be continued with the GO or STEP commands.

Example 4–3: Using BREAK in DEBUG System

```
INTOUCH 4.2                    Program: NONAME                              DEBUG
The Next Generation Language   Status :  BREAK at 10.2
10  PRINT '1',
    PRINT '2',
    BREAK
    PRINT '3',
    PRINT '4',
    PRINT '5'
20  END
DEBUG ON

INTOUCH

RNH
1                   2                   BREAK at 10.2

INTOUCH

EXIT = Exit                                                  \ = Back  HELP = Help
```

4.7.2 STEP

FORMAT:

```
STEP [number]
```

EXAMPLE:

```
STEP
STEP 10
```

PURPOSE:

Use **STEP** to execute a specific number of program statements and then stop execution. That way, you can "step through" your program to find bugs.

DESCRIPTION:

STEP is used to step through your program—to execute a specified number of program statements. DEBUG must be ON for the STEP command to take effect. If DEBUG is not on and the STEP command is given, INTOUCH ignores it. STEP must be given as a command. When the STEP command has been executed, INTOUCH issues a BREAK and prints the break message. Issuing the STEP command without a number causes INTOUCH to execute one program line.

Example 4–4: Using STEP in DEBUG System

```
INTOUCH 4.2                    Program: NONAME                          DEBUG
The Next Generation Language   Status :  BREAK at 10.5

RNH
1                    2                   BREAK at 10.2

INTOUCH

STEP 2
3                    4
INTOUCH


                                  PRINT '5'
EXIT = Exit                                          \ = Back  HELP = Help
```

Issuing the STEP command with a number causes INTOUCH to execute the number of program lines specified. INTOUCH begins executing program lines from the last line executed. It stops when the number of lines specified have been executed or when program execution ends.

You must start a program with RUN or RNH before you can use the STEP statement.

4.8 GO–Continuing Program Execution

4.8.1 GO

FORMAT:

```
GO
```

EXAMPLE:

```
GO
```

PURPOSE:

Use **GO** to continue program execution after it has been interrupted.

DESCRIPTION:

Once execution has stopped, you can enter immediate mode and debugger commands, change code, etc. GO lets you resume execution even after code has been changed. If a HALT or BREAK statement was used, execution will resume at the first statement after the HALT or BREAK.

Example 4–5: Using GO in DEBUG System

```
INTOUCH 4.2                    Program: NONAME                          DEBUG
The Next Generation Language   Status :  BREAK at 20
10  DEBUG ON
    FOR i = 1 TO 6
20    IF  i = 4  THEN  BREAK
30  NEXT i
40  END
RNH
BREAK at 20

INTOUCH

PRINT SQR(i)
 2

INTOUCH

25  PRINT i;
GO
 4  5  6
INTOUCH

EXIT = Exit                                          \ = Back  HELP = Help
```

5 Inputting and Storing Terminal Data

This chapter describes the various ways that data can be entered at the terminal and stored into variables.

5.1 INPUT Statement

FORMAT:

```
[LINE] INPUT var, var...

[KEY] [LINE] INPUT [  ['Prompt_text']
  [, PROMPT str_expr] [, ERASE]
  [, AT row, column] [, LENGTH num_expr] [, DEFAULT str_expr]
  [, VALID str_expr] [, TIMEOUT time_limit] [, ELAPSED num_var]
  [, AREA num_expr, num_expr, num_expr, num_expr] [, ATTRIBUTES attr_list]
  [, SCREEN '[text] <format>...']
  [, MENU str_expr: str_var] :] var [,var. . .]
```

EXAMPLE:

```
10  INPUT 'Please enter your first name': first$
    INPUT 'Now, enter your last name': last$
    LINE INPUT 'Where do you live': city$
20  PRINT
    PRINT 'Hello '; first$; ' '; last$
    PRINT 'From '; city$
30  END

RNH
Please enter your first name? Sunny
Now, enter your last name? Day
Where do you live? San Diego, California

Hello Sunny Day
From San Diego, California
```

PURPOSE:

Use the INPUT statement to ask questions at the terminal and store the answers for use in your program.

DESCRIPTION:

The **INPUT** statement reads data typed by the user at the terminal and assigns it to variables. *var* is a variable that the data is being assigned to. When INTOUCH executes an INPUT statement, it prints any prompt given and waits for the user's response. The user's response is then assigned to the variable(s) specified.

For information on INPUT from a text file, see Chapter 15, File Handling.

The user enters data at the terminal in response to the INPUT statement. The input data must be the same data type as the variable, or INTOUCH generates an exception. If, in response to the INPUT statement, the user presses the Return key and:

- the variable is numeric, INTOUCH assigns it a 0.
- the variable is a string, INTOUCH assigns it a null string ("").

5.1.1 Types of INPUT Statements

There are three types of INPUT statements:

- **INPUT**

  The INPUT statement reads one or more comma-separated data items from the terminal.

- **LINE INPUT**

  The LINE INPUT statement reads an entire line of data from the terminal.

- KEY INPUT

  The KEY INPUT statement reads a single keystroke from the terminal.

5.1.2 INPUT Styles

There are four input styles:

1 Simple input from the terminal.

```
10  INPUT 'Please enter your name': name$
20  PRINT 'Hello '; name$
30  END

RNH
Please enter your name? Toby
Hello Toby
```

2 Formatted data entry screens (see Section 5.16, SCREEN Option).

```
10  INPUT 'Please enter your name': name$
20  INPUT SCREEN 'Soc. sec.: <DIGITS: ###-##-####>': ssn
30  PRINT name$, 'Social security number:'; ssn
40  END

RNH
Please enter your name? Fred
Soc. sec.:  ___-__-____        (before input)
Soc. sec.:  324-11-4533        (after input)
Fred               Social security number: 324114533
```

3 Free format multi-line text input (see Section 5.15, AREA Option).

```
10  CLEAR
    MESSAGE 'Press GOLD/F when done'
20  LINE INPUT AREA 5, 10, 8, 60: text$
30  PRINT AT 10, 1: 'Rewrapped text'
40  wt$ = WRAP$(text$, 1, 30)
50  PRINT wt$
60  END

RNH

This is an example of wrapped text.  The text is
wrapping._____________________________________________
_____________________________________________________
_____________________________________________________

Rewrapped text
This is an example of wrapped
text.  The text is wrapping.
```

4 Pop-up menus (see Section 5.17, MENU Option).

```
10  sample_menu$ = '%TITLE "Options",' &
        + 'Add, Change, Delete, Inquire'
20  LINE INPUT MENU sample_menu$: selection$
30  PRINT 'Menu option was '; selection$
40  END

RNH

  +--Options---+
  |  ADD       |
  |  CHANGE    |
  |  DELETE    |
  |  INQUIRE   |
  +------------+

Menu option was CHANGE
```

5.2 INPUT Options

The INPUT statement has the following options. These options are described in detail in following sections of this chapter.

PROMPT	displays the prompt text
AT row, col	positions the cursor on the desired row and column
LENGTH nn	limits the number of characters that a user can type
DEFAULT	lets you provide defaults for INPUT statements
VALID	validates user responses
TIMEOUT	limits the time the user has to respond to the INPUT prompt
AREA	does free format multi-line text input from an area on the screen
SCREEN	creates formatted data entry screens
MENU	displays and receives input from "pop-up" menus
ELAPSED	keeps track of the time it takes the user to respond to an INPUT prompt
ATTRIBUTES	displays in BOLD, BLINK, REVERSE, UNDERLINE

ERASE	clears the input line prior to accepting input and after the input has been completed

5.3 INPUT Commands and Keystrokes Available to the User

At an input prompt, there are several commands and keystrokes that the user can enter instead of an answer to the prompt. These are:

EXIT or Ctrl/Z

You can type the word **EXIT** or press Ctrl/Z to exit from a prompt or procedure. If one of these options is entered, INTOUCH sets the internal variable **_EXIT** to TRUE (1).

\ (backslash) or UP-arrow

If you enter **** (backslash) or press the **Up-arrow** key, the internal variable **_BACK** will be set to TRUE (1).

HELP or Help

You can get help for an input item by typing the word **HELP** or by pressing the Help key. If one of these options is input, INTOUCH sets the internal variable **_HELP** to TRUE (1).

The internal variables _BACK, _EXIT and _HELP can be examined within your program and appropriate action can be taken.

5.3.1 Editing Input

If the user wants to edit their input, they can use the following keys:

Left-arrow	move left
Right-arrow	move right
Delete key	delete character(s) to the left of the cursor
TAB	move to next input field or terminate input
CTRL/A	toggle between insert and overstrike mode
CTRL/E	move to end of line
CTRL/H	move to beginning of line
CTRL/J	delete the word to the left of the cursor.
CTRL/U	delete all the characters on a line.
Return	terminate the input; you can press Return anywhere in the line, you do not have to be at the end of the line.

5.4 Inputting Strings

If the INPUT is to a string variable, the user can enter an unquoted or a quoted string. If the user enters an unquoted string, INTOUCH ignores any leading spaces, trailing spaces, or tabs. An unquoted string cannot contain commas.

If the user enters a quoted string, INTOUCH removes the quotes, and includes any leading or trailing spaces and tabs.

```
10  INPUT 'Enter your name': name1$
    INPUT 'Enter your name in quotes': name2$
    INPUT 'Enter your name in quotes with spaces': name3$
    INPUT 'Enter last name, comma, space, first name in quotes': name4$
20  PRINT
    PRINT name1$
    PRINT name2$
    PRINT name3$
    PRINT name4$
30  END

RNH
Enter your name? Tony
Enter your name in quotes? 'Tony'
Enter your name in quotes with spaces? '  Tony  '
Enter last name, comma, space, first name in quotes? 'Smith, Tony'

Tony
Tony
  Tony
Smith, Tony
```

5.5 Inputting Multiple Variables

5.5.1 Simple Input Statement

A single INPUT statement can be used to input several variables. The input items and variables must be separated with commas.

```
10  INPUT 'Enter 3 names separated by commas': name1$, name2$, name3$
20  PRINT
    PRINT name1$, name2$, name3$
30  END

RNH
Enter 3 names separated by commas? Tom,Sue,George

Tom                 Sue                 George
```

If an INPUT statement contains a list of variables, INTOUCH asks for input until all of the variables have a value. If the user enters less data or more data than there are variables, INTOUCH generates an exception. If an exception occurs, INTOUCH restarts from the beginning.

Users can enter the data for a variable list in one of two ways.

1 The user can enter each piece of data on a separate line by typing a comma as the last character on each continuing line. The comma tells INTOUCH that there is more data to come.

```
RNH
Enter 3 names separated by commas? Tom,
? Sue,
? George

Tom                 Sue                 George
```

2 Users can enter the data on one line by separating the data with commas as shown in the first example above.

Whichever method is used, INTOUCH will continue accepting input data until all the variables have values.

5.5.2 LINE INPUT Statement

A number of variables can be input with one **LINE INPUT** statement. Simply list the variables separated by line terminators.

```
10  LINE INPUT 'Enter a comment, press Return, enter a comment': &
        comment_1$, comment_2$
    PRINT
    PRINT comment_1$
    PRINT comment_2$
20  END

RNH
Enter a comment, press Return, enter a comment? This is first comment
? This is second comment

This is first comment
This is second comment
```

INTOUCH asks for input until all of the variables listed have received a value.

Unlike the INPUT statement, you cannot separate answers with commas. Each variable is prompted for separately. If you include a comma in your response, the comma is just taken as part of the text.

5.6 INPUT Default Prompt and Text

By default, INTOUCH prints a question mark and a space and then waits for the user's response. However, you can display prompt text before the question mark. To display prompt text, enclose the prompt text in quotes and follow it with a colon. The colon separates the prompt text from the variable(s). The prompt text must follow the keyword INPUT and must be separated from the variable list by a colon.

When INTOUCH executes the INPUT statement, it prints the prompt text ("Your name" in the example below) followed by a question mark and a space.

```
10  INPUT 'Your name': name$
20  PRINT name$
30  END

RNH
Your name? Fred
Fred
```

5.7 PROMPT Option

The **PROMPT** option displays the specified prompt text *without* the question mark and space.

FORMAT:

```
PROMPT str_expr
```

str_expr is a string expression which contains the prompt text. *str_expr* can be any string expression. The prompt text is separated from the variable list with a colon (:).

EXAMPLE:

```
10  INPUT PROMPT 'Please enter your name: ': name$
20  PRINT 'Hello '; name$
30  END

RNH
Please enter your name: Jackie
Hello Jackie
```

5.8 AT Option

The **AT** option positions the cursor on the specified row and column. This is the position where the INPUT statement starts the prompt, not where the user enters data.

FORMAT:

```
AT row, col
```

row is the row to print at. *col* is the column to print at. *row* and *col* can be any integer numeric constants.

EXAMPLE:

```
10  CLEAR
    PRINT AT 1, 1:
20  INPUT AT 3, 10, PROMPT 'Please enter your name: ': name$
30  PRINT 'Hello '; name$
40  END

RNH
   (row 3, col 10)
         |
         V
         Please enter your name: Jack
Hello Jack
```

5.9 ATTRIBUTES Option

The **ATTRIBUTES** option allows input with attributes. The available attributes are:

BOLD
BLINK
REVERSE
UNDERLINE

Multiple attributes used in one INPUT statement are separated by commas.

FORMAT:

```
ATTRIBUTES attr_list
```

attr_list contains a list of input attributes.

EXAMPLE:

```
10  name_attr$ = 'BOLD, UNDERLINE'
20  LINE INPUT 'Enter your name', ATTRIBUTES name_attr$: name$
30  PRINT 'Hello '; name$
40  END

RNH
Enter your name? Susan
Hello Susan
```

5.10 LENGTH Option

The **LENGTH** option limits the number of characters that a user can enter. It causes INTOUCH to display underscore characters following the prompt. The number of underscore characters is the length.

FORMAT:

```
LENGTH num_expr
```

num_expr is the number of characters the user can enter.

EXAMPLE:

```
10  INPUT 'Enter your name', LENGTH 15: name$
20  INPUT 'Enter a city', LENGTH 20: city$
30  PRINT name$, city$
40  END

RNH
Enter your name? Betty__________
Enter a city? 'San Diego'_________
Betty              San Diego
```

5.11 DEFAULT Option

DEFAULT lets you provide defaults for INPUT statements. INTOUCH automatically formats the default appropriately. The user can press Return to accept the default you provide.

FORMAT:

```
DEFAULT str_expr
```

str_expr is a string expression that will be used as the default. When INTOUCH executes an INPUT statement with a default, it prints the default after the prompt.

EXAMPLE:

```
10  INPUT 'Enter the state code', DEFAULT 'CA': state$
20  PRINT 'The state was: '; state$
30  END

RNH
Enter the state code? CA
The state was: CA
```

If the user does not want the default, they can simply type over the default text.

5.11.1 Pre-positioning to a DEFAULT Menu Item

When performing an **INPUT MENU**, the **DEFAULT** option can be used to specify a default menu path. The default takes the format of:

```
#item1;#item2;...
```

#item1 is the number of the item on the first menu, *#item2* is the number of the item on the second menu, and so on.

Upon the completion of a INPUT MENU statement, the concept **_STRING** contains the menu path taken by the user when selecting the menu item. i.e. **#2;#3** means the 3rd item of the 2nd submenu. (For information on menus, see Section 5.17, MENU Option.)

EXAMPLE:

```
10  CLEAR
20  line1$ = '%WIDTH 12, %MENUBAR, %AUTOVBAR ON,'
    line2$ = 'file = {new, get_file, save, save_as},'
    line3$ = 'edit = {cut, copy, paste},'
    line4$ = 'paragraph = {font, alignment, spacing, tabs, headers, footers},'
    line5$ = 'options = {ruler = {on, off}, side_bar = {on, off},'
    line6$ = 'view = {enlarged, normal, small}},exit'
30  test_menu$ = line1$ + line2$ + line3$ + line4$ + line5$ + line6$
    the_default$ = ''
40  DO
      INPUT MENU test_menu$, DEFAULT the_default$: ans$
      IF  _EXIT  THEN  EXIT DO
      MESSAGE 'Menu path was', _STRING
      the_default$ = _STRING
    LOOP
50  END
```

```
+-------------------------------------------------------------------------------+
|  FILE     |   EDIT     |   PARAGRAPH   |  ~OPTIONS~  |   EXIT               |
+-------------------------------------------+---OPTIONS---+--------------------+
                                            |  RULER    [>|
                                            |  SIDE_BAR +----VIEW-----+
                                            | ~VIEW~    |  ENLARGED   |
                                            +-----------| ~NORMAL~    |
                                                        |  SMALL      |
                                                        +-------------+
```

Menu path was **#4;#3;#2**

5.12 ERASE Option

The **ERASE** option clears the input line prior to accepting input. After input has been completed, the input line is cleared again.

```
10  CLEAR
    PRINT AT 1,1:
20  INPUT 'Please enter a name', AT 3,1: name_1$
    INPUT 'Enter a 2nd name', AT 4,1, ERASE: name_2$
    PRINT '1st name: '; name_1$
    PRINT '2nd name: '; name_2$
30  END

RNH
Please enter a name? James

1st name: James
2nd name: Tony
```

5.13 VALID Option

The **VALID** option validates user responses according to specified validation rules.

FORMAT:

```
VALID str_expr
```

str_expr is the list of validation rules.

You can refer to the VALID( ) function in Appendix A, Built-in Functions for information on all the validation rules.

EXAMPLE:

```
10  INPUT 'Enter name', LENGTH 20: name$
20  INPUT 'Enter age', LENGTH 5, VALID 'INTEGER': age$
30  PRINT name$, age$
40  END

RNH
Enter name? Aaron_______________
Enter age? 32___
Aaron               32
```

5.14 TIMEOUT Option

The **TIMEOUT** option limits the time the user has to respond to the INPUT prompt. A time limit must be specified with the TIMEOUT option. If the user does not complete the INPUT statement within the specified time, an exception ("timeout on input at xx") is generated.

FORMAT:

```
TIMEOUT num_expr
```

num_expr is a numeric expression that represents the number of seconds allowed for the response.

EXAMPLE:

```
10  INPUT 'Name', TIMEOUT 4.5: name$
20  END

RNH
Name? Timeout on input at 10

INTOUCH
```

TIMEOUT 30 gives the user approximately 30 seconds to enter a name. Fractions of a second can be indicated by including decimal digits. TIMEOUT 4.5 gives the user approximately 4.5 seconds to enter a name.

5.15 AREA Option

The **AREA** option does input from an area on the screen.

FORMAT:

```
LINE INPUT AREA top, left, bottom, right: str_expr
```

Top	Top row of the area on the screen
Left	Left column of the area
Bottom	Bottom row of the area
Right	Right column of the area

EXAMPLE:

```
10  CLEAR
20  MESSAGE 'Press GOLD/F when done'
30  LINE INPUT AREA 5, 10, 8, 60: text$
40  PRINT AT 10, 1: 'Rewrapped text'
50  wt$ = WRAP$(text$, 1, 30)
60  PRINT wt$
70  END

RNH

        This is an example of wrapped text.  The text is
        wrapping._____________________________________________
        ______________________________________________________
        ______________________________________________________

Rewrapped text
This is an example of wrapped
text.  The text is wrapping.
```

During a LINE INPUT AREA, the following features are available:

- Automatic word-wrap while text is being entered
- Insert and delete capability
- Use of LEFT, RIGHT, UP and DOWN arrow keys

While you are entering text in an area, you can use the following commands:

Text	To enter text, just type it.
Arrow keys	To move around, press UP, DOWN, LEFT, RIGHT
CTRL/H	Moves cursor to beginning of line
CTRL/E	Moves cursor to end of line
GOLD/B	Moves cursor to the top of the area
GOLD/F	Complete (finish) and save the input
F10	Exit (abort) the input
CTRL/Z	Exit (abort) the input
Help	Exits from the input and sets the variable _HELP to true

When you finish a LINE INPUT AREA, INTOUCH removes any trailing line feeds and spaces.

5.16 SCREEN Option

The **SCREEN** option is used to create formatted data entry screens. The SCREEN option lets the user enter data into a fill-in area.

FORMAT:

```
SCREEN str_expr
```

str_expr specifies the screen format.

EXAMPLE:

```
10  INPUT 'Your name, please': name$
20  INPUT SCREEN 'Soc. sec.: <DIGITS:###-##-####>': SSN
30  PRINT name$, 'Social security number:'; SSN
40  END

RNH
Your name, please? John
Soc. sec.: ___-__-____
```

When the program executes, INTOUCH displays the prompt "Soc. sec.:" and a fill-in area with underscore characters where the user must fill in a number. The fill-in area is highlighted. As the user enters the social security number, it appears in the fill-in area and the underscores disappear. When the Return key is pressed, the number is assigned to the the variable SSN.

```
RNH
Your name, please? John
Soc. sec.: 555-48-6662
John                Social security number: 555486662
```

You can use the SCREEN option with any data type. A string expression follows the keyword SCREEN. The expression contains any text to print and the format for the fill-in areas.

A number of commands can be used within the format to control how the fill-in area works. These fall into two categories:

Format options

Format characters

Format options come first. They are followed by a colon and then by format characters.

5.16.1 Screen Format Characters

#

The # is used to specify *digits*. Wherever a # sign appears, only these types of characters are allowed as input. For example, the # might be used for a phone number:

```
10  INPUT SCREEN '<:(###) ###-####>': phone
20  PRINT phone
30  END
```

When INTOUCH executes this program, it displays the following fill-in area:

```
(___) ___-____

RNH
(555) 554-7879
5555547879
```

@

The @ is used to specify any printable character, including letters and numbers.

```
10  INPUT SCREEN 'License #: <@@@ @@@@>': license$
20  PRINT 'License: '; license$
30  END

RNH
License #: INF 5783
License: INF5783
```

.

A **decimal point** is used to align the fractional portion of a floating point number. If the screen format is in the form <###.####>, when the user presses the [.] key, the digits will be right-justified to the decimal point. When the field is complete, the digits to the right of the decimal will be left-zero filled.

```
10  PRINT 'Input .59, please'
20  INPUT SCREEN 'Amount: <###.####>': amt$
    PRINT 'Amount = '; amt$
30  END

RNH
Input .59, please
Amount = .5900
```

^

The ^ is used to specify an UPPER-CASE letter. If the next character input is a lower-case letter, INTOUCH will change it to upper-case and echo it to the terminal in upper-case.

```
10  INPUT SCREEN 'First name: <^@@@@@@@@@@@@@@>': first$
    INPUT SCREEN 'Middle initial: <^>.': middle$
    INPUT SCREEN 'Last name: <^@@@@@@@@@@@@@@@>': last$
20  PRINT first$; ' '; middle$; '. '; last$
30  END

RNH
First name: John__________
Middle initial: B.
Last name: Smithy__________
John B. Smithy
```

~

The ~ (tilde) is used to specify that any character that follows the ~ will not be replaced.

5.16.2 Screen Format Options

If you use format options, they must be placed *before* the format characters and followed by a colon. The following format options are available:

UCASE

Upper-cases all letters that are typed. If you type a letter in lower-case, it is changed to upper-case and echoed to the terminal screen in upper-case.

```
10  PRINT 'Type in some text!'
20  INPUT SCREEN '<UCASE:@@@@@@@@@@@@@@@@@@@@>': text$
30  PRINT 'Here is your text: '; text$
40  END

RNH
Type in some text!
JOHN B. SMITHY______
Here is your text: JOHN B. SMITHY
```

LCASE

Lower-cases all letters that are typed. If you type a letter in upper-case, it is changed to lower-case and echoed to the terminal screen in lower-case.

```
10  PRINT 'Type in some text!'
20  INPUT SCREEN '<LCASE: @@@@@@@@@@@@@@@@@@@@>': text$
30  PRINT 'Here is your text: '; text$
40  END

RNH
Type in some text!
john b. smithy
Here is your text: john b. smithy
```

NOECHO

Typed characters are not echoed (printed) on the screen.

```
10  INPUT SCREEN '<NOECHO:Password? @@@@@>' : pw$
20  PRINT pw$
30  END

RNH
Password? _____
Elene
```

DIGITS

Allows only digits to be entered in the field. The format character # also allows the minus sign and the decimal point.

```
10  INPUT SCREEN 'Phone: <DIGITS:###-####>': phone
20  PRINT phone
30  END

RNH
Phone: 555-6798
 5556798
```

AJ

AJ causes INTOUCH to jump automatically to the next field when the current field has been filled in. The field must be completely filled in before INTOUCH will jump to the next field.

```
10  INPUT SCREEN 'Phone: <AJ, DIGITS:###-####>': phone
20  INPUT SCREEN 'Soc. sec.: <DIGITS:###-##-####>': ssn
30  PRINT phone, ssn
40  END

RNH
Phone: 555-3839
Soc. sec.: 899-75-3432
 5553839            899753432
```

REQ

REQ specifies that input is required and you must enter something. The computer will beep and you will be prompted until valid input is entered.

AT row, column

The **AT** option displays the field on the screen at the row and column position specified. *row* specifies the row to print at. *column* specifies the column to print at.

You can supply defaults for each field in the screen. The format of the defaults is a list of values separated with line feeds. Defaults are set up in the following example.

```
10  city$ = 'St Paul'
    state$ = 'Minnesota'
    df$ = city$ + chr$(10) + state$
    CLEAR
    scr$ = '<AT 6, 10: City @@@@@@@@@@@@@@@@@@@@@@@@@@@@@>' &
          +'<AT 7, 10: State  ^^>'
    INPUT SCREEN scr$, DEFAULT df$: city$, state$
20  END

RNH
City     St. Paul
State    Mi
```

VALID

The **VALID** option allows full validation of each field on the screen. (See the VALID function in Appendix A for a list of validation rules.) The format of the VALID option is:

```
VALID 'rule_str'
```

rule_str is the list of validation rules. Multiple validation rules are separated by semi-colons.

```
10  INPUT SCREEN 'code = <VALID "integer":###-##-##>': ans$
    PRINT ans$
20  END

RNH
code = 121-23-47
1212347
```

ELAPSED

The **ELAPSED** option keeps track of the time it takes the user to respond to an INPUT prompt. INTOUCH assigns the elapsed time to the variable specified. The ELAPSED option starts marking time when INTOUCH finishes displaying the prompt. Time stops when all the variables listed in the INPUT statement have values. The format is:

```
ELAPSED num_var
```

num_var is the numeric variable the elapsed time is assigned to. INTOUCH returns the elapsed time in seconds. If a real numeric variable is specified, INTOUCH returns the time to a hundredth of a second (1/100).

```
10  INPUT 'Name', ELAPSED x: name$
20  PRINT name$; ', your elapsed time is'; x
30  END

RNH
Name? John
John, your elapsed time is 1.39
```

If an integer variable is specified, INTOUCH rounds the time to the nearest second.

```
10  INPUT 'Name', ELAPSED x%: name$
20  PRINT name$; ', your elapsed time is'; x%
30  END

RNH
Name? Julie
Julie, your elapsed time is 1
```

BOLD, BLINK, UNDERLINE, REVERSE

The **BOLD**, **BLINK**, **UNDERLINE** and **REVERSE** options allow each input string to be displayed with its own attributes. If these format options are used together with the ATTRIBUTES option, the ATTRIBUTES option will be supressed.

```
10  INPUT SCREEN '<AT 5,10, BOLD,BLINK:@@@@@@>' + &
                 '<AT 6,10, REVERSE:@@@@@@>': str1$, str2$
20  END
```

5.16.3 The User's Response to the SCREEN Option

When a single INPUT SCREEN statement is used to create a fill-in screen, certain keys on the keyboard have special functions. The following keys can be used to complete the screen:

TAB

Jump to the next field in the input screen statement. Once you have finished filling in a field, you must press the Tab key to jump to the next field.

\

The \ (backslash) can be used to return to a previous field. When the \ key is pressed, you will jump back to the beginning of the previous field and can change the entry.

Return

The Return key is used to complete a data entry screen. If more than one format is included in a screen option, when you have finished filling in all the fields, the Return key must be pressed to complete data entry. You must be at the last field when the Return key is pressed. Until the Return key is pressed, you are free to change your response to any field.

```
10  a$ = '<AT 6, 10: Full name @@@@@@@@@@@@@@@@@@@@@@@@@>' + &
         '<AT 8, 10: Address @@@@@@@@@@@@@@@@@@@@@@@@@@@>' + &
         '<AT 10, 10: City @@@@@@@@@@@>' + &
         '<AT 10, 38, AJ, REQ, LETTERS: State ^^>' + &
         '<AT 10, 50, REQ, DIGITS: Zip #####>'
20  CLEAR
    INPUT SCREEN a$: name$, addr1$, city$, state$, zip$
    DELAY
30  CLEAR
    PRINT AT 6, 10: name$
    PRINT AT 8, 10: addr1$
    PRINT AT 10, 10: city$; ', '; state$; '   '; zip$
40  END

RNH

     Full name Sunny Day_________________

     Address 2356 Main St._______________

     City San Diego___           State CA    Zip 92131

                  Press the RETURN key to continue
```

```
Sunny Day
2356 Main St.
San Diego, CA   92131
```

5.17 MENU Option

Use the **MENU** option to create pop-up menus.

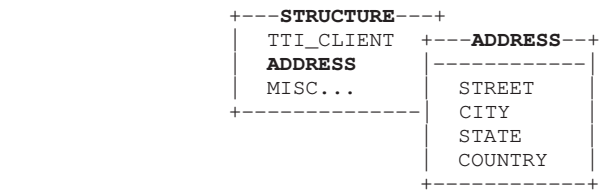
FORMAT:

```
[LINE] INPUT MENU str_expr: str_var
```

str_expr is a string expression that describes the menu. It consists of multiple, comma-separated elements. The elements can be either items or directives.

EXAMPLE:

```
10  title$ = '%TITLE "structure",%MULTI,'
    box_loc$ = '%AT 10,15,'
    client$ = '"TTI_Client"={id,name={Cass,Brock,Errant},phone},'
    address$ = 'address={%BAR,street,city,state,country},'
    misc$ = 'misc={%REPLACE,mail,record}'
    menu$ = title$ & box_loc$ & client$ & address$ & misc$
20  LINE INPUT MENU menu$: ans$
30  END

RNH
```



5.17.1 Items

Menu items consist of either a description or a description and = sign followed by either a data value or a submenu indicator. If no = sign is given, the data value is the same as the description. If the description contains spaces or special characters, it must be enclosed within quotes.

5.17.1.1 Descriptions

Menu descriptions are displayed on the menu for the user to choose.

5.17.1.2 Data Value

The data value is the value returned to the application when the menu item is chosen.

5.17.2 Submenu Indicator

A submenu indicator consists of a "{" followed by menu elements followed by "}".

5.17.3 Menu DIRECTIVES:

The following directives can be used with the [LINE] INPUT MENU statement:

%AT row, column

The menu or submenu box can be placed at a given row and column. **CENTER** can be used in place of row and/or column to center a menu or submenu.

```
%AT row, CENTER
%AT CENTER, column
%AT CENTER, CENTER
```

%AUTODISPLAY

The **%AUTODISPLAY** directive controls display of submenus. Unless specifically turned OFF, INTOUCH automatically displays submenus.

FORMAT:

```
%AUTODISPLAY [ON|OFF]
```

%AUTOVBAR

The **%AUTOVBAR** directive is used to turn vertical bars ON and OFF in menus. By default, menubars do not have vertical bars separating the items. The **%AUTOVBAR** directive can be used to enable this feature, but it must be placed after the %MENUBAR directive.

FORMAT:

```
%AUTOVBAR [ON|OFF]
```

EXAMPLE:

```
10 item1$='%MENUBAR,%AUTOVBAR on,OPEN,SELECT={INCLUDE,EXCLUDE},SORT,PRINT'
   LINE INPUT MENU item1$ : ans$
   END

RNH

+-------------------------------------------------------------------------------+
|  OPEN   |   SELECT   |   SORT   |   PRINT                                      |
+-----------+---SELECT---+-----------------------------------------------------+
            |  INCLUDE   |
            |  EXCLUDE   |
            +------------+
```

%BAR

Separates DESCRIPTIVE-items with "—-". The separated line consists of the characters specified in a "text_str". If no text is given, the separated line defaults to a dashed line.

FORMAT:

```
%BAR ['text_str']
```

EXAMPLE:

```
10 menu$ = '"DATA ENTRY","REPORTS",%BAR"*",MAIL,EXIT'
   LINE INPUT MENU menu$ : ans$
20 END

RNH

+---------------+
|  DATA ENTRY   |
|  REPORTS      |
|  **********   |
|  MAIL         |
|  EXIT         |
+---------------+
```

%COLUMNS

The **%COLUMNS** directive sets the number of columns for the menu.

FORMAT:

```
%COLUMNS number
```

EXAMPLE:

```
10 menu$ = '%COLUMNS 3,"DATA ENTRY","REPORTS",MAIL,HELP,EXIT'
   LINE INPUT MENU menu$ : ans$
20 END

RNH
```

%HEADING

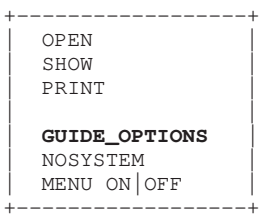
The **%HEADING** directive displays a blank line and a "text_str" between menu items. If no "text_str" is given, the line defaults to a dashed line.

FORMAT:

```
%HEADING ['text_str']
```

EXAMPLE:

```
10 menu$ ='OPEN,SHOW,PRINT,%HEADING"GUIDE_OPTIONS",NOSYSTEM,"MENU ON|OFF"'
   LINE INPUT MENU menu$ : ans$
20 END

RNH
```



%ITEMS

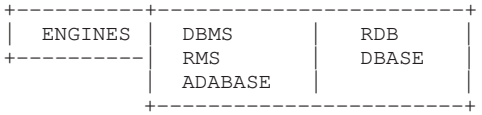
The **%ITEMS** directive creates a multi-column menu. "number" represents the number of items per column. **%ITEMS** creates as many columns as is necessary. Horizontal scroll the columns as needed.

FORMAT:

```
%ITEMS number
```

EXAMPLE:

```
10  menu$='ENGINES={%ITEMS 3, DBMS, RMS, ADABASE, RDB, DBASE}'
    INPUT MENU menu$: ans$
20  END

RNH
```



%LOCKSTEP

The **%LOCKSTEP** directive controls column scrolling when there are multiple columns in one menu. If it is turned OFF, the columns scroll independently. Otherwise, the columns scroll together.

FORMAT:

```
%LOCKSTEP [ON|OFF]
```

EXAMPLE:

```
10 menu$ = '%LOCKSTEP off, %SIZE 4, 1,2,3,4,5,6,%SPLIT,7,8,9,10,11,12'
   LINE INPUT MENU menu$ : ans$
   PRINT 'Please, PRESS the "NEXT" key'
20 END

RNH

Please, PRESS the 'NEXT' key

+----------------+
|  ...  |   7    |
|  4    |   8    |
|  5    |   9    |
|  6    |   ...  |
+----------------+
```

%MAXWIDTH

The **%MAXWIDTH** directive controls the maximum width of the entire menu.

FORMAT:

```
%MAXWIDTH number
```

EXAMPLE:

```
10  menu$ = '%MAXWIDTH 20,%ITEMS 2,a,b,c,d,f,g,h,i,j'
    INPUT MENU menu$ : ans$
20  END

RNH

+--------------------+
|<<  F    |   H     >>|
|    G    |   I       |
+--------------------+
```

%MENUBAR

The **%MENUBAR** directive creates menu bars (menus with choices listed horizontally) with pull-down submenus.

FORMAT:

```
%MENUBAR
```

EXAMPLE:

```
10 item1$='%MENUBAR,OPEN,SELECT={INCLUDE,EXCLUDE},SORT,PRINT'
   LINE INPUT MENU item1$ : ans$
20 END

RNH
```

```
+-------------------------------------------------------------------------------+
|  OPEN      SELECT      SORT      PRINT                                        |
+----------+---SELECT---+-----------------------------------------------------+
           |  INCLUDE   |
           |  EXCLUDE   |
           +------------+
```

%MESSAGE

The **%MESSAGE** directive displays a message when the menu or submenu is displayed.

FORMAT:

```
%MESSAGE 'message'
```

EXAMPLE:

```
10  CLEAR
20  menu$ = 'open,show,print, %MESSAGE "Select a menu option"'
    LINE INPUT MENU menu$: ans$
30  END

RNH
```

```
+----------+
|  OPEN    |
|  SHOW    |
|  PRINT   |
+----------+
```

```
Select a menu option
```

%MULTI

The **%MULTI** directive allows multiple items to be selected from a menu or submenu.

%REPLACE

The **%REPLACE** directive can be located in any submenu. The calling menu is not kept on the screen. Pressing '\' will return to the calling menu.

%SIZE number

Determines the number of DESCRIPTIVE-items that are located in the menu_box.

There is no limit to the number of the items that can be used. If all of the items do not fit within the menu box, the items are vertically scrolled using the UP, DOWN, NEXT and PREV keys. "Str_var" contains the name of the selected item. If the %MULTI directive is used and the Select key is pressed, "Str_var" contains the names of the items. Each item is separated by a linefeed character.

%SPLIT

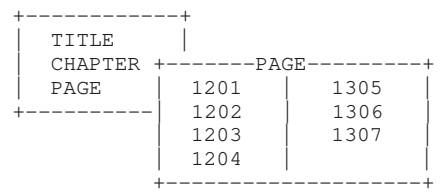
The **%SPLIT** directive instructs INTOUCH to start a new column at that specified point in the menu.

FORMAT:

```
%SPLIT
```

EXAMPLE:

```
10  item1$='title,chapter,page={1201,1202,1203,1204,%SPLIT,1305,1306,1307}'
    LINE INPUT MENU item1$ : ans$
20  END

RNH
```



%TITLE "....."

Assign a title to a menu or submenu.

%VBAR

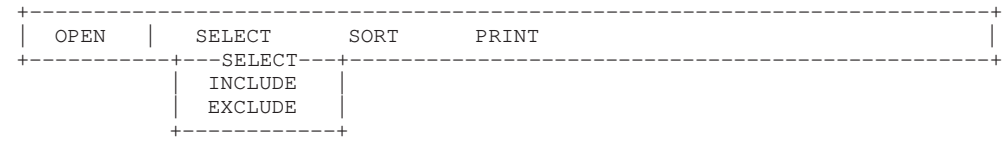
The **%VBAR** directive is used to explicitly insert a vertical bar.

FORMAT:

```
%VBAR
```

EXAMPLE:

```
10 item1$='%MENUBAR,OPEN,%VBAR,SELECT={INCLUDE,EXCLUDE},SORT,PRINT'
   LINE INPUT MENU item1$ : ans$
   END

RNH
```



%WIDTH

The **%WIDTH** directive controls the minimum width of the current column.

FORMAT:

```
%WIDTH number
```

EXAMPLE:

```
10  menu$ = '%WIDTH 20,%ITEMS 2,a,b,c,d,f,g,h,i,j'
    INPUT MENU menu$ : ans$
20  END

RNH
```

5.17.4 User Interface

The following is the user interface in a [LINE] INPUT MENU:

- UP and DOWN arrow keys move the menu cursor from one item to another.
- Backslash (\) or 'LEFT arrow' key return to the calling menu.
- CTRL/Z - exit to INTOUCH prompt.
- 'SPACE', 'RETURN', 'RIGHT arrow' selects the item , if the item is a submenu.
- The 'SPACE' or 'SEL' keys should be used with '%MULTI', select an item.
- The 'REM' key should be used with '%MULTI' to remove the item, or press 'SPACE BAR' a second time.
- The 'DEL' key moves the menu cursor back to the first item in the menu.

5.18 KEY INPUT

FORMAT:

```
KEY INPUT [[#channel] [, PROMPT str_expr]
     [, TIMEOUT time_limit]
     [, ELAPSED num_var]
     :] var, [var,...]
```

EXAMPLE:

```
10  PRINT 'See how quick you are.'
20  KEY INPUT PROMPT 'Press a key, quick!', &
          ELAPSED x: press$
30  PRINT
    PRINT 'You took'; x; 'seconds to Press '; press$; '.'
40  END

RNH
See how quick you are.
Press a key, quick!
You took 1.99 seconds to Press h.
```

PURPOSE:

KEY INPUT inputs a keystroke from the user and stores the value of the key in the string variable specified.

DESCRIPTION:

Some applications require the use of special keys on the terminal or keystroke level validation. The KEY INPUT statement is used for these applications.

KEY INPUT does not echo the key pressed, nor does it generate a line feed.

All the options available with the preceding "INPUT" statement are also available with KEY INPUT. KEY INPUT returns the following:

_BACK
_EXIT
_HELP
_TERMINATOR

_TERMINATOR returns control (Ctrl) characters in the format "CTRL/X". For example, if the user presses Ctrl/G, _TERMINATOR returns "CTRL/G".

```
10  KEY INPUT 'Press a terminator' : z$
    PRINT
    PRINT 'You pressed '; _TERMINATOR
20  END

RNH
Press a terminator?            (User presses CTRL/G)
You pressed CTRL/G
```

6 Printing and Displaying Data

6.1 PRINT

The **PRINT** statement prints or displays text on the screen. The printed text can be formatted using a mask or directive and/or highlighted using video options. This section describes the various ways that text can be displayed on the screen.

FORMAT:

```
PRINT [[AT row, col] [,ERASE] [,WIDE] [,BLINK] [,UNDERLINE] [,REVERSE]
[,BOLD] [,USING "print_mask"]:] expr [{, | ;} expr...] [, | ;]
```

EXAMPLE:

```
10  INPUT name$
    PRINT 'Hello, '; name$
    PRINT BOLD, UNDERLINE: 'Here is a number: 1.93'
20  END

RNH
? Rick
Hello, Rick
Here is a number: 1.93
```

DESCRIPTION:

The simplest version of the PRINT statement is:

```
PRINT expr
```

expr is an expression to print. *expr* can be any INTOUCH expression. INTOUCH prints the value of the expression at the current cursor position and then generates a new line. A PRINT statement without an expression simply generates a new line.

The **?** can be used as a synonym for PRINT at the command prompt.

```
10  PRINT 'Line 1'
    PRINT
    PRINT 'Line 3'
20  END

RNH
Line 1

Line 3
```

6.1.1 Printing Multiple Expressions

One PRINT statement can print several items. Multiple items must be separated with a *comma* or a *semicolon*. The separator determines where the **next** expression will be printed.

You can use two additional features to position the cursor:

- The *TAB* function positions the cursor at the column specified.
- The *AT* option positions the cursor at the coordinates specified. It can only be used once, at the beginning of the PRINT statement.

Semicolons

Separating print items with a **semicolon** causes the items to immediately follow one another. When the items are printed, no spaces appear between the expressions.

```
10  alpha$ = 'ABCDEFGHIJKLM'
    bet$ = 'NOPQRSTUVWXYZ'
    PRINT alpha$; bet$
20  END

RNH
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Commas and Print Zones

You can print in columns by using **print zones.** Each print zone has a default width of twenty characters. To change the width, see Section 9.17.2, SET ZONEWIDTH.

```
|-------------------|-------------------|-------------------|
1                  20                  40                  60
```

Separating items with a comma causes the item **following** the comma to be printed in a new print zone. The terminal width determines the number of zones in each line. (See Section 9.8.1, ASK MARGIN statement to determine the terminal width.)

```
10  INPUT name_1$, name_2$
20  PRINT name_1$, 'MARY', name_2$
30  END

RNH
? FRED, JOHN
FRED                MARY                JOHN
```

If an item is longer than the zone width, INTOUCH continues it into the next print zone. INTOUCH uses as many print zones as necessary to print an item.

```
RNH
? FRED, DILLENSCHNEIDER & SONS
FRED                MARY                DILLENSCHNEIDER & SONS
```

INTOUCH writes data sequentially. If an item is too long (over 132 characters) to write in one record, INTOUCH continues it in the next record.

```
10  OPEN #1: NAME 'test.tmp', ACCESS OUTPUT
    PRINT #1: REPEAT$('+-', 100)
    CLOSE #1
20  OPEN #1: NAME 'test.tmp', ACCESS INPUT
    LINE INPUT #1: record_1$, record_2$
    PRINT 'Record 1: '; record_1$
    PRINT 'Record 2: '; record_2$
    CLOSE #1
30  END
```

```
RNH
Record 1:
+-+-+-
```

6.1.3 AT Option

The **AT** option moves the cursor to a specificed row and column on the screen.

FORMAT:

```
AT row, column
```

row is the row to print at. *column* is the column to print at. Row and column can be any integer numeric constants.

```
10  INPUT name_1$, name_2$
20  CLEAR
30  PRINT AT 3,10: name_1$; TAB(20); 'Mary'; TAB(35); name_2$
40  END
```

```
RNH
? Fred, John

         FRED      MARY           JOHN
```

6.1.4 ERASE Option

The **ERASE** option erases from the end of the printed text to the end of the line.

```
10  CLEAR
20  PRINT AT 10, 1: 'Mary had a little lamb'
    DELAY 2
30  PRINT AT 10, 1: 'Jeff'
    DELAY 2
40  PRINT ERASE, AT 10, 1: 'Caroline'
    DELAY 2
50  END
```

```
RNH
Mary had a little lamb
.
.
.
Jeff had a little lamb
.
.
.
Caroline
```

6.1.5 WIDE Option

This option prints *double-width* characters on the terminal screen.

```
10  PRINT WIDE: 'INTOUCH'
20  END
```

6.1.6 Printing Numbers

The following rules govern the printing of numbers:

- Numbers are always followed by a space, even if the separator is a semicolon. Positive numbers and zero are also preceded by a space.

```
10  LET x = 7
    LET y = 10
    PRINT x; y
20  END

RNH
 7  10
```

- Negative numbers are preceded by a minus sign.

```
10  LET x = -7
    LET y = -10
    PRINT x; y
20  END

RNH
-7 -10
```

- If a number can be represented as an integer of 12 or fewer digits, it is printed as such.

```
10  LET x = 700000000001
    PRINT x
20  END

RNH
 700000000001
```

- If a number can be rounded to a decimal number twelve digits or less, it is printed as such.

```
10  LET x = 700000000001.89
    PRINT x
20  END

RNH
 700000000002
```

- If a number can't be printed in one of the above ways, it is printed in standard scientific *E notation*. (See E Notation in Section 3.2.4, Constants.)

```
10  LET X = 70000000000001.89
    PRINT X
20  END

RNH
 .7E+14
```

INTOUCH prints:

- 10 digits of precision for integer numbers.
- 12 digits of precision for real numbers.

6.1.7 Positioning the Cursor for the Next Print Statement

If a separator (comma, semicolon, TAB function or AT option) is the last item in the PRINT statement, INTOUCH advances the cursor to the position specified and **does not** generate a new line.

```
10  PRINT TAB(5); 7,
    PRINT 10;
    PRINT 20
20  END

RNH
      7             10  20
```

6.1.8 Printing Attributes–Highlighting Options

The video attribute options highlight text on the screen. Separate the options from the print list with a colon. The options are:

BLINK	causes the expressions in the print list to blink in low and high intensity
UNDERLINE	causes the print list to be underlined
BOLD	causes the expressions in the print list to appear in bold (high intensity)
REVERSE	causes the print list to appear in reverse video

The video options can be combined. For example, the following program will print "Hello" boldfaced and in reverse video. It will then blink and underline the word "Goodbye".

```
10  PRINT BOLD, REVERSE: 'Hello'
20  PRINT UNDERLINE, BLINK: 'Goodbye'
30  END
```

6.1.9 USING Option

The **USING** option is used to format text. The print mask indicates the format for the data in the print list.

FORMAT:

```
USING print_mask
```

The *print_mask* consists of **fields** or **directives** and text. The text can precede or follow a field or directive. The print mask tells how to print the expressions in the print list.

```
               fields
                /   \
10  LET a$  = "#.## ##.##"
20  PRINT USING "amount = #.##": 1.9
30  PRINT USING a$: 1.93, -1.93
40  END

RNH
amount = 1.90
1.93 -1.93
```

The USING Print List

Expressions in the print list with the USING option are separated by commas. A trailing semicolon is allowed. The expressions are printed according to the the format. However, a trailing semicolon causes INTOUCH to leave the cursor at the end of the print line.

```
10  PRINT USING "###.## ###.##": 22.88, 45;
20  PRINT " and others."
30  END

RNH
 22.88  45.00 and others.
```

6.1.10 Fields

Fields are made up of format characters. The format characters tell INTOUCH how to print the expressions in the print list.

6.1.10.1 String Format Characters

character

The # is used to indicate a character position—a place in the format where a character can occur. For example:

```
10  PRINT USING '#### ##': 'Test', 'Hi'
20  END

RNH
Test Hi
```

In the above example, there are two fields. When the first string is printed, the word "Test" occupies all four character positions. When the second expression is printed (Hi), only two character positions are used.

If the string expression being printed is smaller than the field, the expression will be printed **centered** within the field.

```
10  PRINT USING '#### ####': 'Test', 'Hi'
15  PRINT '123456789'
20  END

RNH
Test  Hi
123456789
```

If the string expression is longer than the field, INTOUCH generates an exception.

6.1.10.2 Numeric Format Characters

character

The # can also be used to specify digits. Each # represents one numeric digit position.

```
10  PRINT USING "##": 19
20  END

RNH
19
```

If you indicate more positions than the numeric expression contains, the expression will be right-justified and padded with spaces.

```
10  PRINT '1st 2nd 3rd'
20  PRINT USING "### ### ###": 193, 19, 1
30  END

RNH
1st 2nd 3rd
193  19   1
```

INTOUCH prints a minus sign in front of negative numbers. INTOUCH does not print a sign in front of positive numbers.

```
10  PRINT '1st 2nd 3rd'
20  PRINT USING "### ### ###": 193, 19, -1
30  END

RNH
1st 2nd 3rd
193  19  -1
```

If you indicate more positions to the left of the decimal point than the expression contains, the expression will be printed with leading spaces.

```
10  PRINT USING "###.##": 1.9
20  END

RNH
  1.90
```

If you indicate more positions to the right of the decimal point than the expression contains, the expression will be printed with trailing zeros.

```
10  PRINT '--1-- --2--'
20  PRINT USING "##.## ##.##": 1.3, 1.25
30  END

RNH
--1-- --2--
 1.30  1.25
```

< character

The less than sign left-justifies text within a field. The less than sign must appear at the beginning of a field. The less than sign counts as a character position. In this example, justification occurs only in the second field.

```
10  PRINT USING "#### <###": 'Test', 'Hi'
20  PRINT '123456789'
30  END

RNH
Test Hi
123456789
```

In the above example, there are two fields. When the first string is printed, the word "Test" occupies all four character positions. The less than sign (<) causes INTOUCH to left-justify the second expression.

> character

The greater than sign is used to right-justify text within a field. The greater than sign must appear at the beginning of a field. The greater than sign counts as a character position.

```
10  PRINT USING "#### >###": 'Test', 'Hi'
20  PRINT '123456789'
30  END

RNH
Test   Hi
123456789
```

In the above example, there are two fields. The greater than sign (>) causes INTOUCH to right-justify the second expression.

@ character

The @ indicates one character position with no translation.

```
10  PRINT USING '####': 0001
20  PRINT USING '@@@@': 0001
30  END

RNH
   1
0001
```

. character

You can include a decimal point in a number by putting a period or decimal point in the format.

```
10  PRINT USING "###.##": 19.3
20  END

RNH
 19.30
```

, character

You can include commas in your numbers by putting commas in the format.

```
10  a$ = "##,###.##"
20  PRINT USING a$: 28290.06
    PRINT USING A$: 8290.06
    PRINT USING A$: 290.06
30  END

RNH
28,290.06
 8,290.06
   290.06
```

Commas cannot be used in exponential format.

% character

The **%** character pads on the left with zeros.

```
10  PRINT '-1- -2- -3-'
    PRINT USING "%%% %%% %%%": 193, 19, 1
20  END

RNH
-1- -2- -3-
193 019 001
```

* character

The * character pads on the left with asterisks. You can use this symbol for setting up check amounts.

```
10  PRINT USING '***,***.**': 19.42
20  END

RNH
*****19.42
```

If the expression is smaller than the format, INTOUCH will right justify the expression and pad it with **asterisks**.

```
10  PRINT '-1- -2- -3-'
    PRINT USING "*** *** ***": 193, 19, 1
20  END

RNH
-1- -2- -3-
193 *19 **1
```

+ character

A plus sign causes INTOUCH to print a leading plus or minus sign. INTOUCH will print a plus sign in front of positive numbers and a minus sign in front of negative numbers.

The "+" sign adds a character position to the format. The character position is used for the sign of the number.

```
10  PRINT ' -1-  -2-  -3-'
    PRINT USING "+### +### +###": 193, 19, -1
20  END

RNH
 -1-  -2-  -3-
+193  +19   -1
```

- character

Prints a leading or trailing minus sign for negative numbers, and a leading space for positive numbers. The "-" adds a character position to the format. The character position is used to print the minus sign or space.

```
10  PRINT ’ -1-  -2-  -3-’
    PRINT USING "-### -### -###": 193, 19, -1
20 END

RNH
 -1-  -2-  -3-
 193   19   -1
```

$ character

Prints a floating dollar sign. The dollar sign appears before the number. $ causes INTOUCH to print ’$-’ for negative numbers and ’$’ for positive numbers. The minus sign appears immediately **after** the dollar sign and **before** the number.

```
10  PRINT "1st col 2nd col"
    PRINT USING "$###.## $###.##": 11.93, -1.93
20  END

RNH
1st col 2nd col
 $11.93  $-1.93
```

$+ characters

Prints a floating dollar sign. The dollar sign appears before the numeric expression. $+ causes INTOUCH to print a minus sign before negative numbers, and a plus sign before positive numbers.

The sign appears after the dollar sign and before the number.

```
10  PRINT "1st  col 2nd  col"
    PRINT USING "$+###.## $+###.##": 11.93, -1.93
20 END

RNH
1st  col 2nd  col
 $+11.93   $-1.93
```

-$ characters

Prints a floating dollar sign. The dollar sign appears immediately before the numeric expression. -$ causes INTOUCH to print a minus sign before negative numbers and a **space** before positive numbers. The minus sign or space appears immediately before the dollar sign.

```
10  PRINT "1st  col 2nd  col"
    PRINT USING "-$###.## -$###.##": 11.93, -1.93
20  END

RNH
1st  col 2nd  col
  $11.93   -$1.93
```

+$ characters

+$ causes INTOUCH to print a floating dollar sign. The dollar sign appears immediately before the number. +$ causes INTOUCH to print a plus sign before positive numbers and a minus sign before negative numbers. The plus or minus sign appears immediately before the dollar sign.

```
10  PRINT "1st  col 2nd  col"
    PRINT USING "+$###.## +$###.##": 11.93, -1.93
20  END

RNH
1st  col 2nd  col
 +$11.93   -$1.93
```

Notice that +$ adds two character positions to the format. One position contains the dollar sign, the other contains the plus or minus sign.

$- characters

Prints a floating dollar sign. The dollar sign appears before the number. $- causes INTOUCH to print a minus sign before negative numbers and a **space** before positive numbers. The minus sign or space appears after the dollar sign and before the number.

```
10  PRINT "1st  col 2nd  col"
    PRINT USING "$-###.## $-###.##": 11.93, -1.93
20  END

RNH
1st  col 2nd  col
 $ 11.93   $-1.93
```

If your expression is too large to fit in a field, INTOUCH gives an exception.

6.1.11 Directives

The **directives** used with the USING option of the PRINT statement tell INTOUCH what to do with the text.

FORMAT:

```
PRINT USING 'directive' : str_expr
```

6.1.11.1 {UCASE}?

The **UCASE** directive converts the str_expr to upper-case characters.

```
10  PRINT USING '{UCASE}?' : 'march'
20  END

RNH
MARCH
```

6.1.11.2 {LCASE}?

The **LCASE** directive converts the str_expr to lower-case characters.

```
10  PRINT USING '{LCASE}?' : 'MARCH'
20  END

RNH
march
```

6.1.11.3 {HYPHEN}

The **HYPHEN** directive causes INTOUCH to suppress the hyphen character if it is the last non-blank character after the format is applied.

```
10  PRINT USING '<#####~-####' : '92123'
    PRINT USING '{HYPHEN}<#####~-####' : '92123'
20  END

RNH
92123 -
92123
```

6.1.11.4 {DATE [argument]}?

Given a str_expr that contains a date in the format YYMMDD or CCYYMMDD, the **DATE** directive converts the str_expr to a default or specified, optionally-masked, date format.

These date arguments can be used: YMD, CYMD, MDY, MDCY, DMY, DMCY, DMONY, DMONCY, MONTHDY, MONTHDCY. If no argument is provided, the default is MDCY. (See FORMAT$(expr, str_expr) for examples of date argument usage.)

To format the resulting date, replace the **?** with a print mask.

```
10  PRINT USING '{DATE}?': '950401'
    PRINT USING '{DATE DMY}?': '950401'
    PRINT USING '{DATE DMCY}?': '950401'
    PRINT
    PRINT USING '{DATE MDY}?': '19950215'
    PRINT USING '{DATE MDY}##/##/##': '19950215'
    PRINT USING '{DATE MDCY}##/##/####': '19950215'
20  END

RNH
04011995
010495
01041995
```

```
021595
02/15/95
02/15/1995
```

6.1.11.5 {ROTATE n}?

The **ROTATE** directive rotates the last n characters in a str_expr to the first position in the str_expr. Optionally, the resulting str_expr can be masked by replacing the **?** with a print mask.

```
10  PRINT USING '{ROTATE 3}?': '5552527800'
    PRINT USING '{ROTATE 3}###~ ###~-####': '5552527800'
    PRINT
    PRINT USING '{ROTATE 5}?': 'TuneTommy'
    PRINT USING '{ROTATE 5}#####~ ####': 'TuneTommy'
20  END
```

```
RNH
8005552527
800 555-2527

TommyTune
Tommy Tune
```

6.1.11.6 {TIME}?

Given a str_expr containing a 4 digit time in HHMM or HH:MM format or a 6 digit time in HHMMSS or HH:MM:SS format, the **TIME** directive converts the str_expr to HH:MM AM/PM or HH:MM:SS AM/PM.

```
10  PRINT USING '{TIME}?': '1022'
    PRINT USING '{TIME}?': '19:45'
    PRINT
    PRINT USING '{TIME}?': '102255'
    PRINT USING '{TIME}?': '19:45:36'
20  END
```

```
RNH
10:22 AM
07:45 PM

10:22:55 AM
07:45:36 PM
```

6.1.11.7 {ZIPCODE}?

Given a str_expr containing a 5, 6 or 9 digit zipcode, the **ZIPCODE** directive converts the str_expr to an appropriate zipcode format.

```
10  PRINT '5 character zipcode : ';
    PRINT USING '{ZIPCODE}?': '92126'
    PRINT '6 character zipcode : ';
    PRINT USING '{ZIPCODE}?': 'K8A3P9'
    PRINT '9 character zipcode : ';
    PRINT USING '{ZIPCODE}?': '931327845'
20  END

RNH
5 character zipcode : 92126
6 character zipcode : K8A 3P9
9 character zipcode : 93132-7845
```

6.2 Using the INTOUCH Screen

INTOUCH has several features that allow you to control the appearance of the screen. The **FRAME ON** and **FRAME OFF** commands discussed previously turn the INTOUCH frame on and off. The statements described in the following sections can be used, within your program, to manipulate the screen.

6.3 MESSAGE

The **MESSAGE** statement prints a message at line 23 (the default line) on the screen. When the INTOUCH frame is on, line 23 is within the lower frame.

FORMAT:

```
MESSAGE [ERROR: | DELAY:] expr [; | , expr]
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:
20  DO
      MESSAGE 'Enter EXIT to exit'
      INPUT 'Please enter your name': name$
      IF  _EXIT  THEN
        MESSAGE 'The End'
        EXIT DO
      ELSE
        PRINT name$
        REPEAT DO
      END IF
    END DO
30  END

RNH
Please enter your name?
                        Enter EXIT to exit       (first message)
```

```
Please enter your name? Tester
Tester
Please enter your name? exit
                              The End              (second message)
```

DESCRIPTION:

INTOUCH uses the message line (defaults to line 23) to display messages and errors. You can use the MESSAGE statement to display your own messages and errors on this line.

When the INTOUCH frame is on, your messages will appear within the lower frame. The MESSAGE statement can print several items. Each item can be any INTOUCH numeric or string expression. Multiple items must be separated with a comma or a semicolon. The separator determines where the *next* expression will be printed.

Semicolons

Separating message items with a semicolon causes the items to immediately follow one another. When the items are printed, no spaces appear between the items.

Commas

Separating items with a comma puts a space between each item.

INTOUCH would display this message:

```
MESSAGE 'number is', 123; 456; 789
```

as:

```
number is 123456789
```

ERROR Option

INTOUCH displays a message for at least three seconds before clearing the message. When the **ERROR** option is used, INTOUCH rings the device's bell, purges typeahead and displays the message. (Typeahead is the feature that accepts characters typed ahead of the computer's request for input.)

FORMAT:

```
MESSAGE ERROR: expr [; | , expr]
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:
    INPUT 'Enter your age': age$
    MESSAGE ERROR: 'Is this really your age?'
20  END

RNH
Enter your age? 99

                    Is this really your age?
```

DELAY Option

The **DELAY** option of the MESSAGE statement causes INTOUCH to set an automatic delay, giving the user time to view the message before clearing the message. Starting with a minimum delay of approximately three seconds, INTOUCH increases the delay a little for lengthier messages.

FORMAT:

```
MESSAGE DELAY: expr [; | , expr]
```

FORMAT:

```
10  CLEAR
    z$ = 'This is a very, very, very, very, very, very long message'
    MESSAGE DELAY: z$
    MESSAGE DELAY: 'Short message'
20  END

RNH

   This is a very, very, very, very, very, very long message

                        Short message
```

6.4 DELAY

FORMAT:

```
DELAY [num_expr]
```

EXAMPLE:

```
10  PRINT 'Waiting a bit'
20  DELAY 4.5
30  PRINT 'Done'
40  END

RNH
Waiting a bit
Done
```

PURPOSE:

Use **DELAY** when you need to cause a timed delay before continuing program execution—for instance, to give the user time to read a message before clearing the screen.

DESCRIPTION:

DELAY causes INTOUCH to pause for the specified number of seconds before continuing program execution. The numeric expression, *num_expr*, does not have to be a whole number, you can use fractions. For example:

```
DELAY 3.5
```

If *num_expr* is omitted, INTOUCH prints this message at the bottom of the screen:

```
Press the RETURN key to continue
```

and waits for the user to respond.

If, at the "Press RETURN..." prompt, a user enters:

Ctrl/Z	_EXIT is set to TRUE (1)
\ or UP-arrow	_BACK is set to TRUE (1)
Help	_HELP is set to TRUE (1)

6.5 CLEAR

The **CLEAR** statement can be used to clear the INTOUCH screen (everything within the INTOUCH frame) or to clear a specific area of the screen. CLEAR can be used to clear any rectangular area within the frame. When the INTOUCH frame is off, the entire screen can be manipulated with the CLEAR statement. CLEAR always removes messages, even if they are in the frame. You can use this statement to clear the screen before executing code or printing information on the screen.

FORMAT:

```
CLEAR [AREA [BOX] [, attr_list:] row_1, col_1, row_2, col_2]
```

EXAMPLE:

```
10  CLEAR
    INPUT 'Please enter your name': name$
    PRINT 'Hello, '; name$
20  END

RNH
Please enter your name? Tester
Hello, Tester
```

DESCRIPTION:

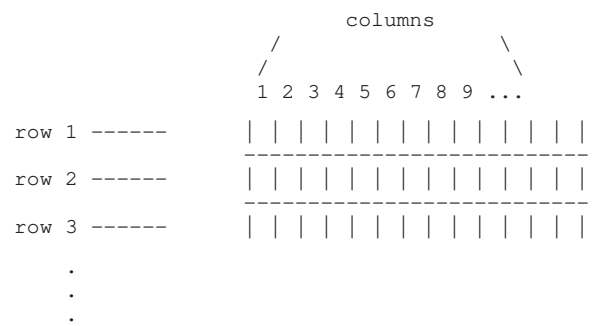
CLEAR, by itself, clears all text from the screen. CLEAR does not remove the INTOUCH frame if the frame is on, however, it does remove any message text that is displayed within the frame. If the frame is on, INTOUCH positions the prompt within the frame.

AREA Option

The **AREA** option clears a specific section of the screen. The cleared area is rectangular in shape.

row specifies a vertical position on the screen. Rows are numbered sequentially from the top of the screen to the bottom. With the INTOUCH frame on, there are typically 19 rows. With the frame off, there are usually 24 rows (The number of rows can vary depending on the terminal type and setting.)

col specifies a column–a horizontal position on the screen. Columns are numbered sequentially from the first character position on the left of the screen to the last character position on the right of the screen. Usually there are either 80 or 132 columns (depending on the terminal setting).



Two coordinates must be specified. These coordinates mark opposite corners of a rectangular area. INTOUCH clears the rectangular area specified by these two coordinates. For instance, the statement CLEAR AREA 2,3,8,20 would clear a rectangular area:

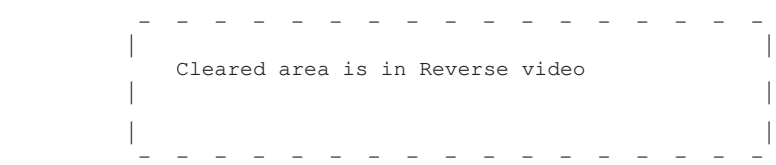
The first coordinate (2,3) indicates the upper-left corner. The second coordinate (8,20) indicates the lower-right corner.

FORMAT:

```
CLEAR AREA [, attr_list:] row_1, col_1, row_2, col_2
```

EXAMPLE:

```
10  CLEAR
    CLEAR AREA REVERSE: 5, 10, 11, 60
    PRINT AT 7, 20: 'Cleared area is in Reverse video'
20  END

RNH
```



CLEAR AREA allows the following attributes to be used when clearing an area: **BOLD**, **BLINK**, **REVERSE**, **UNDERLINE**. Multiple attributes used in one statement are separated by commas.

BOX Option

The **BOX** option creates an empty box with a frame.

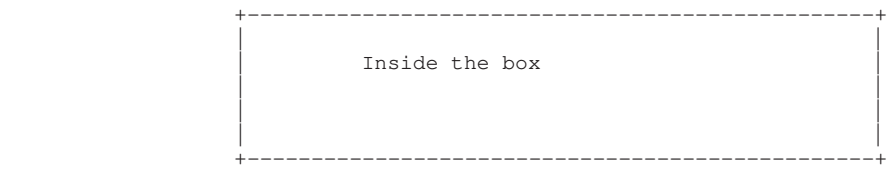
The **BOLD**, **BLINK**, **REVERSE** and **UNDERLINE** attributes can also be used with the BOX option. Separate attributes in one statement with commas.

FORMAT:

```
CLEAR AREA BOX [, attr_list:] row_1, col_1, row_2, col_2
```

EXAMPLE:

```
10  CLEAR
    CLEAR AREA BOX, BOLD: 5, 10, 11, 60
20  PRINT AT 7, 20: 'Inside the box'
    PRINT AT 12, 1: 'Outside the box'
30  END

RNH
```




```
Outside the box
```

6.6 OPTION COMMUNICATE ON | OFF

Some VMS facilities can broadcast messages to your terminal. INTOUCH will trap those messages and display them within the message area. Broadcast messages (communications) will stay on the screen until another message of any type is displayed.

FORMAT:

```
OPTION COMMUNICATE ON | OFF
```

EXAMPLE:

```
10  CLEAR
    OPTION COMMUNICATE OFF
    PRINT AT 1,1:
20  INPUT 'Enter your name': name$
    PRINT name$
    OPTION COMMUNICATE ON
30  INPUT 'Enter your name': name$
    PRINT name$
40  END

RNH

Enter your name? Tester
Tester
Enter your name?

   .New mail on node FAST from FAST::MOLLY        (12:01:08)
```

PURPOSE:

Determines whether to display messages broadcast to the terminal.

DESCRIPTION:

The **OPTION COMMUNICATE** statement turns the display of messages ON or OFF. The default is OPTION COMMUNICATE ON. If there is a message (such as a mail message), INTOUCH displays it in the message area (at the bottom of the screen). If you enter OPTION COMMUNICATE OFF, INTOUCH will not display messages.

7 Program Directives

7.1 Compile Directives

The following sections describe the directives that are available for use in your programs. These directives are invoked when the compiler OLDs in a program and/or when a program is compiled.

7.1.1 %MESSAGE

FORMAT:

```
%MESSAGE 'quoted_text'
```

EXAMPLE:

```
10  %MESSAGE 'Including HELP module'
20  %INCLUDE 'tti_run:help'
30  END
```

DESCRIPTION:

This compiler directive displays a message, *quoted_text*, on the message line. The message is displayed when a program is OLD'ed into a workspace or compiled.

7.1.2 %MESSAGE ERROR

FORMAT:

```
%MESSAGE ERROR: 'quoted_text'
```

EXAMPLE:

```
10  %MESSAGE ERROR: 'Using experimental HELP module'
20  %INCLUDE 'tti_run:help'
30  END
```

DESCRIPTION:

This compiler directive rings the bell and displays an error message, *quoted_text*, on the message line. The message is displayed when a program is OLD'ed into a workspace or compiled.

7.1.3 %INCLUDE

FORMAT:

```
%INCLUDE 'file_spec'
```

EXAMPLE:

```
1000  %INCLUDE 'tti_run:example'
```

PURPOSE:

%INCLUDE allows you to put common subroutines into a separate file to be shared among applications.

DESCRIPTION:

%INCLUDE includes a source code, *file_spec*, file into the current INTOUCH program. The included file cannot contain line numbers. The default extension for the included file is **.INC**.

7.1.4 %INCLUDE CONDITIONAL

FORMAT:

```
%include CONDITIONAL: 'file_spec'
```

DESCRIPTION:

If the file to be included, *file_spec*, does not exist, no error is generated. This allows programs to conditionally include modules.

%INCLUDE CONDITIONAL includes a source code, *file_spec*, file into the current INTOUCH program if the file to be included is found. The included file cannot contain line numbers. The default extension for the included file is **.INC**.

7.1.5 %DEBUG

FORMAT:

```
%DEBUG INTOUCH_statement
```

EXAMPLE:

```
10  %DEBUG PRINT 'DEBUG Items'
20  END
```

DESCRIPTION:

The **%DEBUG** directive gives an error only if an image compile (/COMPILE) is being done. Use this directive to make sure that DEBUG code does not show up in production applications.

8 Assigning Variables

Variables specify storage locations. Numeric and string variables are assigned values with the LET statement. String variables can also be assigned values with the LSET, RSET and CSET statements.

The LET statement assigns the value of an expression to a variable. The expression is evaluated and its value is stored in the location specified by the variable. The data types of the expression and the variable must match. Thus, you must assign a string variable string values and you must assign numeric variables numeric values. A string variable must end with a dollar sign "$" unless you declare it. An integer variable must end with a percent sign (%) unless you declare it.

The DECLARE statement allows you to dispense with data type designations. DECLARE declares a variable as either string, integer or real. Once the variable has been declared, you do not need to attach a $ or % to it. Functions, arrays, etc., can also be declared with the DECLARE statement.

8.1 DECLARE

FORMAT:

```
DECLARE [STRING | INTEGER | REAL | DYNAMIC] var, var...
```

EXAMPLE:

```
10  DECLARE STRING name, sex
    DECLARE INTEGER age
    DECLARE REAL amount
    DECLARE DYNAMIC anything
20  INPUT 'Enter your name': name
    INPUT 'Enter your age': age
    INPUT 'Enter your sex': sex
    INPUT 'Enter an amount': amount
30  PRINT
    PRINT name; ' is a'; AGE; 'year old '; sex
    PRINT name; ' entered the amount'; amount
40  anything = 4 + 5
    PRINT 'This dynamic variable contains: '; anything
    anything = 'kitty'
    PRINT 'Now it contains a string value: '; anything
50  END

RNH
Enter your name? Sammy
Enter your age? 28
Enter your sex? male
Enter an amount? 25.38

Sammy is a 28 year old male
Sammy entered the amount 25.38
This dynamic variable contains:  9
Now it contains a string value: kitty
```

PURPOSE:

Use **DECLARE** to specify the data types of variables, functions, etc. Once you have declared the data type of a variable or function, you do not have to designate them with a trailing $ or %.

DESCRIPTION:

DECLARE declares the data type of a variable or function. The **STRING** option indicates that the following are string variables or functions. **INTEGER** declares integer numeric. **REAL** indicates real numeric. Only one of the three data type options can be used in each DECLARE statement. Any number of variables can be declared with a DECLARE statement.

DECLARE DYNAMIC declares one or more variables to be of type DYNAMIC. A variable of type DYNAMIC receives the data type of the data that you put into it.

You can find out the current data type of a dynamic variable with the DTYPE function. (See DTYPE(expr))

8.1.1 DECLARE STRUCTURE

FORMAT:

```
DECLARE STRUCTURE struc_name1 [, struc_name2 ...]
```

EXAMPLE:

```
10  DECLARE STRUCTURE str
    OPEN STRUCTURE cl: NAME 'tti_run:client'
    ASK STRUCTURE cl: ID cl_id$
    SET STRUCTURE str: ID cl_id$
20  EXTRACT STRUCTURE str
    END EXTRACT
    FOR EACH str
      PRINT str(#1); ' '; str(#2)
    NEXT str
30  END

RNH
20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

DECLARE STRUCTURE declares one or more symbols to be of type **STRUCTURE**. Once you have declared a symbol to be of type STRUCTURE, you can use it in statements such as SET STRUCTURE..ID to write generalized routines where you do not know at compile time which structure you are going to use.

Usage example: This statement could be used in the situation where you have a transaction structure and a transaction history structure and, optionally, want a report on one or the other. You could use one report program and the DECLARE STRUCTURE statement to declare which structure to use when the user makes the report selection.

8.1.2 DECLARE PREFIX

FORMAT:

```
DECLARE PREFIX str_expr
```

EXAMPLE:

```
irp$total   prefix is IRP
doit$now$   prefix is DOIT
doing_it$   there is no prefix
```

DESCRIPTION:

Any variable that contains as part of the body of its name a "$" is said to be a PREFIXED variable.

In some cases, you might want an IMPLICIT prefix for variables in a routine. To declare an IMPLICIT prefix:

```
DECLARE PREFIX prefix_name
```

EXAMPLE:

```
PRINT total    <-->  PRINT doit$total
PRINT name$    <-->  PRINT doit$name$
PRINT xyz$sum  <-->  PRINT xyz$sum    (prefix is explicitly given)
```

DESCRIPTION:

The **DECLARE PREFIX** statement causes all non-prefixed variable references, structure name references and field name references to inherit a prefix (i.e. "DOIT").

Declared prefixes are local to the routine that they are declared in. For this first release, nested declared prefixes are not supported. The following statements turn OFF any declared prefixing:

```
ROUTINE routine_name
END ROUTINE
DECLARE PREFIX none
```

In addition, at run time, prefixing is turned off before and after a string is **EXECUTE**d. (See Section 11.3 for information on the EXECUTE statement.) This allows you to locally prefix executed code:

```
10  a = 100
20  EXECUTE 'DECLARE PREFIX excode \ a = 45 \ PRINT a'
30  PRINT a
40  EXECUTE 'PRINT excode$a'
50  END

RNH
45
100
45
```

If you want the same prefix to be used throughout an entire procedure, you need to put a DECLARE PREFIX at the beginning of the procedure and after every ROUTINE statement.

When referencing structure field names and using a prefix, you must:

```
STRUC(#'field_name')

       so:

VEND(city)  --> vend(#'city')
```

Without the quotes, CITY would receive a prefix and would be an invalid field name.

Prefixing also applies to structure names. Therefore, if the structure is not opened within the prefixed routine that references it, you must have opened the structure with an explicit prefix.

```
 10  OPEN STRUCTURE ar$cust : NAME 'tti_run::customer', ACCESS INPUT
     GOSUB find_customer
     STOP

100  ROUTINE find_customer
       DECLARE PREFIX find
       c$ = '13727'
       SET STRUCTURE ar$cust, FIELD custnbr : KEY c$
       PRINT ar$cust(#'name')
     END ROUTINE
999  END
```

8.2 OPTION Statements

8.2.1 OPTION REQUIRE DECLARE

FORMAT:

```
OPTION REQUIRE DECLARE
```

EXAMPLE:

```
10  OPTION REQUIRE DECLARE
20  DECLARE STRING name, comment
30  INPUT 'Please enter your name': name
40  LINE INPUT 'Enter a comment in quotes': comment
50  PRINT name; ' says, '; comment
60  END

RNH
Please enter your name? George
Enter a comment in quotes? 'Have a nice day!'
George says, 'Have a nice day!'
```

DESCRIPTION:

OPTION REQUIRE DECLARE causes INTOUCH to require all variables in the program to be declared. If the OPTION REQUIRE DECLARE statement is used and a variable is left undeclared, INTOUCH will return an error when program execution is attempted. The OPTION REQUIRE DECLARE statement should occur before any DECLARE statements and before any variables are assigned.

8.2.2 OPTION BASE

FORMAT:

```
OPTION BASE  [0 | 1]
```

DESCRIPTION:

OPTION BASE sets the lowest subscript or base for arrays. The base can be either zero or one. If you use OPTION BASE 0, the lowest element of an array has the subscript zero (0). If you use OPTION BASE 1, the lowest element is subscript one (1).

See Section 10.1.3 for an example and detailed information on this statement.

8.3 LET

FORMAT:

```
[LET] var = expr
```

EXAMPLE:

```
10  INPUT 'Last name': last$
    INPUT 'First name': first$
    LET name$ = first$ & ' ' & last$
    PRINT name$
20  END

RNH
Last name? Taylor
First name? Rick
Rick Taylor
```

PURPOSE:

Use the **LET** statement to store information into a variable or data structure.

DESCRIPTION:

var is the variable being assigned a value. *expr* is an expression. The expression is evaluated and its result is assigned to the variable. The expression can be any INTOUCH expression (see Section 3.2.) The variable and the expression data types must match. For instance, if *var* is a string variable, *expr* must be a string expression.

NOTE: The keyword LET is optional. For example:

```
LET name$ = first$ & ' ' & last$
```

can be stated as:

```
name$ = first$ & ' ' & last$
```

When INTOUCH executes the LET statement, it first evaluates the expression on the right side of the equal sign. It then assigns this value to the variable on the left side of the equal sign. The variable represents a location in memory. The value of the expression is stored in this location. Each time a new value is assigned, the old value is lost and the new value is stored in its memory location.

Assigning Numeric Values

- Assigning a numeric value to a string variable results in an error.
- Assigning a real value to an integer variable causes the real number to be rounded to the nearest integer and assigned.

```
10  INPUT 'Amount': amount
    LET rounded% = amount
20  PRINT 'Real numeric amount:'; amount
    PRINT 'Integer amount (after rounding):'; rounded%
30  END

RNH
Amount? 1.54
Real numeric amount: 1.54
Integer amount (after rounding): 2
```

Assigning String Values

- Assigning a string value to a numeric variable results in an error.
- String variables are dynamic in length. String variables can be assigned values of up to 65535 characters in length.

8.4 LSET, RSET and CSET

LSET, RSET and CSET assign string values to variables. LSET left-justifies the new value. RSET right-justifies and CSET center-justifies it. These statements can only be used to assign *string* values.

LSET, RSET and CSET justify the new value *within* the length of the previous value. For example, in the following program, HEADING$ has a length of twenty characters and consists of twenty dots:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$; ')'
30  END

RNH
(....................)
```

In the following example, the RSET statement is used to assign the new value 'Page 12' to HEADING$. INTOUCH uses the current length of HEADING$, 20 characters, and replaces it with the new value, 'Page 12'. INTOUCH *right-justifies* this value by padding it with 13 leading spaces. Thus, HEADING$ still has a length of twenty characters.

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
    PRINT '('; heading$; ')'
20  RSET heading$ = 'Page 12'
    PRINT '('; heading$; ')'
30  END

RNH
(....................)
(             Page 12)
```

8.4.1 LSET

FORMAT:

```
LSET str_var = str_expr
```

EXAMPLE:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$; ')'
30  LSET heading$ = 'Page 12'
    PRINT '('; heading$; ')'
40  END

RNH
(.....................)
(Page 12              )
```

PURPOSE:

Use **LSET** to left-justify string data.

DESCRIPTION:

When INTOUCH executes an LSET statement, it evaluates the string expression on the right side of the equal sign. INTOUCH assigns the new value to the string variable and left-justifies this value within the length of the old value. If the new value has leading or trailing spaces, these spaces are carried over and the string is justified with those spaces. LSET can only be used with strings.

8.4.2 RSET

FORMAT:

```
RSET str_var = str_expr
```

EXAMPLE:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$ ;')'
30  RSET heading$ = 'Page 12'
    PRINT '(' ; heading$ ; ')'
40  END

RNH
(.....................)
(             Page 12)
```

PURPOSE:

Use **RSET** to right-justify string data.

DESCRIPTION:

When INTOUCH executes the RSET statement, it evaluates the string expression on the right side of the equal sign. INTOUCH assigns the new value to the string variable and right-justifies this value within the length of the old value. If the new value has leading or trailing spaces, these spaces are carried over and the string is justified with those spaces. RSET can only be used with strings.

8.4.3 CSET

FORMAT:

```
CSET str_var = str_expr
```

EXAMPLE:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$; ')'
30  CSET heading$ = 'Page 12'
    PRINT '('; heading$; ')'
40  END

RNH
(.....................)
(       Page 12       )
```

PURPOSE:

Use **CSET** to center string data.

DESCRIPTION:

When INTOUCH executes the CSET statement, it evaluates the expression on the right side of the equal sign. Next, INTOUCH assigns the new value to the string variable and centers it within the length of the old value. If the string value has leading or trailing spaces, the spaces are carried over and the string is centered with those spaces. CSET can only be used with strings.

8.5 LSET, RSET, CSET FILL

FORMAT:

```
LSET | RSET | CSET FILL str_expr: str_var = expr
```

EXAMPLE:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$; ')'
30  CSET FILL '*': heading$ = 'Page 12'
    PRINT '('; heading$; ')'
40  END
```

```
RNH
(.....................)
(*******Page 12******)
```

DESCRIPTION:

The value of *expr* is left-justified, right-justified or centered inside the *str_var*. The remaining part of the string is filled with the pattern specified by *str_expr*. If *str_expr* is the null string, no filling occurs—the remaining part of the string is left as is.

8.6 DATA, READ, RESTORE Statements

8.6.1 READ

FORMAT:

```
DATA [num_const | str_const] [,[num_const | str_const]...]
                .
                .
                .
READ [num_var | str_var] [,[num_var | str_var]...]
```

EXAMPLE:

```
10  DIM months$(6)
20  DATA January, February, March
30  DATA April, May, June
40  FOR i = 1 TO 6
      READ months$(i)
      PRINT months$(i)
    NEXT i
50  END
```

```
RNH
January
February
March
April
May
June
```

PURPOSE:

Use **DATA** and **READ** statements to assign data to variables in cases where the data will not change with successive runs of the program.

DESCRIPTION:

The READ and DATA statements assign data to variables. DATA specifies a list of data to assign. The data must be given as constants and can be string, numeric or integer types. Multiple data items must be separated by commas.

The READ statement specifies a list of variables to assign data to. The variables can be string, numeric or integer variables. They can be substrings, array elements, etc..

When INTOUCH executes the first READ statement, it goes to the first DATA statement and assigns the items in the DATA list to the variables in the READ list. The first variable in the READ list is assigned the first value in the DATA list. The second variable in the READ list is assigned the second value in the DATA list, and so on.

```
DATA constant, constant, constant, constant...
.
.       |         |         |         |
.
READ variable, variable, variable, variable...
```

If the data item contains a commas, the data item should be enclosed with single or double quotes. For example:

```
10  DIM amounts$(3)
20  DATA '$25,000', '$250,000', '$2,500,000'
30  READ amounts$(1), amounts$(2), amounts$(3)
40  PRINT amounts$(1), amounts$(2), amounts$(3)
50  END

RNH
$25,000            $250,000           $2,500,000
```

The variable types and data types must match or an exception will result. For example, if the third item in the DATA list is a string constant, and the third variable in the READ list is a numeric variable, an exception will result.

When the second READ statement is executed, INTOUCH starts reading from the first unread data item in the DATA list. For example:

```
 10  DIM months$(4)
 20  DATA January, February, March, April, May, June
 30  READ months$(1), months$(2)
 40  READ months$(3), months$(4)
 50  PRINT months$(1), months$(2), months$(3), months$(4)
 60  END

RNH
January     February      March      April
```

In the example above, when the first READ statement is executed, INTOUCH reads the months January and February. When the second READ statement is executed, INTOUCH will continue at the first unread month–March–and read it into months$(3).

If you attempt to read more data than exists, that is, if your READ list has more items than than your DATA list, an exception will result. You can avoid this by using the **RESTORE** statement to restore the DATA list and read from the beginning again.

The READ and DATA statements must occur in the same program unit. For example, you cannot not have your DATA statements in the main program unit and your matching READ statements in a subprogram.

See Section 8.6.2 for information on using RESTORE.

8.6.2 RESTORE

FORMAT:

```
RESTORE
```

EXAMPLE:

```
10  DIM months$(3)
    DIM more_months$(3)
20  DATA January, February, March
30  FOR i = 1 TO 3
      READ months$(i)
      PRINT months$(i)
    NEXT i
40  RESTORE
    PRINT
50  FOR i = 1 TO 3
      READ more_months$(i)
      PRINT more_months$(i)
    NEXT i
60  END

RNH
January
February
March

January
February
March
```

PURPOSE:

Use **RESTORE** when you want to use the same set of data (from a DATA statement) for a number of READ statements.

DESCRIPTION:

RESTORE restores the DATA statements in a program unit so that you can use them again. When the RESTORE statement is executed, all the DATA statements which have been read are restored. The next READ statement causes INTOUCH to go back to the first DATA statement and begin assigning the items in its list.

In the example program, the months will be read and assigned to the array MONTHS$. When the RESTORE is executed, the DATA statements will be restored. When the READ statement is executed, the months will be read into the new array MORE_MONTHS$.

9 Set and Ask Statements

SET and **ASK** statements find and change characteristics within an INTOUCH program. SET sets various characteristics, and ASK returns the value of various characteristics. SET and ASK have several different options.

You can use SET and ASK on a channel of a device. Use SET and ASK if you need to do some special printing to the terminal. You can use ASK to find the terminal's current print zone width and right margin setting. If they are not correct, you can use SET to change them and then print your material to the terminal.

Note

For information on SET #chnl and ASK #chnl statements, refer to Chapter 15, File Handling.

For information on SET STRUCTURE and ASK STRUCTURE statements, refer to Chapter 14, Data Structure Statements.

9.1 SET AUTOEXIT

FORMAT:

```
SET AUTOEXIT num_expr
```

EXAMPLE:

```
10  SET AUTOEXIT 1
20  DO
      INPUT 'Who': a$
      IF  _EXIT  OR  _BACK  THEN  EXIT DO
30  LOOP
    PRINT 'Finished'
40  END

RNH
Who? Greg
Who? Sammy
Who?              (User fails to respond within 1 minute.)

INTOUCH
```

PURPOSE:

Use to slowly back a user out of a program if the terminal is left idle.

DESCRIPTION:

SET AUTOEXIT causes an idle terminal waiting at an input prompt to set _EXIT to TRUE and complete the input. *num_expr* is the length of time in minutes. If *num_expr* is assigned a value of 0, INTOUCH turns off the feature.

If the terminal is left idle for *num_expr* minutes at the input prompt, EXIT will be forced as the response, the _EXIT flag will be set on and the program will execute the code indicated for _EXIT, if any.

9.2 SET BACK ON | OFF

9.2.1 SET BACK ON

FORMAT:

```
SET BACK ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _BACK
30  SET BACK ON
40  PRINT _BACK
50  END

RNH
Name? TESTER_____________________________
0
1
```

DESCRIPTION:

SET BACK ON sets the internal variable _BACK to TRUE (1).

9.2.2 SET BACK OFF

FORMAT:

```
SET BACK OFF
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _BACK
30  SET BACK OFF
40  PRINT _BACK
50  END

RNH
Name? \_______________________________
1
0
```

DESCRIPTION:

SET BACK OFF sets the internal variable _BACK to FALSE (0).

9.3 SET ERROR ON | OFF

9.3.1 SET ERROR ON

FORMAT:

```
SET ERROR ON
```

EXAMPLE:

```
10  DO
      INPUT 'Enter the age': age
      IF  age < 1  THEN
        PRINT 'Too young:'; age
        SET ERROR ON
      ELSE
        SET ERROR OFF
      END IF
    LOOP WHILE _ERROR
20  END

RNH
Enter the age? .5
Too young: .5
Enter the age? 38
```

PURPOSE:

Use to set the **_ERROR** flag on.

DESCRIPTION:

_ERROR is a general purpose error flag. You can use it to indicate that an error has occurred, and to test later on whether an error has occurred.

The following statements SET the _ERROR flag:

- SET ERROR ON
- the MESSAGE ERROR: statement
- the execution of the END WHEN statement

9.3.2 SET ERROR OFF

FORMAT:

```
SET ERROR OFF
```

EXAMPLE:

```
10  DO
      INPUT 'Enter the age': age
      IF  age < 1  THEN
        PRINT 'Too young:'; age
        SET ERROR ON
      ELSE
        SET ERROR OFF
      END IF
    LOOP WHILE _ERROR
30  END

RNH
Enter the age? .5
Too young: .5
Enter the age? 38
```

PURPOSE:

Use to clear the **_ERROR** flag.

DESCRIPTION:

_ERROR is a general purpose error flag. You can use it to indicate that an error has occurred, and to test later on whether an error has occurred.

The following statements CLEAR the _ERROR flag:

- SET ERROR OFF
- any GOSUB or DISPATCH statement
- the WHEN EXCEPTION IN statement

9.4 ASK ERRORS

FORMAT:

```
ASK ERRORS num_var
```

EXAMPLE:

```
10  DO
      INPUT 'Enter the age': age
      IF  age < 1  THEN
        MESSAGE ERROR: age; ' Too Young'
        REPEAT DO
      ELSE
        EXIT DO
      END IF
    LOOP
20  ASK ERRORS num_errors
    PRINT 'Errors:'; num_errors
30  END
```

```
RNH
Enter the age? 0                  0 Too Young
Enter the age? .5                 .5 Too Young
Enter the age? 21
Errors: 2
```

DESCRIPTION:

ASK ERRORS asks for the number of user errors. The MESSAGE ERROR: statement increments this internal counter.

9.5 SET EXIT ON | OFF

9.5.1 SET EXIT ON

FORMAT:

```
SET EXIT ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _EXIT
30  SET EXIT ON
40  PRINT _EXIT
50  END
```

```
RNH
Name? ELAINE_____________________________
 0
 1
```

DESCRIPTION:

SET EXIT ON sets the internal variable _EXIT to TRUE (1).

9.5.2 SET EXIT OFF

FORMAT:

```
SET EXIT OFF
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _EXIT
30  SET EXIT OFF
40  PRINT _EXIT
50  END

RNH
Name? EXIT_____________________________
 1
 0
```

DESCRIPTION:

SET EXIT OFF sets the internal variable _EXIT to FALSE (0).

9.6 SET HELP ON | OFF

9.6.1 SET HELP ON

FORMAT:

```
SET HELP ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _HELP
30  SET HELP ON
40  PRINT _HELP
50  END

RNH
Name? MIKE_____________________________
 0
 1
```

DESCRIPTION:

SET HELP ON sets the internal variable _HELP to TRUE (1).

9.6.2 SET HELP OFF

FORMAT:

```
SET HELP OFF
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _HELP
30  SET HELP OFF
40  PRINT _HELP
50  END

RNH
Name? HELP_____________________________
 1
 0
```

DESCRIPTION:

SET HELP OFF sets the internal variable _HELP to FALSE (0).

9.7 ASK KEYSTROKES

FORMAT:

```
ASK KEYSTROKES num_var
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    PRINT 'Hello '; name$
20  ASK KEYSTROKES strokes
    PRINT 'Keystrokes:'; strokes
30  END

RNH
Please enter your name? Maryanne
Hello Maryanne
Keystrokes: 8
```

DESCRIPTION:

ASK KEYSTROKES asks for the number of user-entered keystrokes.

9.8 ASK | SET MARGIN

9.8.1 ASK MARGIN

FORMAT:

```
ASK MARGIN num_var
```

DESCRIPTION:

ASK MARGIN finds the right margin of the device specified and assigns its value to the numeric variable, *num_var*.

9.8.2 SET MARGIN

FORMAT:

```
SET MARGIN num_expr
```

EXAMPLE:

```
10  PRINT REPEAT$('.' ,200)
    PRINT
    ASK MARGIN old_marg
20  INPUT 'What do you want the margin set to': new_marg
30  SET MARGIN new_marg
40  PRINT REPEAT$('.' ,200)
    SET MARGIN old_marg
50  END

RNH
.............................................
.............................................
.............................................
.............................................

What do you want the margin set to? 20
.....................
.....................
.....................
.....................
```

DESCRIPTON:

SET MARGIN sets the right margin on the device specified to the number indicated. *num_expr* specifies the column to set the margin to. The margin must be greater than zonewidth.

9.9 ASK | SET MESSAGELINE

9.9.1 ASK MESSAGELINE

FORMAT:

```
ASK MESSAGELINE num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:;
20  ASK MESSAGELINE orig_mess_line
    MESSAGE 'Current message line is '; orig_mess_line
    DELAY 4
30  new_line = 12
    SET MESSAGELINE new_line
    MESSAGE 'New message line is '; new_line
    DELAY 4
40  SET MESSAGELINE orig_mess_line
    MESSAGE 'Message line has been reset to its original position'
50  END

RNH
                    Current message line is 23

                      New message line is 12

        Message line has been reset to its original position
```

DESCRIPTION:

The **MESSAGELINE** option of the ASK statement returns the line number on which the messages are displayed. This numeric value is stored in *num_var*.

9.9.2 SET MESSAGELINE

FORMAT:

```
SET MESSAGELINE num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:;
20  ASK MESSAGELINE orig_mess_line
    MESSAGE 'Current message line is '; orig_mess_line
    DELAY 4
30  new_line = 12
    SET MESSAGELINE new_line
    MESSAGE 'New message line is '; new_line
    DELAY 4
40  SET MESSAGELINE orig_mess_line
    MESSAGE 'Message line has been reset to its original position'
50  END

RNH
                    Current message line is 23
```

```
New message line is 12

Message line has been reset to its original position
```

DESCRIPTION:

The **MESSAGELINE** option of the of the SET statement sets the line on which the next message is displayed. If **SET MESSAGELINE 0** is used, no messages will be displayed.

9.10 ASK PAGESIZE

FORMAT:

```
ASK PAGESIZE num_var
```

EXAMPLE:

```
10  ASK PAGESIZE no_lines
    PRINT 'There are'; no_lines; 'lines or rows on this screen'
20  END

RNH
There are 24 lines or rows on this screen
```

DESCRIPION:

ASK PAGESIZE returns the number of rows or lines of terminal screen output.

9.11 SET PORT ON | OFF

9.11.1 SET PORT ON

FORMAT:

```
SET PORT ON
```

EXAMPLE:

```
10  PRINT 'About to print to my printer port'
20  SET PORT ON
    PRINT 'This is on the printer port'
    SET PORT OFF
30  PRINT 'Back to the screen again'
40  END

RNH
About to print to my printer port
Back to the screen again
```

PURPOSE:

Allows a program to use an attached printer port on a terminal.

DESCRIPTION:

SET PORT ON | OFF turns the printer port ON or OFF. SET PORT ON turns on the attached port. SET PORT OFF turns off the attached port.

TTI_PRINTER_TYPE

To indicate the type of printer that is attached to the terminal's printer port, you can use the symbol **TTI_PRINTER_TYPE**.

To set up the printer type, enter the following at the system prompt (DCL level):

```
$ TTI_PRINTER_TYPE == 'printer_type'
```

where "printer_type" is one of the following:

DEC
HP
OTHER

Example: $ TTI_PRINTER_TYPE == 'HP'

If the symbol is not defined, the type DEC is assumed.

If a terminal has a non-DEC printer port, no escape sequences are sent to the printer when you print to the printer port.

9.11.2 SET PORT OFF

FORMAT:

```
SET PORT OFF
```

EXAMPLE:

```
10  PRINT 'About to print to my printer port'
20  SET PORT ON
    PRINT 'This is on the printer port'
    SET PORT OFF
30  PRINT 'Back to the screen again'
40  END

RNH
About to print to my printer port
Back to the screen again
```

PURPOSE:

Disable printing to an attached printer port on a terminal.

DESCRIPTION:

SET PORT OFF turns off the attached port.

9.12 ASK RESPONSES

FORMAT:

```
ASK RESPONSES num_var
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    INPUT 'What day is this': what_day$
    PRINT 'Hello '; name$
    PRINT 'Have a good '; what_day$
20  ASK RESPONSES answers
    PRINT
    PRINT 'Responses:'; answers
30  END

RNH
Please enter your name? Ginger
What day is this? Wednesday
Hello Ginger
Have a good Wednesday

Responses: 2
```

DESCRIPION:

ASK RESPONSES asks for the number of completed input responses.

9.13 SET SCROLL

FORMAT:

```
SET SCROLL num_expr1, num_expr2
```

EXAMPLE:

```
10  FRAME OFF
    PRINT AT 21, 1: 'This text will not scroll.'
    SET SCROLL 5, 20
    PRINT AT 20, 1:;
    DELAY 1
    PRINT 'This'
    DELAY 1
    PRINT 'text'
    DELAY 1
    PRINT 'will'
    DELAY 1
    PRINT 'scroll.'
    DELAY 1
20  SET SCROLL  1,24
30  END

RNH

This
text
will
scroll

This text will not scroll.
```

DESCRIPTION:

This statement sets up a scrolling region from line *num_expr1* to line *num_expr2*.

9.14 ASK | SET SEED

FORMAT:

```
ASK SEED num_var
SET SEED num_expr
```

EXAMPLE:

```
10 RANDOMIZE
   ASK SEED seed_num
   FOR i = 1 TO 3
     PRINT RND(1000)
   NEXT i
   PRINT 'Reset the random sequence'
   SET SEED seed_num
   FOR i = 1 TO 3
     PRINT RND(1000)
   NEXT i
20 END

RNH
 608
 88
 506
Reset the random sequence
 608
 88
 506
```

PURPOSE:

Allows you to set or reset the pseudo-random number sequence.

DESCRIPTION:

ASK SEED returns the current starting point of a pseudo-random sequence and stores the number in *num_var*.

SET SEED sets the starting point of a pseudo-random sequence with the number in *num_expr*.

9.15 ASK | SET SYSTEM

There are a number of ASK SYSTEM and SET SYSTEM statements. These are described in the following sections. The ASK/SET statements ask about and set various system operation features.

9.15.1 ASK SYSTEM: COMMENT

FORMAT:

```
ASK SYSTEM: COMMENT str_var
```

EXAMPLE:

```
10  SET SYSTEM: COMMENT 'Invoice Entry'
20  ASK SYSTEM: COMMENT c$
30  PRINT c$
40  END

RNH
Invoice Entry
```

DESCRIPTION:

The **ASK SYSTEM: COMMENT** statement asks for the INTOUCH operating system comment for the process.

9.15.2 SET SYSTEM: COMMENT

FORMAT:

```
SET SYSTEM: COMMENT str_expr
```

EXAMPLE:

```
10  SET SYSTEM: COMMENT 'Invoice Entry'
20  ASK SYSTEM: COMMENT c$
30  PRINT c$
40  END

RNH
Invoice Entry
```

DESCRIPTION:

The **SET SYSTEM: COMMENT** statement loads the INTOUCH operating system comment area with the specified string. This statement can be used in combination with the TTI_RUN:ISHOW.COM command procedure. The INTOUCH program sets the comment to some text, such as the name of the specific routine being executed. When you run TTI_RUN:ISHOW.COM, you see the specific INTOUCH program and routine within that program being run by all INTOUCH users.

9.15.3 ASK SYSTEM: DIRECTORY

FORMAT:

```
ASK SYSTEM: DIRECTORY str_var
```

EXAMPLE:

```
10  ASK SYSTEM: DIRECTORY z$
    PRINT 'Current directory is: '; z$
20  END

RNH
Current directory is: USER:[PAYROLL]
```

DESCRIPTION:

ASK SYSTEM: DIRECTORY asks the operating system for the current default device and directory.

9.15.4 SET SYSTEM: DIRECTORY

FORMAT:

```
SET SYSTEM: DIRECTORY str_var
```

EXAMPLE:

```
10  ASK SYSTEM: DIRECTORY z0$
    PRINT 'Current directory     '; z0$
    SET SYSTEM: DIRECTORY 'SYS$LOGIN'
    ASK SYSTEM: DIRECTORY z1$
    PRINT 'Directory set to      '; z1$
    DELAY 2
    SET SYSTEM: DIRECTORY z0$
    PRINT 'Directory set back to '; z0$
20  END

RNH
Current directory      USER:[TESTER.INTOUCH]
Directory set to       USER:[TESTER]
Directory set back to  USER:[TESTER.INTOUCH]
```

DESCRIPTION:

SET SYSTEM: DIRECTORY sets the default device and directory.

9.15.5 ASK SYSTEM, LOGICAL: VALUE

FORMAT:

```
ASK SYSTEM, LOGICAL str_expr: VALUE str_var
```

EXAMPLE:

```
10  ASK SYSTEM, LOGICAL "SYS$SCRATCH": VALUE scr$
20  PRINT '"SYS$SCRATCH" points to: '; scr$
30  END

RNH
"SYS$SCRATCH" points to: USER:[TESTER]
```

DESCRIPTION:

ASK SYSTEM, LOGICAL asks the operating system to translate the logical name in *str_expr* and place the result into the variable specified by *str_var*.

9.15.6 SET SYSTEM, LOGICAL: VALUE

FORMAT:

```
SET SYSTEM, LOGICAL str_expr1: VALUE str_expr2
```

EXAMPLE:

```
10  SET SYSTEM, LOGICAL 'SYS$SCRATCH': VALUE 'USER:[TESTER]'
    ASK SYSTEM, LOGICAL 'SYS$SCRATCH': VALUE z$
    PRINT 'Logical set to '; z$
20  END

RNH
Logical set to USER:[TESTER]
```

DESCRIPTION:

This statement sets the operating system logical name in *str_expr1* to the value in *str_expr2*.

9.15.7 ASK SYSTEM: MODE

FORMAT:

```
ASK SYSTEM: MODE str_var
```

EXAMPLE:

```
10  ASK SYSTEM: MODE process_mode$
20  PRINT 'Process Mode: '; process_mode$
30  END

RNH
Process mode: INTERACTIVE
```

DESCRIPTION:

This statement returns the mode of the process which is one of the following:

INTERACTIVE
BATCH
NETWORK
OTHER

9.15.8 ASK SYSTEM: PARAMETER

FORMAT:

```
ASK SYSTEM: PARAMETER str_var
```

EXAMPLE:

```
 1  PROGRAM test_param.int
10  ASK SYSTEM: PARAMETER pdata$
20  PRINT 'Parameter was: '; pdata$
30  END

INTOUCH

SAVE 'test_param'

INTOUCH
                                parameter
                              ------------
                             |            |
$ INTOUCH/SOURCE test_param show parameter
Parameter was: SHOW PARAMETER

INTOUCH
```

PURPOSE:

ASK SYSTEM: PARAMETER returns any parameter from the command line given after the program name and places it in *str_var*.

DESCRIPTION:

ASK SYSTEM: PARAMETER lets you obtain the command line that invoked INTOUCH. The statement gives you the part of the command line after the program name.

9.15.9 ASK SYSTEM, PASS: SUCCESS

FORMAT:

```
ASK SYSTEM, PASS: SUCCESS num_var
```

EXAMPLE:

```
10  OPEN #1: NAME 'report.lis', ACCESS OUTPUT
    PRINT #1: 'sample report'
    CLOSE #1
    PASS 'PRINT/HOLD REPORT.LIS'
    ASK SYSTEM, PASS: SUCCESS okay
    IF  okay  THEN
      PRINT 'Report queued for printing'
    ELSE
      PRINT 'Unable to queue report for printing'
    END IF
20  END

RNH
Report queued for printing
```

DESCRIPTION:

This statement asks the operating system to return a **TRUE** (1) if true or **FALSE** (0) if false indicating whether or not the last PASS command executed successfully.

9.15.10 ASK SYSTEM: PROCESS

FORMAT:

```
ASK SYSTEM: PROCESS str_var
```

EXAMPLE:

```
10  ASK SYSTEM: PROCESS process$
20  PRINT 'Process is: '; process$
30  END

RNH
Process is: TESTER_13B_3
```

DESCRIPTION:

ASK SYSTEM: PROCESS *str_var* asks the operating system for the current process name.

9.15.11 SET SYSTEM: PROCESS

FORMAT:

```
SET SYSTEM: PROCESS str_expr
```

EXAMPLE:

```
10  ASK SYSTEM: PROCESS process$
    curr_process$ = process$
    PRINT 'Current process is: '; curr_process$
20  new_process$ = 'DO_TEST'
    SET SYSTEM: PROCESS new_process$
    ASK SYSTEM: PROCESS process$
    PRINT 'New process is: '; process$
30  SET SYSTEM: PROCESS curr_process$
    ASK SYSTEM: PROCESS process$
    PRINT 'Old process restored: '; process$
40  END

RNH
Current process is: TESTER_13B_3
New process is: DO_TEST
Old process restored: TESTER_13B_3
```

DESCRIPTION:

SET SYSTEM: PROCESS *str_expr* changes the operating system process name to *str_expr*.

9.15.12 ASK SYSTEM: RIGHTS

FORMAT:

```
ASK SYSTEM: RIGHTS str_var
```

EXAMPLE:

```
10  ASK SYSTEM: RIGHTS process_rights$
    PRINT 'Your process rights are: '; process_rights$
20  END

RNH
Your process rights are: FAST_ACCESS,TEST_ACCESS
```

DESCRIPTION:

ASK SYSTEM: RIGHTS asks the operating system to return a list of the rights explicitly granted to the calling process.

9.15.13 ASK SYSTEM, SYMBOL: VALUE

FORMAT:

```
ASK SYSTEM, SYMBOL str_expr: VALUE str_var
```

EXAMPLE:

```
10  ASK SYSTEM, SYMBOL 'INTOUCH': VALUE symbol$
    PRINT 'Value of symbol INTOUCH is: '; symbol$
20  END

RNH
Value of symbol INTOUCH is: $intouch_image:
```

DESCRIPTION:

This statement asks the operating system to translate the symbol name in *str_expr* and place the result into the variable specified by *str_var*.

9.15.14 SET SYSTEM, SYMBOL: VALUE

FORMAT:

```
SET SYSTEM, SYMBOL str_expr1: VALUE str_expr2
```

EXAMPLE:

```
10  SET SYSTEM, SYMBOL 'mysym': VALUE 'hello'
    ASK SYSTEM, SYMBOL 'mysym': VALUE z$
    PRINT 'Symbol set to '; z$
20  END

RNH
Symbol set to HELLO
```

DESCRIPTION:

This statement sets the operating system symbol name in *str_expr1* to the value in *str_expr2*.

9.15.15 ASK SYSTEM: USER

FORMAT:

```
ASK SYSTEM: USER str_var
```

EXAMPLE:

```
10  ASK SYSTEM : USER uname$
20  PRINT 'User is: '; uname$
30  END

RNH
User is: TESTER
```

PURPOSE:

This statement returns the operating system name or ID for the user. Under VMS this is the **Username**.

9.16 ASK | SET WINDOW

There are various ASK WINDOW and SET WINDOW statements. These are described in the following sections. The ASK/SET WINDOW statements ask about and reset different screen features.

9.16.1 ASK WINDOW AREA

FORMAT:

```
ASK WINDOW AREA row, col, row, col: DATA str_var
```

EXAMPLE:

```
10  CLEAR
20  PRINT AT 10, 4: 'Mary had a';
    PRINT AT 11, 4: 'little lamb';
30  ASK WINDOW AREA 10, 4, 11, 15: DATA x$
40  PRINT
    PRINT x$
50  END
```

```
RNH
   Mary had a
   little lamb
Mary had a
          little lamb
```

DESCRIPTION:

This statement reads the text displayed on the screen within the area defined by the given upper-left/lower-right coordinates into a string variable, *str_var*. The coordinates are specified by upper-left row, upper-left column, lower-right row, lower-right column. The statement returns a <LF> delimited string. No screen attributes are stored.

9.16.2 SET WINDOW AREA

FORMAT:

```
SET WINDOW AREA row, col, row, col: DATA str_expr
```

EXAMPLE:

```
10  CLEAR
20  x$ = 'Mary had a' + chr$(10) + 'little lamb'
30  SET WINDOW AREA 6, 5, 7, 15: DATA x$
40  END

RNH

    Mary had a
    little lamb
```

DESCRIPTION:

This statement sets the screen within the area defined by the given upper-left/lower-right coordinates to the specified string. This is the mirror image of ASK WINDOW AREA row, col, row, col: DATA str_var.

9.16.3 ASK WINDOW: COLUMN

FORMAT:

```
ASK WINDOW: COLUMN num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
20  ASK WINDOW: COLUMN cur_col
30  PRINT 'Cursor is at column'; cur_col
40  END

RNH
Cursor is at column 10
```

DESCRIPTION:

This statement returns the current column of the cursor's position.

9.16.4 SET WINDOW: COLUMN

FORMAT:

```
SET WINDOW: COLUMN num_expr
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
20  SET WINDOW: COLUMN 4
30  PRINT 'Hi!'
40  END

RNH
   Hi!
```

DESCRIPTION:

This statement positions the cursor at the *num_expr* column within the current row.

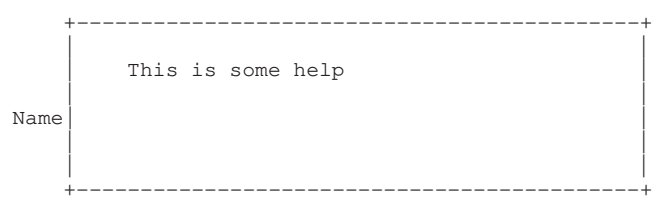
9.16.5 ASK | SET WINDOW: CURRENT

FORMAT:

```
ASK WINDOW: CURRENT str_var
SET WINDOW: CURRENT str_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,20, UNDERLINE: 'Sample screen'
20  DO
      LINE INPUT 'Name', AT 5,1, LENGTH 30: name$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  _HELP  THEN
        ASK WINDOW: CURRENT old_w$
        CLEAR AREA BOX: 1, 5, 10, 50
        PRINT AT 3, 10, REVERSE: 'This is some help'
        DELAY
        SET WINDOW: CURRENT old_w$
        REPEAT DO
      END IF
30  END DO
40  END

RNH
```




```
                Press the RETURN key to continue
```

DESCRIPTION:

ASK WINDOW: CURRENT and **SET WINDOW: CURRENT** allow you to save the image of the current screen and later restore it easily. This is useful for help messages and menus, where you must temporarily change the screen and then want to restore it back to what it was.

9.16.6 ASK WINDOW: DATA

FORMAT:

```
ASK WINDOW: DATA str_var
```

EXAMPLE:

```
10  CLEAR
20  PRINT AT 10, 4: 'Mary had a';
    PRINT AT 11, 4: 'little lamb';
30  ASK WINDOW: DATA x$
40  PRINT
50  PRINT x$
60  END

RNH
   Mary had a
   little lamb
.
.
.

Mary had a
             little lamb
```

DESCRIPTION:

This statement reads the text displayed on the whole screen into a string variable. The statement returns a <LF> delimited string. No screen attributes are stored.

9.16.7 SET WINDOW: DATA

FORMAT:

```
SET WINDOW: DATA str_expr
```

EXAMPLE:

```
10  CLEAR
20  x$ = 'Mary had a' + chr$(10) + 'little lamb'
30  SET WINDOW: DATA x$
40  END

RNH
Mary had a
little lamb
```

DESCRIPTION:

This statement sets the whole screen to the specified string. This is the mirror image of ASK WINDOW: DATA str_var.

9.16.8 ASK | SET WINDOW: KEYMAP

FORMAT:

```
ASK WINDOW: KEYMAP str_var
SET WINDOW: KEYMAP str_expr
```

EXAMPLE:

```
10  PRINT 'Save the current keymap, reset keymap to default value.'
    ASK WINDOW: KEYMAP old_keymap$
    SET WINDOW: KEYMAP ''

    PRINT 'Changing DOWN key to be the EXIT key'
    SET WINDOW KEYSTROKE 'down': VALUE '_exit'
    LINE INPUT 'Press the DOWN key': down$

    PRINT 'Changing dollar sign key to *'
    SET WINDOW KEYSTROKE '$': VALUE '*'
    LINE INPUT 'Press the dollar sign key, then RETURN': e$

    PRINT 'Restore saved keymap'
    SET WINDOW: KEYMAP old_keymap$
    LINE INPUT 'Press the DOWN key': down$
    LINE INPUT 'Press the dollar sign key, then RETURN' : e$
20  END

RNH
Save the current keymap, reset keymap to default value.
Changing DOWN key to be the EXIT key
Press the DOWN key? EXIT
Changing dollar sign key to *
Press the dollar sign key, then RETURN? *
Restore saved keymap
Press the DOWN key?
Press the dollar sign key, then RETURN? $

INTOUCH
```

PURPOSE:

To allow a generalized routine to save the current keymap, change the meaning of keys, and then restore the original keymap when done.

DESCRIPTION:

ASK WINDOW: KEYMAP and **SET WINDOW: KEYMAP** allow you to save the image of the keymap and later restore it. This is helpful for applications where you must temporarily change the meaning of the keys using the SET WINDOW KEYSTROKE statement. You can restore the keymap to its default setting by using SET WINDOW: KEYMAP.

9.16.9 SET WINDOW KEYSTROKE: VALUE

FORMAT:

```
SET WINDOW KEYSTROKE str_expr1: VALUE str_expr2
```

EXAMPLE:

```
10  PRINT 'Saving the current keymap.'
    ASK WINDOW: KEYMAP old_keymap$
    SET WINDOW: KEYMAP ''

    PRINT 'Changing DOWN key to be the EXIT key'
    SET WINDOW KEYSTROKE 'down': VALUE '_exit'
    LINE INPUT 'Press the DOWN key': down$

    PRINT 'Changing dollar sign key to *'
    SET WINDOW KEYSTROKE '$': VALUE '*'
    LINE INPUT 'Press the dollar sign key, then RETURN': e$

    PRINT 'Restoring saved keymap.'
    SET WINDOW: KEYMAP old_keymap$
    LINE INPUT 'Press the DOWN key': down$
    LINE INPUT 'Press the dollar sign key, then RETURN' : e$
20  END

RNH
Saving the current keymap.
Changing DOWN key to be the EXIT key
Press the DOWN key? EXIT
Changing dollar sign key to *
Press the dollar sign key, then RETURN? *
Restoring saved keymap.
Press the DOWN key?
Press the dollar sign key, then RETURN? $

INTOUCH
```

PURPOSE:

SET WINDOW KEYSTROKE allows you to change the meaning of a keystroke within an INTOUCH program. This allows you to completely redefine the keyboard for a given application.

DESCRIPTION:

str_expr1 describes the name of the key that you want to change. It can be a single keystroke name, or a comma separated list of names. Keystroke names can be a single letter, or the name of the letter (such as Tab, or Ctrl/Z).

str_expr2 defines the new meaning of the keystroke. A keystroke meaning consists of one or two components. The keystroke value, and the keystroke concept. For example, the Ctrl/Z key usually has a value of CHR$(26), and the concept of EXIT. The keystroke value and/or the keystroke concept can be changed. If changing both a value and a concept, separate the two with a comma. You can restore the original meaning of the key by using " ".

The following keystroke concepts are supported:

Table 9–1 Supported Keystroke Concepts

Concept name	Description
_EXIT	an EXIT key
_BACK	a BACK key
_HELP	a HELP key
_IGNORE	ignore this keystroke
_INVALID	beep when pressed
_TERMINATOR	keystroke is a line terminator

9.16.10 ASK WINDOW: ROW

FORMAT:

```
ASK WINDOW: ROW num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
20  ASK WINDOW: ROW cur_row
30  PRINT 'Cursor is at row'; cur_row
40  END

RNH
Cursor is at row 5
```

DESCRIPTION:

This statement returns the current row of the cursor's position.

9.16.11 SET WINDOW: ROW

FORMAT:

```
SET WINDOW: ROW num_expr
```

EXAMPLE:

```
10  CLEAR
20  SET WINDOW: ROW 3
30  PRINT 'Hi!'
40  END

RNH

Hi!
```

DESCRIPTION:

This statement positions the cursor at the *num_expr* row within the current column.

9.16.12 ASK WINDOW: TYPEAHEAD

FORMAT:

```
ASK WINDOW: TYPEAHEAD str_var
```

EXAMPLE:

```
10  DO
      GOSUB process
      ASK WINDOW: TYPEAHEAD z$
      IF  POS(UCASE$(z$), 'STA') > 0  THEN  GOSUB show_status
      IF  POS(z$, CHR$(26)) > 0  THEN  EXIT DO
    LOOP
    STOP

    ROUTINE process
      DELAY 1                      ! simulated processing
      PRINT '.';
    END ROUTINE

    ROUTINE show_status
      PRINT
      PRINT 'Showing status'
      SET WINDOW: TYPEAHEAD ''
    END ROUTINE
20  END

RNH
.......       (type STA)
showing status
.......       (press CTRL/Z )
```

DESCRIPTION:

ASK WINDOW: TYPEAHEAD gets data from the typeahead buffer. You can use this statement to determine, for example, whether the user has typed Ctrl/Z or other special keystrokes. Asking for typeahead data does not lose what is already in the typeahead buffer.

9.16.13 SET WINDOW: TYPEAHEAD

FORMAT:

```
SET WINDOW: TYPEAHEAD str_expr
```

EXAMPLE:

```
10  SET WINDOW: TYPEAHEAD 'FRED' + CHR$(13)
    INPUT 'Name': name$
    PRINT name$
20  END

RNH
Name? FRED
FRED
```

DESCRIPTION:

SET WINDOW: TYPEAHEAD puts data into the typeahead buffer as though the user had typed the data in from the terminal.

9.17 ASK | SET ZONEWIDTH

9.17.1 ASK ZONEWIDTH

FORMAT:

```
ASK ZONEWIDTH num_var
```

EXAMPLE:

```
10  ASK ZONEWIDTH x
    PRINT 'The current print zone width is'; x
20  END

RNH
The current print zone width is 20
```

DESCRIPTION:

ASK ZONEWIDTH finds the print zone width of the device specified and assigns the value to the numeric variable, *num_var*.

9.17.2 SET ZONEWIDTH

FORMAT:

```
SET ZONEWIDTH num_expr
```

EXAMPLE:

```
10  PRINT 1,2,3
20  SET ZONEWIDTH 10
    PRINT 1,2,3
30  SET ZONEWIDTH 20
40 END
RNH
1                   2                   3
1         2         3
```

DESCRIPTION:

SET ZONEWIDTH sets the print zone width of the device specified to the number designated. *num_expr* indicates the width to set the device's print zones.

10 Defining Arrays

Arrays are a type of variable. They are used to store and manipulate tables of variable information. An array must be defined before it is used in a program. Array variables are described in Section 3.2.5.1, Arrays.

10.1 Dimensioning Arrays

Arrays are dimensioned with a **DIM** statement. The **REDIM** statement can be used to redimension an array–that is, to change the dimensions of an array which has been defined with the DIM statement. The **OPTION BASE** statement changes the default low bound. By default, the low bound is 1.

About arrays:

- The highest bound allowed is 2147483648.
- The lowest bound allowed is -2147483647.
- Each array can have up to 32 dimensions.
- See Section 3.2.5.1, Arrays for more information on the storage allocation of arrays.
- If an array element outside of the specified bounds is referenced, an exception is generated.

10.1.1 DIM

FORMAT:

```
DIM [INTEGER | REAL | STRING]
          array_name ([int_expr TO] int_expr [, ...])
```

EXAMPLE:

```
10  DIM name$(4)
20  FOR i = 1 TO 4
      INPUT 'Enter a name': name$(i)
    NEXT i
30  PRINT
40  FOR i = 1 TO 4
      PRINT i; ' '; name$(i)
    NEXT i
50  END

RNH
Enter a name? Jim
Enter a name? Jane
Enter a name? Bob
Enter a name? Betty
```

```
1  Jim
2  Jane
3  Bob
4  Betty
```

PURPOSE:

Use **DIM** to dimension arrays. Arrays are used to store tables of variable information. You must dimension an array before you can use it.

DESCRIPTION:

The simplest version of a DIM statement is:

```
DIM array_name(int_expr)
```

array_name is the name of the array being defined. The array name must meet the rules for variable names. *int_expr* is the high bound for the array—the highest element allowed in a dimension. The low bound is the lowest element allowed in a dimension. The low bound defaults to 1. For example:

```
DIM NAME$(4)
```

This statement defines a one-dimensional array with four elements:

```
NAME$(1)
NAME$(2)
NAME$(3)
NAME$(4)
```

Multiple Dimensions

An array can have up to 32 dimensions. A high bound must be specified for each dimension.

```
DIM array_name(int_expr [, int_expr, ...])
```

For example:

```
10  DIM name$(4,2)
```

This statement defines the following two-dimensional array:

```
NAME$(1,1)   NAME$(1,2)
NAME$(2,1)   NAME$(2,2)
NAME$(3,1)   NAME$(3,2)
NAME$(4,1)   NAME$(4,2)
```

Low Bounds

The low bound is the lowest element a dimension can have. Low bounds can be specified for each dimension of an array. If no low bound is specified, the default is 1. To specify a low bound, use the following format:

```
DIM array_name (int_ expr TO int_expr)
```

The number preceding TO is the low bound. For example:

```
10  DIM name$(4,18 TO 20)
```

This statement creates an array whose first dimension contains elements 1-4 and whose second dimension contains elements 18-20:

```
NAME$(1,18)   NAME$(1,19)   NAME$(1,20)
NAME$(2,18)   NAME$(2,19)   NAME$(2,20)
NAME$(3,18)   NAME$(3,19)   NAME$(3,20)
NAME$(4,18)   NAME$(4,19)   NAME$(4,20)
```

10.1.2 REDIM

FORMAT:

```
REDIM array_name (int_expr, int_expr...) ...

     OR

REDIM array_name [( [int_expr TO] int_expr,
              [int_expr TO] int_expr... )] ...
```

EXAMPLE:

```
10  DIM name$(2)
    INPUT 'How many names': num
    REDIM name$(num)
20  FOR i = 1 TO num
      INPUT 'Enter a name': name$(i)
    NEXT i
30  DO
      PRINT
      FOR i = 1 TO num
        IF  name$(i) = ''  THEN
          PRINT i; ' '; 'empty slot'
        ELSE
          PRINT i; ' '; name$(i)
        END IF
      NEXT i
      PRINT
      INPUT 'How many names': num
      IF  _BACK  or  _EXIT  THEN  EXIT DO
      REDIM name$(num)
    LOOP
40  END
```

```
RNH
How many names? 3
Enter a name? Tim
Enter a name? Sammy
Enter a name? Fred
```

```
 1  Tim
 2  Sammy
 3  Fred

How many names? 4

 1  Tim
 2  Sammy
 3  Fred
 4  empty slot

How many names? exit
```

PURPOSE:

The **REDIM** statement is used to change the size of an array.

DESCRIPTION:

REDIM redimensions arrays. REDIM can only be used on arrays that have already been dimensioned with the DIM statement. The REDIM statement has the same rules, options and limits as the DIM statement.

You can dynamically expand arrays as needed. If you REDIM a single dimension array or the first dimension of a multi dimensioned array to a larger size, the old values are kept. If you REDIM any array to a smaller size or REDIM two or more dimensions in a multi dimensioned array to a larger size, the old values are lost.

If your application depends on REDIM initializing all array values, change your code as follows:

```
Old Code:       REDIM X(100)

New Code:       REDIM X(1)
                REDIM X(100)
```

The REDIM X(1) forces all array values to be initialized by the second REDIM statement.

10.1.3 OPTION BASE

FORMAT:

```
OPTION BASE [0 | 1]
```

EXAMPLE:

```
10  OPTION BASE 0
    DIM name$(4)
20  FOR i = 0 TO 4
      INPUT 'Enter a name': name$(i)
      PRINT i; ' Hello, '; name$(i)
    NEXT i
30  END

RNH
Enter a name? June
 0  Hello, June
Enter a name? Tony
 1  Hello, Tony
Enter a name? Sandy
 2  Hello, Sandy
Enter a name? Carl
 3  Hello, Carl
Enter a name? Liz
 4  Hello, Liz
```

PURPOSE:

Use **OPTION BASE** to set the default low bound for arrays to suit your needs. You have the option of starting the array with element O or element 1.

DESCRIPTION:

When no low bound is specified for a dimension, the default is 1. The OPTION BASE statement lets you specify a default low bound of 0 or 1. When any following DIM or REDIM statements are executed, INTOUCH defaults the low bound to 0 or 1 as specified.

11 Loops, Conditionals, Chaining and Branching

A LOOP is a section of code that can be repeated. There are two types of loops.

- FOR loops execute a block of code a specific number of times.
- DO loops execute a block of code either: infinitely, until a specified condition is met, or while a specified condition remains true.

A FOR loop is used when you need to repeat a block of code a specific number of times. FOR loops are also good when you need to know how many times the loop was executed. You might use a FOR loop if you need to input 10 similar data items. You might use a FOR loop as a counter–say to count from 1 to 1000, or if you need to make calculations from each of the 10 data items entered.

DO loops are used when you need to execute a block of code until a specific condition is met. For instance, you might use a DO loop if you need to enter numbers until a 0 is entered. Or you might use DO loops if you need to do calculations until two numbers match, or if you need to continue a process until either the user chooses to stop or until a final result is reached.

Loops are constructs. They are created by using several statements which can only be used in conjunction with one another (FOR/NEXT, DO/LOOP). The statements which make up the constructs are described together.

11.1 FOR/NEXT Loop

FORMAT:

```
FOR index_var = num_expr1 TO num_expr2 [STEP num_expr3]
        ---
        ---  block of code
        ---
NEXT index_var
```

EXAMPLE:

```
10  DIM name$(4)
20  FOR j = 1 TO 4
      INPUT 'Enter name': name$(j)
      PRINT j; ' '; name$(j)
    NEXT j
30  PRINT 'Finished'
    PRINT 'Final value:'; j
40  END
```

```
RNH
Enter name? Jack
 1  Jack
Enter name? Tom
 2  Tom
Enter name? Sue
 3  Sue
Enter name? Toby
 4  Toby
Finished
Final value: 5
```

PURPOSE:

Use the **FOR** loop to execute a block of code a specific number of times. You can use this construct to repeat a section of code a certain number of times.

DESCRIPTION:

In the above example, the INPUT and PRINT statements make up the body of the loop. This block of code is executed each time the loop is repeated. (For clarity, the body of the loop is indented two spaces.) The **FOR** statement marks the beginning of the loop and determines how many times the loop is repeated. The **NEXT** statement marks the end of the loop.

```
index variable
        |
        V
    FOR J = 1 TO 4   <-- limit expression
            ^
            |
    initial expression
```

General Information

- *index_var* can be an integer or a real numeric variable. **It cannot be an array element**.
- Each FOR statement must have a corresponding NEXT statement or an exception will be generated.
- The index variable can be changed from within the loop. However, this can affect the number of times the loop is repeated.

The **index variable** keeps track of how many times the loop has been executed. The **initial expression** is the number INTOUCH begins counting at. The **limit expression** is the number INTOUCH stops counting at. In the example, INTOUCH counts from 1 to 4, so the loop executes four times.

When INTOUCH runs the example program, it executes the loop four times. The first time the FOR statement is executed, the variable J is set to 1. INTOUCH executes the body of the loop. Since J = 1, INTOUCH inputs NAME$(1), Jack, and prints NAME$(J).

```
            RNH
J = 1       Enter name? Jack
             1  Jack
```

When INTOUCH reaches the NEXT J, it adds one to J and jumps back to the beginning of the loop. J is now 2. INTOUCH checks to see if J is greater than 4. Since J isn't greater than 4, INTOUCH repeats the loop. When INTOUCH executes the loop for the last time, it jumps back to the beginning of the loop and checks to see if J is greater than 4. Since J **is** greater than 4, INTOUCH jumps to the statement following the NEXT J (PRINT 'Finished') and continues normal program execution.

```
            RNH
J = 1       Enter name? Jack
             1  Jack
J = 2       Enter name? Tom
             2  Tom
J = 3       Enter name? Sue
             3  Sue
J = 4       Enter name? Toby
             4  Toby
            Finished
            Final value: 5
```

11.1.1 The STEP Option

By default, when a FOR loop is executed, INTOUCH increments the index variable by one. The increment can be changed with the STEP option. The format of the FOR statement with the STEP option is:

```
FOR index_var = num_expr1 TO num_expr2 STEP num_expr3
    ---
    ---  block of code
    ---
NEXT index_var
```

num_expr3 is a a numeric expression specifying the increment. Each time the FOR statement is executed, INTOUCH adds the increment to the index variable. INTOUCH stops executing the loop when the index variable is greater than the limit.

```
10  DIM name$(4)
20  FOR j = 1 TO 4 STEP 3
      INPUT 'Enter name': name$(j)
      PRINT j; ' '; name$(j)
    NEXT j
30  PRINT 'Finished'
    PRINT 'Final value:'; j
40  END
```

```
RNH
Enter name? Fred
 1  Fred
Enter name? John
 4  John
Finished
Final value: 7
```

11.1.2 Nesting Loops

FOR loops can be nested. A nested loop is a loop which begins and ends inside of another loop. **Loops cannot overlap**. The inner loop must begin and end completely within the outer loop.

```
start of        10  DIM name$(4)
outer loop ---  20  FOR j = 1 TO 4
                      INPUT name$(j)

                      FOR k = 1 TO j
       inner loop    ---   PRINT name$(k); '  ';
                      NEXT k

  end of              PRINT

  outer loop ---    NEXT j

                30  PRINT 'Finished'
                40  END

                RNH
                ? FRED
                FRED
                ? JOHN
                FRED  JOHN
                ? MARY
                FRED  JOHN  MARY
                ? KATE
                FRED  JOHN  MARY  KATE
                Finished
```

11.1.3 EXIT FOR

FORMAT:

```
EXIT FOR
```

EXAMPLE:

```
10  FOR i = 1 TO 5
      INPUT 'Your name, please': name$
      IF  _EXIT  THEN  EXIT FOR
      PRINT 'Hello, '; name$
    NEXT i
20  PRINT 'Finished'
30  END

RNH
Your name, please? James
Hello, James
Your name, please? Marian
Hello, Marian
Your name, please? exit
```

PURPOSE:

Use **EXIT FOR** to exit from a FOR loop.

DESCRIPTION:

When INTOUCH executes an EXIT FOR statement, it jumps to the first statement following the matching NEXT statement. EXIT FOR can only be used within FOR loops. If EXIT FOR is used within a nested loop, INTOUCH exits the innermost loop.

11.1.4 REPEAT FOR

FORMAT:

```
REPEAT FOR
```

EXAMPLE:

```
10  FOR i = 1 TO 3
      PRINT i
      INPUT 'Your name, please': name$
      IF  name$ = ''  THEN  REPEAT FOR
      PRINT 'Hello, '; name$
    NEXT i
20  END
```

```
RNH
 1
Your name, please? George
Hello, George
 2
Your name, please?
 2
Your name, please? Sam
Hello, Sam
 3
Your name, please? Tom
Hello, Tom
```

PURPOSE:

To repeat a FOR loop without incrementing the index variable.

DESCRIPTION:

REPEAT FOR repeats all or part of the body of a loop. REPEAT FOR can only be used in FOR loops. When INTOUCH executes REPEAT FOR, it jumps to the first statement following the FOR statement.

If REPEAT FOR is used within a nested loop, INTOUCH repeats the innermost loop.

```
                    10  FOR i = 1 TO 10
                          FOR j = 1 to 5
INTOUCH will                PRINT j
repeat this                 INPUT 'Your name, please': name$
inner loop --------         IF  name$ = ''  THEN  REPEAT FOR
                            PRINT 'Hello, '; name$
                          NEXT j
                          PRINT 'We now have'; i; 'set(s) of names.'
                        NEXT i
                    20  END
```

11.1.5 ITERATE FOR

FORMAT:

```
ITERATE FOR
```

EXAMPLE:

```
10  FOR i = 1 TO 3
      PRINT i
      INPUT 'Your name, please' : name$
      IF  name$ = 'SKIP'  THEN  ITERATE FOR
      PRINT 'Hello, '; name$
    NEXT i
20  END
```

```
RNH
 1
Your name, please? Toby
Hello, Toby
 2
Your name, please? SKIP
 3
Your name, please? Sam
Hello, Sam
```

PURPOSE:

To skip code processing.

DESCRIPTION:

When INTOUCH executes **ITERATE FOR**, it jumps to the NEXT statement. Any statements between the ITERATE FOR and the NEXT statement will be skipped.

If ITERATE FOR is used in a nested loop, INTOUCH iterates the innermost loop.

```
                  10  FOR i = 1 TO 10
                        FOR j = 1 TO 5
                          PRINT j
                          INPUT 'Your name, please': name$
INTOUCH will              IF name$ = 'SKIP' THEN ITERATE FOR
skip this line -----      PRINT 'Hello, '; name$
                        NEXT j
                        PRINT 'We now have'; i; 'sets of name(s).'
                      NEXT i
                  20  END
```

11.2 DO LOOP

FORMAT:

```
DO [WHILE expr | UNTIL expr]
        ---
        ---  block of code
        ---
LOOP [WHILE expr | UNTIL expr]
```

EXAMPLE:

```
10  a = 3
20  DO UNTIL a = 0
      INPUT 'Your name, please': name$
      PRINT 'Hello, '; name$
      a = a - 1
    LOOP
30  END
```

```
RNH
Your name, please? Sam
Hello, Sam
Your name, please? Sue
Hello, Sue
Your name, please? Bart
Hello, Bart
```

PURPOSE:

Use a **DO LOOP** to execute a block of code repeatedly – until a specified condition is met.

DESCRIPTION:

The simplest type of DO LOOP is an infinite DO LOOP:

```
10  DO
      INPUT 'Your name, please' : name$
      PRINT 'Hello, '; name$
    LOOP
20  END
```

In the above example, the INPUT and PRINT statements make up the body of the loop. This block of code is executed each time the loop is repeated. **DO** begins the loop. **LOOP** marks the end of the loop. When INTOUCH reaches the LOOP statement it jumps back to DO and executes the loop again.

The Ctrl/C command can be used to break out of an infinite DO loop. (For a full description of Ctrl/C, see Interrupting the Program.)

DO loops can be nested. **Loops cannot overlap**. The inner loop must be completely within the DO and LOOP statements of the outer loop.

```
start of
outer loop  --- 10  DO
                      a = 5
                      DO UNTIL a = 0
                   /    INPUT 'Your name' : name$
        inner loop      PRINT 'Hello, '; name$
                   \    a = a - 1
                      LOOP
   end of             PRINT 'Done with a loop'
   outer loop ---   LOOP
                20  END
```

DO loops can be made conditional with WHILE and UNTIL options. WHILE and UNTIL set up a condition. The loop is executed if the condition is met.

11.2.1 WHILE and UNTIL Options

FORMAT:

```
WHILE cond_expr
```

EXAMPLE:

```
10  a = 3
20  DO
      INPUT 'Your name, please': name$
      PRINT 'Hello, '; name$
      a = a - 1
    LOOP WHILE a > 0
30  PRINT 'Finished'
40  END

RNH
Your name, please? FRED
Hello, FRED
Your name, please? JOHN
Hello, JOHN
Your name, please? KATE
Hello, KATE
Finished
```

DESCRIPTION:

cond_expr is a conditional expression. When INTOUCH executes the **WHILE** option, it evaluates the conditional expression as either **TRUE** (1) or **FALSE** (0). If the expression is TRUE, the condition is met and INTOUCH executes the loop. INTOUCH continues executing the loop until the expression becomes FALSE. When the expression becomes FALSE, the condition is not met. INTOUCH stops executing the loop and jumps to the statement following LOOP.

FORMAT:

```
UNTIL cond_expr
```

EXAMPLE:

```
10  a = 3
    DO UNTIL a = 0
      INPUT 'Your name, please': name$
      PRINT 'Hello, '; name$
      a =  a - 1
    LOOP
    PRINT 'Finished'
20  END

RNH
Your name, please? FRED
Hello, FRED
Your name, please? JOHN
Hello, JOHN
Your name, please? KATE
Hello, KATE
Finished
```

DESCRIPTION:

cond_expr is a conditional expression. When INTOUCH executes the **UNTIL** option, it evaluates the conditional expression as either **TRUE** (1) or **FALSE** (0). If the expression is FALSE, INTOUCH executes the loop. INTOUCH continues executing the loop until the expression becomes TRUE. When the expression becomes TRUE, INTOUCH stops executing the loop and jumps to the statement following LOOP.

Placement of WHILE and UNTIL

WHILE and UNTIL can be attached to the DO and/or to the LOOP statements. Whenever INTOUCH encounters a WHILE or UNTIL clause, it checks whether to execute the loop. So, where the WHILE and UNTIL clauses are placed affects the execution of the loop.

If a WHILE or UNTIL is attached to the DO statement, INTOUCH first checks to see whether the condition is TRUE or FALSE before it executes the loop (again). In the case of a WHILE statement, if the condition is still met (i.e. is TRUE), INTOUCH executes the loop. If the condition is not met (i.e. is FALSE or is no longer TRUE), INTOUCH does not execute the loop.

In the case of an UNTIL statement, if the condition has not been met (or is still FALSE), INTOUCH executes the loop once more. If the condition has been met (i.e. is TRUE), INTOUCH does not execute the loop again.

Creating Two Conditions

WHILE and UNTIL options can be placed at both ends of the loop. INTOUCH evaluates each expression in turn. When it finds that one of the conditions has or has not been met (depending upon whether it is a WHILE or UNTIL clause), INTOUCH stops executing the loop. For example, when the following program runs, INTOUCH executes the loop until A equals 5 or the user enters EXIT.

```
10  DIM name$(4)
    a = 1
20  DO UNTIL a = 5
      INPUT 'Your name, please' : name$(a)
      a = a + 1
    LOOP WHILE NOT _EXIT
30  PRINT 'Finished'
40  END
```

11.2.2 EXIT DO

FORMAT:

```
EXIT DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please' : name$
      IF  _EXIT  THEN  EXIT DO
      PRINT 'Hello, '; name$
    LOOP
20  PRINT 'Finished'
30  END
```

```
RNH
Your name, please? Fred
Hello, Fred
Your name, please? exit
Finished
```

PURPOSE:

Use **EXIT DO** to exit from a DO loop.

DESCRIPTION:

When INTOUCH executes an EXIT DO statement, it jumps to the first statement following the LOOP or END DO statement. If EXIT DO is used within a nested loop, INTOUCH exits the innermost loop.

DO...END DO is a single iteration loop. The code between DO and END DO is processed only once unless conditional code specifies exiting or repeating the DO.

11.2.3 REPEAT DO

FORMAT:

```
REPEAT DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please': name$
      IF  _EXIT  THEN  EXIT DO
      IF  name$ = ''  THEN  REPEAT DO
      PRINT 'Hello, '; name$
    LOOP
20  END

RNH
Your name, please? Fred
Hello, Fred
Your name, please?
Your name, please? exit
```

PURPOSE:

Use **REPEAT DO** to repeat part of a DO loop.

DESCRIPTION:

REPEAT DO repeats all or part of the body of a loop. When INTOUCH executes REPEAT DO, it jumps to the first statement following the DO statement.

If REPEAT DO is used within a nested loop, INTOUCH repeats the innermost loop.

```
                        10  DO
                              i = i + 1
                              DO
INTOUCH will repeat             INPUT 'Your name, please': name$
this inner loop --------        IF  _EXIT  THEN  EXIT DO
                                IF  name$ = ''  THEN  REPEAT DO
                                PRINT 'Hello, '; name$
                              LOOP
                              PRINT 'We now have'; i; 'set(s) of names.'
                            LOOP
                        20  END
```

11.2.4 ITERATE DO

FORMAT:

```
ITERATE DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please': name$
      IF  _EXIT  THEN  EXIT DO
      IF  name$ = 'SKIP'  THEN  ITERATE DO
      PRINT 'Hello, '; name$
    LOOP
20  END
```

```
RNH
Your name, please? FRED
Hello, Fred
Your name, please? SKIP
Your name, please? EXIT
```

PURPOSE:

Use **ITERATE DO** to repeat a loop, skipping part of the loop.

DESCRIPTION:

ITERATE DO repeats a loop. When INTOUCH executes ITERATE DO, it jumps to the LOOP or END DO statement. Any statements between the ITERATE DO and the end of the DO block statement will be skipped.

If ITERATE DO is used in a nested loop, INTOUCH iterates the innermost loop.

```
                          10  DO
                                LET i = i + 1
                                DO
INTOUCH will                      INPUT 'Your name, please' : name$
iterate this inner loop ----      IF  name$ = 'SKIP'  THEN  ITERATE DO
                                  IF  _EXIT  THEN  EXIT DO
                                  PRINT 'Hello, '; name$
                                LOOP
                                PRINT 'We now have'; i; 'set(s) of names.'
                              LOOP
                          20  END
```

11.3 EXECUTE

FORMAT:

```
EXECUTE str_expr
```

EXAMPLE:

```
10  INPUT 'Enter a video attribute': video$
20  z$ = 'print ' + video$ + &
        ': "This will be printed using ' + video$ + '"'
30  EXECUTE z$
40  END

RNH
Enter a video attribute? bold
This will be printed using bold
```

EXAMPLE:

```
10  nbr_fields = 5
20  DIM check$(nbr_fields)
30  check$(1) = &
      'DO \' &
      + '  IF  len(ans$) <> 9  THEN \' &
      + '    MESSAGE ERROR : "SSN must be 9 digits" \' &
      + '    EXIT DO \' &
      + '  END IF \' &
      + '  IF  not valid(ans$, "number") THEN  \' &
      + '    MESSAGE ERROR : "SSN must be numeric" \' &
      + '    EXIT DO \' &
      + '  END IF \' &
      + '  PRINT "SSN is valid" \' &
      + 'END DO'
40  field_nbr = 1
50  INPUT 'SSN' : ans$
60  EXECUTE check$(field_nbr)
70  END

RNH
SSN? 123456789
SSN is valid
```

DESCRIPTION:

EXECUTE allows you to incorporate new code into the program at run time. It is mostly used for generalized procedures, utilities, and tools.

A string is built which contains the INTOUCH statements to execute. Multiple INTOUCH statements are separated by either a line feed character (chr$(10) or a "\" character.

When an EXECUTE statement is encountered, INTOUCH compiles the code contained in the string and then runs that code. All program variables are available to the executed code and any variables established by the executed code are available to the rest of the program.

An executed string is only compiled once. When a string has been compiled, the code contained within that string is processed as efficiently as the main program code.

The EXECUTE statement makes the coding of powerful generalized routines very easy.

11.4 Conditionals

Conditionals are constructs which allow you to execute specific blocks of code depending on one or more conditions. For instance, suppose you are doing a tax program. You need to use the EZ form if certain conditions are met, the short form if others are met and the long form otherwise. You can use a conditional to determine which form to use.

There are two types of conditionals, IF and SELECT. The IF construct is useful when you need to check one or more conditions and execute a different block of code depending on the result. For instance, say that you need to print one statement if the user is male and under 20, another if the user is male and between 20 and 40, and still another if the user is male and over 40. The IF construct works well in this kind of situation.

The SELECT CASE construct is useful when you need to compare one main expression with several values and execute a different block of code for each possible match. For instance, suppose that in the tax program we mentioned before, you need to execute a different block of code depending on the tax bracket the user is in. SELECT CASE would let you compare a main expression—BRACKET with all the possible tax brackets (10000-15000, 15000-25000, etc.).

11.5 IF/THEN ... END IF

FORMAT:

```
IF cond_expr THEN statement [ELSE statement]

        or

IF cond_expr1 THEN
    ---
    ---  block of code
    ---
[ELSEIF cond_expr2 THEN
    ---
    ---  block of code
    ---   ...]
[ELSE
    ---
    ---  block of code
    ---      ]
END IF
```

EXAMPLE:

```
10  INPUT PROMPT 'Enter your age: ': age
    INPUT PROMPT 'Enter your sex: ': sex$
20  IF  UCASE$(sex$[1:1]) = 'M'  THEN  GOTO  done  ELSE PRINT
30  IF  age < 20  THEN
      PRINT 'Please go to line A.'
    ELSEIF  age > 19  AND  age < 40  THEN
      PRINT 'Please go to line B.'
    ELSE
      PRINT 'Please go to line C.'
    END IF
40  done: END

RNH
Enter your age: 25
Enter your sex: female

Please go to line B.
```

PURPOSE:

Use the **IF** construct when you want to execute a statement or block of code only under specific conditions.

DESCRIPTION:

The simplest form of the IF construct is a one-line statement:

```
IF cond_expr THEN statement
```

cond_expr is a conditional expression. INTOUCH evaluates this expression as either **TRUE** (1) or **FALSE** (0). If the condition is TRUE, INTOUCH executes the statement following the **THEN**. If the condition is FALSE, INTOUCH skips the statement following the THEN and goes to the next line.

In the example program, when INTOUCH executes the first IF statement, it evaluates the conditional expression, SEX$[1:1] = 'M'. IF the user is 'Male' the condition is TRUE and INTOUCH executes the statement following the THEN and jumps to DONE:.

IF can be used to execute a block of code. The IF block construct looks like this:

```
IF cond_expr THEN
      ---
      ---  block of code
      ---
END IF
```

If the conditional expression is TRUE, INTOUCH executes the block of code beginning on the next line. **END IF** marks the end of this block of code. If the expression is FALSE, INTOUCH skips to the statement following the END IF.

11.5.1 ELSE Option

The **ELSE** option executes a statement if the conditional expression is FALSE. The format of the IF statement with the ELSE option is:

```
IF cond_expr THEN statement ELSE statement
```

When INTOUCH executes the IF statement, it evaluates the conditional expression. If the expression is TRUE, the statement following the THEN is executed. If the expression is FALSE, the ELSE statement is executed. (Please refer to previous example.)

```
RNH
Enter your age: 19
Enter your sex: Female

Please go to line A.
```

In the above program, when INTOUCH executes the first IF statement, it evaluates the expression, SEX$[1:1] = 'M'. Since the user is Female, the expression is FALSE, so INTOUCH skips the THEN clause and jumps to the ELSE clause. INTOUCH executes the code between the ELSE clause and the END IF.

The ELSE option can be used to execute a block of code if the conditional expression is FALSE. The IF construct with the ELSE option looks like this:

```
IF cond_expr THEN
      ---
      ---  block of code
      ---
ELSE
      ---
      ---  block of code
      ---
END IF
```

If the conditional expression is TRUE, INTOUCH executes the block of code between the IF and the ELSE statements. If the expression is FALSE, INTOUCH executes the block of code between the ELSE and the END IF.

```
10  INPUT PROMPT 'Enter your age: ': age
    INPUT PROMPT 'Enter your sex: ': sex$
20  IF  UCASE$(sex$[1:1]) = 'M'  THEN  GOTO the_end
    PRINT
30  IF  age < 40  THEN
      PRINT 'Please go to line A.'
    ELSE
      PRINT 'Please go to line B.'
    END IF
40  the_end: END

RNH
Enter your age: 45
Enter your sex: female

Please go to line B.
```

In the above program, when INTOUCH executes the second IF statement, it checks to see if "AGE < 40". Since AGE is not less than 40, the condition is FALSE. INTOUCH skips the code following the THEN and jumps to the ELSE clause. INTOUCH executes the code following the ELSE clause.

11.5.2 ELSEIF Option

The **ELSEIF** option sets up a new condition to check. The format of the IF construct with the ELSEIF option is:

```
IF cond_expr1 THEN
      ---
      ---  block of code
      ---
ELSEIF cond_expr2 THEN
      ---
      ---  block of code
      ---
ELSE
      ---
      ---  block of code
      ---
END IF
```

ELSEIF establishes a new condition. INTOUCH evaluates this condition. If the condition is TRUE (1), INTOUCH executes the code following the ELSEIF. If the condition is FALSE (0), INTOUCH jumps to the next clause in the IF construct.

```
10  INPUT PROMPT 'Enter your age: ': age
    INPUT PROMPT 'Enter your sex: ': sex$
20  IF  UCASE$(sex$[1:1]) = 'M'  THEN  GOTO the_end ELSE PRINT
30  IF  age < 20  THEN
      PRINT 'Please go to line A.'
    ELSEIF  age > 19  AND  age < 40  THEN
      PRINT 'Please go to line B.'
    ELSE
      PRINT 'Please go to line C.'
    END IF
40  the_end: END

RNH
Enter your age: 25
Enter your sex: Female

Please go to line B.
```

In the above program when INTOUCH executes the second IF statement, it checks to see if "AGE < 20". Since AGE is not less than 20, the first condition is FALSE. INTOUCH skips the code following the THEN and checks the condition set up by the ELSEIF. Since "AGE > 19" and "AGE < 40", the second condition is TRUE and INTOUCH executes the code following the ELSEIF and prints 'Please go to line B.', then exits the conditional.

11.6 SELECT CASE/END SELECT

FORMAT:

```
SELECT CASE main_expr
CASE expr1[, expr2,...]
    ---
    ---  block of code
    ---
[CASE expr3[, expr4,...]
    ---
    ---  block of code
    ---             ...]
[CASE IS {numeric operator | boolean operator} expr5
    ---
    ---  block of code
    ---             ...]
[CASE ELSE
    ---
    ---  block of code
    ---             ...]
END SELECT
```

EXAMPLE:

```
10  DO
      INPUT 'Your income per year': income
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
20    SELECT CASE income
      CASE 0
        PRINT 'No income?'
      CASE IS < 0
        PRINT 'A negative income?  You are in debt!'
      CASE IS > 0
        PRINT 'A positive income.'
      END SELECT
    LOOP
30  END
```

```
RNH
Your income per year? 0
No income?
Your income per year? -15000
A negative income? You are in debt!
Your income per year? 30000
A positive income.
Your income per year? exit
```

PURPOSE:

Sometimes you need to compare one main expression with several values and execute a different block of code for each possible match. Use **SELECT CASE** to check a set of conditions and execute code depending on the results.

DESCRIPTION:

The SELECT CASE statement begins the construct and gives the main expression (*main_expr*). In the example, the main expression is INCOME. The **CASE** statements are compared with the main expression. The first CASE expression (*expr1*) is 0. Following this CASE is a block of code. If INCOME = 0 the block of code following CASE 0 is executed.

- The main expression and all CASE expressions must be the same data type. For example, if the main expression is a string expression, all the CASE expressions must be strings also. If the data types do not match, an exception is generated.
- If the main expression is an integer and a real numeric CASE expression is given, the CASE expression is rounded and the integer portion is compared.
- There must be at least one CASE expression.
- If none of the CASE expressions match (and there is no CASE ELSE), an exception is generated.

11.6.1 CASE expr, expr, expr...

Each **CASE** statement can include several expressions separated by commas. INTOUCH compares each of the expressions in a CASE statement. If any of them match, the block of code following the CASE is executed.

```
10  DO
      INPUT 'Procedure (add, del, exit)': pro$
      IF  _EXIT  THEN  EXIT DO
      pro$ = UCASE$(pro$)
20    SELECT CASE pro$
      CASE 'ADD'
        PRINT 'Adding...'
      CASE 'DEL', 'DELETE'
        PRINT 'Deleting...'
      END SELECT
    LOOP
30  END

RNH
Procedure (add, del, exit)? ADD
Adding...
Procedure (add, del, exit)? DEL
Deleting...
Procedure (add, del, exit)? EXIT
```

The following example illustrates how to check for a range of values:

```
10  a = 5
    SELECT CASE a
      CASE 1 : PRINT 'one'
      CASE 2 to 6 : PRINT 'range'
      CASE ELSE : PRINT 'else'
    END SELECT
20  b$ = 'c'
    SELECT CASE b$
      CASE 'a' : PRINT 'a'
      CASE 'b' to 'e' : PRINT 'range'
      CASE ELSE : PRINT 'else'
    END SELECT
30  END

RNH
range
range
```

11.6.2 CASE ELSE Option

The **CASE ELSE** option executes a block of code only if none of the CASE expressions match. SELECT CASE with the CASE ELSE option looks like this:

```
SELECT CASE main expr
[CASE expr1, expr2...
    ---
    ---  block of code
    ---             ...]
[CASE IS {numeric operator| boolean operator} expr3
    ---
    ---  block of code
    ---             ...]
CASE ELSE
    ---
    ---  block of code
    ---
END SELECT
```

CASE ELSE must follow the last CASE statement. If none of the CASE expressions match, INTOUCH executes the block of code following the CASE ELSE statement.

```
10  DO
      INPUT 'Procedure (add, del, exit)' : pro$
      IF  _EXIT  THEN  EXIT DO
      pro$ = UCASE$(pro$)
20    SELECT CASE pro$
      CASE 'ADD'
        PRINT 'Adding...'
      CASE 'DEL', 'DELETE'
        PRINT 'Deleting...'
      CASE ELSE
        MESSAGE ERROR: 'Procedure must be: add, del or exit'
        REPEAT DO
      END SELECT
    LOOP
30  END

RNH
Procedure (add, del, exit)? add
Adding...
Procedure (add, del, exit)? del
Deleting...
Procedure (add, del, exit)? funny

                Procedure must be add, del, or exit
```

```
Procedure (add, del, exit)? EXIT
```

11.6.3 CASE IS Option

CASE IS lets you form a conditional expression to be checked against the main expression. The format of the CASE IS option is:

```
CASE IS {relational operator} expr
```

When the CASE IS statement executes, INTOUCH compares *expr* to the *main_expr* using the relational operator.

```
10  DO
      INPUT 'Your income per year': income
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
20    SELECT CASE income
      CASE 0
        PRINT 'No income?'
      CASE IS < 0
        PRINT 'A negative income?  You are in debt!'
      CASE IS > 0
        PRINT 'A positive income.'
      END SELECT
    LOOP
30  END
```

```
RNH
Your income per year? -15000
A negative income?  You are in debt!
Your income per year? 0
No income?
Your income per year? 25000
A positive income.
Your income per year? exit
```

11.7 CHAIN Programs

FORMAT:

```
CHAIN 'file_spec'
```

EXAMPLE:

```
10  LINE INPUT 'Your name (last, first)': name$
    OPEN #1: NAME 'storage.tmp', ACCESS OUTPUT
    PRINT #1: name$
    CLOSE #1
20  INPUT 'Add to CLIENT structure (Y/N)': reply$
    IF  reply$ = 'Y'  THEN  CHAIN 'ADD'
30  END
```

```
RNH
Your name (last, first)? Woods, Jack
Add to CLIENT structure (Y/N)? N
```

DESCRIPTION:

CHAIN exits the current program and runs the program specified.

file_spec is the specification for the program being chained to. The file specification can be any string expression. INTOUCH searches for the file specified. INTOUCH exits the current program and runs the program named. Control **does not** return to the current program when the chained program is finished. If INTOUCH cannot find the file, or if the file is not an executable program, an exception is generated.

When INTOUCH executes the CHAIN statement, it:

- writes all active output buffers, closes all files in the current program and releases all storage.
- exits the current program and executes the program named.
- does not pass any variables, functions, etc., from the previous program.

11.8 Pass Commands to the Operating System

11.8.1 PASS [NOWAIT: | NORETURN:]

FORMAT:

```
PASS [NOWAIT: | NORETURN:] str_expr
```

EXAMPLE:

```
10  INPUT 'What would you like to show': info$
20  PASS 'SHOW ' + info$
30  END

RNH
What would you like to show? user t

      VAX/VMS User Processes at  9-MAY-1995 10:22:29.87
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER       TTI            1
```

PURPOSE:

Use **PASS** to perform system commands without leaving the INTOUCH environment or exiting a program.

DESCRIPTION:

PASS passes the specified string expression to the operating system command interpreter. Generally, it passes the string to DCL. The operating system will respond to the string as it would if you entered it at the DCL $ prompt. When the system finishes, control returns to the INTOUCH program.

When you use the **NOWAIT** option with PASS, the operating system executes the passed command and immediately returns to INTOUCH without waiting for the passed command to finish.

EXAMPLE:

```
10  PASS NOWAIT: 'show user t'
20  END

INTOUCH

RNH

INTOUCH

      VAX/VMS User Processes at  9-MAY-1995 10:23:19.87
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER       TTI            1

end

INTOUCH
```

The **PASS NORETURN** statement passes a command to the operating system but does not return to INTOUCH.

EXAMPLE:

```
10  PRINT 'B E F O R E'
20  DELAY 2
30  PASS NORETURN: 'show user t'
40  END

RNH
B E F O R E

      VAX/VMS User Processes at  9-MAY-1995 01:43:14.36
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER      TTI            1
$
```

11.9 Branching

Branching means jumping from one part of a program to another. Branching is especially useful when combined with conditionals. For example, you can ask the user to choose a procedure. You can branch to different subroutines in your program, depending on the procedure chosen.

There are several statements used for branching: GOTO, GOSUB, DISPATCH, ON GOTO, and ON GOSUB. GOTO causes INTOUCH to jump to the specified location and continue normal program execution from there.

GOSUB and DISPATCH execute routines and subroutines. Subroutines are blocks of code that INTOUCH branches to, executes, and then returns from.

GOTO, GOSUB, and DISPATCH cause unconditional branching. When they are executed, INTOUCH jumps to the statement specified. To conditionally branch, combine these statements with conditional statements. Two other constructs, ON GOTO and ON GOSUB, are conditional. Both ON GOTO and ON GOSUB branch to one of a number of places in the program depending an integer expression. When the ON GOSUB construct is used, INTOUCH will return to the statement following the ON GOSUB when the subroutine has finished executing.

11.9.1 Branching and Code Blocks

Certain constructs (such as GOSUB) include a block of code which is executed within the construct. You can transfer control within these code blocks and out of these code blocks. For example:

```
                    10  DIM name$(4)
                    20  FOR i = 1 TO 4
                          INPUT 'Your name, please': name$(i)
branches within block --    IF  name$(i) <> ''  THEN  GOTO hello
branches out of block --    IF  _EXIT  THEN  EXIT FOR
                          hello:
                            PRINT 'Hello!'
                    30  NEXT i
                    40  PRINT 'Finished'
                    50  END
```

11.10 GOTO

FORMAT:

```
GOTO target
```

EXAMPLE:

```
10  start:
    INPUT 'Your Name': name$
    IF  _EXIT  THEN  GOTO the_end
    PRINT 'Hello, '; name$
    GOTO start
20 the_end: END
```

```
RNH
Your Name? Tony
Hello, Tony
Your Name? Betty
Hello, Betty
Your Name? exit
```

PURPOSE:

Use **GOTO** when you want to jump from one part of your program to another.

DESCRIPTION:

target is the line to which control is being transferred. *target* can be a label or line number.

Note

It is recommended that you branch to labels rather than line numbers. Should the line numbers change, the change would have no effect on your branching code.

When INTOUCH executes a GOTO statement, it branches to the *target* line specified. Any code between the GOTO and the *target* line is skipped. INTOUCH continues normal program execution at the target line.

11.10.1 ON...GOTO...[ELSE]

FORMAT:

```
ON int_expr GOTO target1 [, target2...] [ELSE statement]
```

EXAMPLE:

```
10  ask: INPUT 'Procedure (1=add, 2=del, 3=exit)': pro
20  ON pro GOTO add, del, done
    add:
      PRINT 'Adding...'
      GOTO ask
    del:
      PRINT 'Deleting...'
      GOTO ask
30  done: END
```

```
RNH
Procedure (1=add, 2=del, 3=exit)? 2
Deleting...
Procedure (1=add, 2=del, 3=exit)? 1
Adding...
Procedure (1=add, 2=del, 3=exit)? 3
```

PURPOSE:

Sometimes you want to jump to one of several places in your program. **ON...GOTO** lets you jump to one of several locations depending on a condition you set up.

DESCRIPTION:

int_expr is the integer expression or condition to check. *target1, target2 ...* is a list of places to jump to. Each target can be a label or line number.

ON..GOTO branches to one of several targets depending on the value of an integer expression. The simplest version of ON..GOTO is,

```
ON int_expr GOTO target1, target2...
```

int_expr is an integer expression whose value determines where control is transferred. When INTOUCH executes the ON GOTO statement, it evaluates the integer expression. If the value is 1, INTOUCH branches to the first target in the list. If the value is 2, INTOUCH branches to the second target, and so on.

If the expression does not evaluate to an integer, INTOUCH rounds it. The value must be from 1 to the number of targets (unless there is an ELSE clause). For example, if there are five targets, the integer value must be from 1 to 5.

If the value is less than 1 or greater than the number of targets, and there is no ELSE clause, INTOUCH generates an exception.

Targets can be labels or line numbers.

INTOUCH supports 128 targets for ON...GOTO. More targets than 128 gives an "expression too complex" exception.

11.10.1.1 ELSE Option

The **ELSE** option executes a statement if the integer expression does not find a target. ON GOTO with the ELSE option looks like this:

```
ON int_expr GOTO target1 [, target2...] ELSE statement
```

ELSE must be followed by a *statement*. The ELSE is executed if the integer value exceeds the number of targets in the list, is zero or is negative.

```
10  start:
    INPUT 'Procedure (1=add, 2=del, 3=exit)': pro
20  ON pro GOTO add, del, done ELSE PRINT 'Enter 1, 2 or 3'
    GOTO start
    add:
      PRINT 'Adding...'
      GOTO start
    del:
      PRINT 'Deleting...'
      GOTO start
30  done:
    PRINT 'Finished'
    END

RNH
Procedure (1=add, 2=del, 3=exit)? -2
Enter 1, 2 or 3
Procedure (1=add, 2=del, 3=exit)? 1
Adding...
Procedure (1=add, 2=del, 3=exit)? 3
Finished
```

11.11 GOSUB/RETURN

11.11.1 GOSUB

FORMAT:

```
GOSUB target
       .
       .
       .
target
    ---
    ---  block of code
    ---
RETURN
```

EXAMPLE:

```
10  GOSUB get_input
20  PRINT 'Hello, '; name$
    GOTO exit
30  get_input:
    INPUT 'Enter your name': name$
    name$ = UCASE$(name$)
    RETURN
40  exit:
    PRINT 'Finished'
50  END

RNH
Enter your name? Julian
Hello, JULIAN
Finished
```

PURPOSE:

Use **GOSUB** to jump to another location in your program, execute the code at that location and then return to the place you left.

DESCRIPTION:

GOSUB transfers control to a subroutine. The subroutine is a block of code. INTOUCH executes the code, then returns to the statement following the GOSUB. The target can be a label, a ROUTINE statement or a line number.

Note

It is recommended that you branch to labels rather than line numbers. Should the line numbers change, the change would have no effect on your branching code.

If the *target* in the GOSUB statement does not exist, an exception is generated. RETURN or END ROUTINE ends the subroutine and transfers control to the statement following the GOSUB.

You can nest 64 GOSUBs up to 64 levels deep.

Note

If your label or routine name contains an underscore (_) character (e.g. ask_name, do_add), you do NOT need to use the word "GOSUB". Under these conditions, the "GOSUB" is implied. For example: using ASK_NAME would be the same as using GOSUB ASK_NAME

11.11.2 ON...GOSUB

FORMAT:

```
ON int_expr GOSUB target1 [, target2...] [ELSE statement]
```

EXAMPLE:

```
10  start: INPUT 'Procedure (1=add, 2=del, 3=exit)': pro
20  ON pro GOSUB add, del, done ELSE PRINT 'Enter 1, 2 or 3'
    GOTO start
    add:
      PRINT 'Adding...'
      RETURN
    del:
      PRINT 'Deleting...'
      RETURN
30  done:
    PRINT 'Finished'
    END
```

```
RNH
Procedure (1=add, 2=del, 3=exit)? add
Non-numeric input when number expected at START
Procedure (1=add, 2=del, 3=exit)? 5
Enter 1, 2 or 3
Procedure (1=add, 2=del, 3=exit)? 1
Adding...
Procedure (1=add, 2=del, 3=exit)? 3
Finished
```

PURPOSE:

Use **ON..GOSUB** when you want to jump to one of several subroutines depending on a condition you set up. ON..GOSUB jumps to one of the locations, executes the code at that location and then returns to the first statement after the ON..GOSUB.

DESCRIPTION:

ON..GOSUB transfers control to one of several subroutines depending on the value of an integer expression. The simplest version of ON..GOSUB is:

```
ON int_expr GOSUB target1, target2...
```

int_expr is an integer expression whose value determines where control is transferred. The targets begin subroutines. They can be ROUTINE statements, labels or line numbers. The subroutines are blocks of code. When INTOUCH branches to a subroutine, it executes the block of code until it reaches a RETURN or END ROUTINE statement. INTOUCH then returns to the statement following the ON..GOSUB.

When INTOUCH executes the ON GOSUB, it evaluates the integer expression. If the value is one, INTOUCH branches to the first target and executes the subroutine. If the integer is two, INTOUCH branches to the second subroutine, and so on.

int_expr must evaluate to an integer. If it does not, INTOUCH rounds the value and uses the resulting integer. The integer expression must yield an integer within the range of the target list. For example, if there are five targets in the list, the integer value must be in the range of 1 to 5. If the integer value is not in the range of the target list, and there is no ELSE clause, an exception is generated.

INTOUCH supports 128 targets for ON..GOSUB. More targets than 128 gives an "expression too complex" exception.

11.11.2.1 ELSE Option

The **ELSE** option executes a statement if the integer value does not yield a target. ON GOSUB with the ELSE option looks like this:

```
ON int_expr GOSUB target1, target2... ELSE statement
```

ELSE must be followed by a *statement*. The ELSE is executed if the integer value exceeds the number of targets in the list, is 0 or is negative.

The previous example shows how ELSE is used.

11.12 DISPATCH

FORMAT:

```
DISPATCH str_expr
     .
     .
     .
target
     ---
     ---  block of code
     ---
RETURN
```

EXAMPLE:

```
10  INPUT 'Routine name', DEFAULT 'add': routine$
    DISPATCH routine$
    STOP
20  add:
      PRINT 'Adding information...'
    RETURN

30  change:
      PRINT 'Changing information...'
    RETURN

40  END

RNH
Routine name? add
Adding information...
```

PURPOSE:

DISPATCH executes a routine that the program determines at run-time.

DESCRIPTION:

DISPATCH looks at the contents of the string expression (*str_expr*), searches for a routine with that name and GOSUBS to the routine.

str_expr is the name of the subroutine to execute.

11.13 ROUTINE/END ROUTINE

FORMAT:

```
ROUTINE routine_name
   ---
   --- block of code
   --- [REPEAT ROUTINE]
   --- [EXIT ROUTINE]
END ROUTINE
```

EXAMPLE:

```
10      get_username
20      END

12000   ROUTINE get_username
          INPUT PROMPT 'Username: ': uname$
          IF  _BACK  or  _EXIT  THEN  EXIT ROUTINE
        END ROUTINE

RNH
Username: Tester
```

DESCRIPTION:

ROUTINES are block-structured subroutines. They provide a convenient way to name a block of code. Routines can be edited as a complete unit, or partially edited by line number.

To execute a ROUTINE, use the GOSUB or DISPATCH statement or just the name of the routine. You CANNOT fall into a routine the way you can a subroutine that starts with a label.

11.13.1 EXIT ROUTINE

FORMAT:

```
EXIT ROUTINE
```

DESCRIPTION:

The **EXIT ROUTINE** statement enables you to exit from the executed routine.

The above example shows how EXIT ROUTINE can be used.

11.13.2 REPEAT ROUTINE

FORMAT:

```
REPEAT ROUTINE
```

EXAMPLE:

```
10      get_username
20      END

12000   ROUTINE get_username
          INPUT PROMPT 'Username: ': uname$
          IF  _BACK  or  _EXIT  THEN  EXIT ROUTINE
          IF  uname$ = ''  THEN  REPEAT ROUTINE
        END ROUTINE

RNH
Username:
Username: Sunny
```

DESCRIPTION:

The **REPEAT ROUTINE** statement enables you to repeat the entire routine execution. When INTOUCH executes the REPEAT ROUTINE statement, the routine is re-executed.

12 Exception Handling

Exception handling routines intercept run-time exceptions and execute a block of code which handles them. If you do not have an exception handler, INTOUCH returns an exception message specifying what the exception was and where it occurred. INTOUCH stops program execution or tries the offending statement again.

There are two types of exception handlers: attached handlers and detached handlers.

When you have an **attached handler**, you use a WHEN EXCEPTION IN statement. WHEN EXCEPTION IN puts the handler (the block of code to execute) right after the block of code it protects.

You can also have a **detached handler**. For a detached handler, use the statement WHEN EXCEPTION USE. When an exception occurs in the protected block, WHEN EXCEPTION USE calls a handler routine located in some other part of the program. The same handler routine can be used by any number of WHEN EXCEPTION USE statements and can be placed anywhere in your program.

This section explains exception handlers. It also describes the handling statements–RETRY, CONTINUE and EXIT HANDLER.

The following functions are also related to exception handling:

- EXLINE
- EXTYPE
- EXLABEL$
- EXTEXT$
- SYSTEXT$

For a full description of these functions, see Section A.2, Other Functions.

12.1 CAUSE EXCEPTION

FORMAT:

```
CAUSE EXCEPTION exception_number
```

EXAMPLE:

```
10  DO
      INPUT 'Select a number between 1 and 10': no
      IF  no < 1  OR  no > 10  THEN  CAUSE EXCEPTION 1001
      REPEAT DO
    END DO
20  END

RNH
Select a number between 1 and 10? 8
Select a number between 1 and 10? 99
Illegal number at 10.2
```

PURPOSE:

Use **CAUSE EXCEPTION** when you need to generate an exception under specific conditions.

DESCRIPTION:

CAUSE EXCEPTION causes the specified exception to occur. *exception_number* can be any integer expression. When INTOUCH executes the CAUSE EXCEPTION statement, it generates the exception specified. See Section C.2, Exceptions for a list of exception messages.

12.2 WHEN EXCEPTION IN

FORMAT:

```
WHEN EXCEPTION IN
  ---
  ---  protected block
  ---
USE
  ---
  ---  handler
  ---
END WHEN
```

EXAMPLE:

```
10  INPUT 'Your name, please': name$
    WHEN EXCEPTION IN
      INPUT 'How old are you': age
    USE
      PRINT 'Not a valid age'
      RETRY
    END WHEN
    PRINT
    PRINT NAME$; ' is'; age
20  END

RNH
Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

PURPOSE:

Use WHEN EXCEPTION IN to catch run-time exceptions and resolve them within your program when you want the code, handling the exception, to be next to the code you are protecting.

DESCRIPTION:

WHEN EXCEPTION IN begins the protected block of code. Everything between the WHEN EXCEPTION IN and **USE** statements constitutes the section of code protected by the handler—the *protected block*.

USE begins the handler routine. Everything between the USE and the **END WHEN** statements constitutes the handler. If an exception occurs in the protected block, the handler code is executed. END WHEN ends the routine.

12.3 WHEN EXCEPTION USE

FORMAT:

```
WHEN EXCEPTION USE handl_name
  ---
  ---  protected block
  ---
END WHEN
         .
         .
         .
HANDLER handl_name
  ---
  ---  handler
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
    WHEN EXCEPTION USE fix_average
      average = tsales/nsales
    END WHEN
    PRINT 'The average is:'; average

20  HANDLER fix_average
      average = 0
      CONTINUE
30  END HANDLER
40  END

RNH
Enter total sales amount? 25.00
Enter number of sales? 0
The average is: 0
```

PURPOSE:

Use WHEN EXCEPTION USE to catch run-time exceptions and resolve them within a program when you need the same handler for several protected blocks, or when you want all the handlers in one place in your program.

DESCRIPTION:

WHEN EXCEPTION USE begins the protected block of code and specifies the **HANDLER** routine to use. *handl_name* is the name of the handler routine. The handler name must meet the specifications for variable names. The protected block is everything between the WHEN EXCEPTION USE and the **END WHEN** statements. If an exception occurs in this block of code, INTOUCH calls the handler specified. If the handler does not exist, an error is generated.

HANDLER begins the handler routine. Everything between the HANDLER and the **END HANDLER** constitutes the handler routine. END HANDLER returns control to the statement following the END WHEN. When the handler is called, this block of code is executed. In the example above, INTOUCH would normally return an exception when it tried to divide 25.00 by 0. The exception handler FIX_AVERAGE intercepts this exception and sets AVERAGE equal to 0.

The handler routine can occur before or after the protected block. For example:

```
10  HANDLER fix_average
      average = 0               <--- handler routine
      CONTINUE
    END HANDLER

20  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
    WHEN EXCEPTION USE fix_average
      average = tsales/nsales
    END WHEN
30  PRINT 'The average is'; average
40  END
```

One of the advantages of WHEN EXCEPTION USE is that the same handler routine can be called by any number of WHEN EXCEPTION USE statements. For example:

```
10  WHEN EXCEPTION USE numbers
      INPUT 'How old are you': age      <--- first protected block
    END WHEN

20  INPUT 'Your name, please': name$

30  WHEN EXCEPTION USE numbers
      INPUT 'Your birthdate': birth     <--- second protected block
    END WHEN

40  PRINT name$; ' was born on'; birth

50  HANDLER numbers
      PRINT 'Enter numbers only, please.'    <--- handler routine
      RETRY
    END HANDLER
60  END
```

12.4 HANDLER Routine Actions

12.4.1 RETRY

FORMAT:

```
USE
  ---
  ---  RETRY
  ---
END WHEN

   or

HANDLER handl_name
  ---
  ---  RETRY
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Your name, please': name$
20  WHEN EXCEPTION IN
      INPUT 'How old are you': age
    USE
      PRINT 'Not a valid age'
      RETRY
    END WHEN
30  PRINT
    PRINT name$; ' is'; age
40  END

RNH
Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

PURPOSE:

Use RETRY after an exception to re-execute the statement that generated the exception.

DESCRIPTION:

RETRY can *only* be used in a HANDLER routine. RETRY causes INTOUCH to leave the handler and re-execute the statement that generated an exception.

12.4.2 CONTINUE

FORMAT:

```
USE
  ---
  ---  CONTINUE
  ---
END WHEN

   or

HANDLER handl_name
  ---
  ---  CONTINUE
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
20  WHEN EXCEPTION USE fix_average
      average = tsales / nsales
    END WHEN
30  PRINT 'The average is:'; average

40  HANDLER fix_average
      average = 0
      CONTINUE
    END HANDLER
50  END

RNH
Enter total sales amount? 18.00
Enter number of sales? 0
The average is: 0
```

PURPOSE

Use CONTINUE to continue normal program execution at the statement following the one that generated the exception.

DESCRIPTION:

CONTINUE causes INTOUCH to exit the exception handler and continue program execution at the first statement following the statement which generated the exception. CONTINUE can *only* be used in a HANDLER routine.

12.4.3 RESUME

FORMAT:

```
USE
  ---
  ---  RESUME target
  ---
END WHEN

   or

HANDLER handl_name
  ---
  ---  RESUME target
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
20  WHEN EXCEPTION USE fix_average
      average = tsales / nsales
    END WHEN
30  PRINT 'The average is:'; average

40  HANDLER fix_average
      average = 0
      PRINT 'Invalid numbers.  Try again.'
      RESUME 10
    END HANDLER
50  END

RNH
Enter total sales amount? 75.00
Enter number of sales? 0
Invalid numbers.  Try again.
Enter total sales amount? 75.00
Enter number of sales? 3
The average is: 25
```

PURPOSE:

Use RESUME to resume normal program execution at the statement label that you specify.

DESCRIPTION:

RESUME causes INTOUCH to exit the exception handler and resume program execution at the target line specified. The target can be a line number or a label.

RESUME can *only* be used in a handler routine.

12.4.4 EXIT HANDLER

FORMAT:

```
USE
  ---
  ---  EXIT HANDLER
  ---
END WHEN

   or

HANDLER handl_name
  ---
  ---  EXIT HANDLER
  ---
END HANDLER
```

EXAMPLE:

```
10  WHEN EXCEPTION USE mistake
      INPUT 'Enter your age': age
    END WHEN
    PRINT 'You are'; age; 'years old'

20  HANDLER mistake
      PRINT 'Oops...'
      DELAY 2
      EXIT HANDLER
    END HANDLER
30  END

RNH
Enter your age? 3x
Oops...
Non-numeric input when number expected at 10.1
Enter your age? 35
You are 35 years old
```

PURPOSE:

Use EXIT HANDLER to exit a handler routine and pass the exception up to the next level of exception handling.

DESCRIPTION:

EXIT HANDLER exits the current HANDLER routine. If the current handler is nested within another handler, INTOUCH jumps to the outside handler and executes the code for that handler. If there is no other exception handler, control returns to the INTOUCH system. INTOUCH prints the exception message and takes whatever action it normally would.

EXIT HANDLER can *only* be used within an exception handler.

13 Calling Routines Written In Other Languages

You can call routines written in other languages and run them from an INTOUCH program.

Callable routines are stored in libraries. The LIBRARY statement tells INTOUCH what library a routine is located in. You must use the LIBRARY statement to specify where routines are located before you call them in your program.

The CALL statement calls routines and executes them. You can call any routine in a shared VMS library and execute it. The CALL and LIBRARY statements make your programs more powerful, more versatile, and give you more programming options.

13.1 LIBRARY

FORMAT:

```
LIBRARY 'libr_name'
```

EXAMPLE:

```
10  LIBRARY 'BASRTL'
20  PRINT 'Waiting 5 seconds...'
30  CALL BAS$SLEEP(5% BY VALUE)
40  END

RNH
Waiting 5 seconds...
```

DESCRIPTION:

You can use the **LIBRARY** statement to specify the libraries you will use in your program.

The LIBRARY statement specifies libraries from which you CALL routines. The routines can be written in any VMS language that supports the standard calling interface (FORTRAN, BASIC, COBOL, etc.).

libr_name is the file specification of a library. The library can be one of the VMS supplied libraries, or a user-defined library. You must name a library with the LIBRARY statement before you call a routine from it. The library name must be a string constant in quotes.

To create libraries for INTOUCH, you can refer to Appendix E, Creating Libraries for Use with INTOUCH.

13.2 CALL

FORMAT:

```
CALL routine_name(arg [BY pass_mech], arg...)
```

EXAMPLE:

```
10  LIBRARY 'BASRTL'
20  PRINT 'Waiting 5 seconds...'
30  CALL BAS$SLEEP(5% BY VALUE)
40  END

RNH
Waiting 5 seconds...
```

DESCRIPTION:

You can use the **CALL** statement to call and execute library routines. You can use these routines to perform procedures rather than writing the code yourself.

The library to which the routine belongs must have been specified in a LIBRARY statement.

routine_name is the name of the routine you are calling. Some routines take arguments. *arg* is an argument you are passing to or getting from the routine. If more than one argument is passed, separate the arguments with commas:

```
CALL routine_name(arg, arg, arg...)
```

13.2.1 Optional Arguments

Some routines have optional arguments–arguments which **can** be used, but do not have to be used. To pass an optional argument, include it in the appropriate place in the argument list. If you do not want to pass an optional argument, specify that no argument is to be used by typing two commas side by side at the appropriate place in the argument list.

For example, the routine LIB$GET_INPUT gets a record of ASCII text. Its argument list contains two optional arguments:

```
LIB$GET_INPUT(get-str.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
                  |                 |                  |
    string variable the      a prompt string        length of the
    input is assigned to                           string variable
```

To input a line without using a prompt string, your CALL statement would have the following format:

```
CALL routine_name(str_var,,num_var)
```

The double commas tell INTOUCH that the optional variable *prompt-str* is not used. For example:

```
10  LIBRARY 'LIBRTL'
    BUF$ = REPEAT$(' ', 200)
    CALL LIB$GET_INPUT(BUF$, , OUT_LEN%)
    PRINT 'You typed: ';BUF$[1:OUT_LEN%]
20  END
```

13.2.2 Passing Mechanisms

pass_mech refers to the mechanism by which values are passed to the routine. The default passing mechanism for reals and integers is by **reference**. The default passing mechanism for strings is by **descriptor**. Here is an explanation of the passing mechanisms available:

BY REF By reference. This is the default passing mechanism for real and integer data. Arguments passed by reference can be changed by the routine they are passed to.

BY VALUE By value. Arguments passed by value cannot be changed by the routine they are passed to.

BY DESC By descriptor. This is the default passing mechanism for strings. By descriptor causes INTOUCH to use the argument's descriptor when passing the argument. Arguments passed by descriptor can be changed by the routine they are passed to.

The system functions _REAL and _INTEGER can be used in conjunction with the library statements. These functions return resulting data associated with the CALL. (See Section A.1, System Functions for more information.)

14 Data Structure Statements

This chapter explains the INTOUCH data structure statements. One of the major features of INTOUCH is its ability to perform database operations as part of the language. The data structure statements allow you to manipulate data structures in your own programs.

List of Data Structure Statements:

OPEN STRUCTURE
CLOSE STRUCTURE
EXTRACT STRUCTURE
REEXTRACT STRUCTURE
CANCEL EXTRACT
ADD STRUCTURE
DELETE STRUCTURE
INCLUDE
EXCLUDE
SORT
FOR EACH
ASK STRUCTURE
SET STRUCTURE
LOCK STRUCTURE
UNLOCK STRUCTURE

14.1 General Information

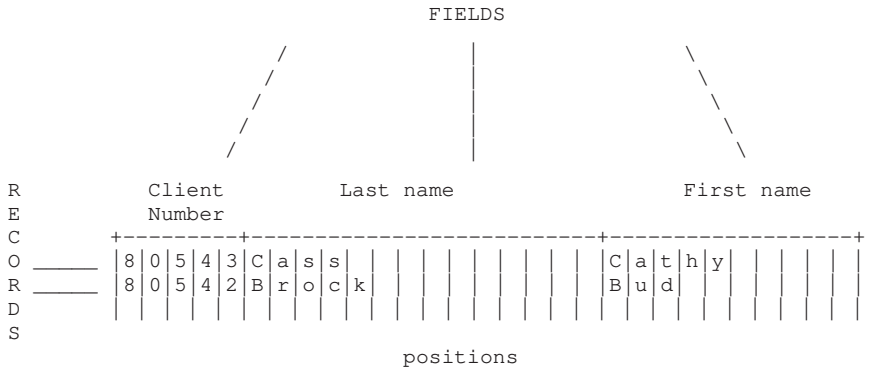
INTOUCH's data management system stores data file information in **structures**. (Chapter 16, Creating Structures, Field Definitions with SETUP tells how structures are created.) A structure file contains the following information:

- name of the database engine (RMS, Rdb, Oracle, etc.)
- dataset name (data file name)
- data dictionary name (definition file name)
- file security and other miscellaneous information

In INTOUCH, you address the structure file and not the dataset directly.

An INTOUCH structure is made up of **records** *and* **fields**. A structure looks something like this:

Data Structure Statements



In the above example, the records are the horizontal lines of information. In the CLIENT structure above, there is a record for each customer.

Each record consists of fields. For example, the customer records alone might contain a field for the customer's ID number, last name, first name, address, phone number, company name, etc. Each of these pieces of data is stored in its own field–the name field, address field, phone number field, etc.. The fields appear as columns in the diagram above.

To reference a field, you indicate the structure and the field you want. The field name is enclosed in parentheses.

```
struc_name(field_name)
```

struc_name is the name associated with the structure. *field_name* is the name of a field in the structure. INTOUCH searches for the field specified in the current record and reads its contents.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl: KEY ID = '80522'
      PRINT cl(last), cl(first) ! Print last and first
                                ! fields from the CL structure
    END EXTRACT
    CLOSE STRUCTURE cl
30  END

RNH
Errant             Earl
```

The remainder of this chapter describes the data structure statements and how to use them.

14.2 OPEN STRUCTURE

FORMAT:

```
OPEN STRUCTURE struc_name: NAME 'struc_filename'
  [, ACCESS INPUT | OUTIN] [, LOCK] [, DATAFILE filename]
  [, OPTIMIZE OFF]
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', &
      ACCESS INPUT, LOCK
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California clients by last name'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of California clients by last name
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

DESCRIPTION:

The **OPEN STRUCTURE** statement is used to open a data structure. The structure must be open in order to reference data in the structure (i.e. change field data, add or delete records). *struc_name* is a name (i.e. nickname) you assign to the structure. You use this name to refer to the structure within the program. The structure name must be unique within the program or program system. If the structure name is associated with another open structure, an exception is generated. The structure name must meet the requirements for variable names.

After the keyword **NAME**, is the file specification of the structure file being opened. (See Chapter 16, Creating Structures, Field Definitions with SETUP for information on legal structure file names.) The file specification can be any valid string expression.

The **ACCESS** option tells INTOUCH whether to open the structure for input (reading field data) or for input and output (reading, changing and adding field data). ACCESS *input* is the default open mode for structures, meaning, that if no ACCESS option is provided, INTOUCH opens the structure for INPUT.

When INTOUCH executes the OPEN statement, it searches for the structure file specified. INTOUCH opens the structure and assigns it the structure name. If the structure file does not exist, an exception is generated.

14.2.1 ACCESS Option

The **ACCESS** option determines how you can access the structure.

The access options are:

- ACCESS INPUT

  The structure is opened for input only. Fields can be referenced but not assigned new values. This is the default access method.

- ACCESS OUTIN

  The structure is opened for input and output. Fields can be referenced and assigned new values.

14.2.2 LOCK Option

The **LOCK** option causes INTOUCH to lock the structure from write access by others. As long as the structure is open, the structure cannot be modified by others. This can speed up INTOUCH's structure statements (EXTRACTs especially). The particular effect depends on the file management system being used.

14.2.3 DATAFILE Option

The **DATAFILE** option overrides the default datafile as specified by the structure.

FORMAT:

```
DATAFILE file_spec
```

file_spec is the file specification of the data file you want to open. The file_spec can be any string expression.

14.3 CLOSE STRUCTURE struc_name

FORMAT:

```
CLOSE STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California Clients'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of California Clients
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

DESCRIPTION:

CLOSE STRUCTURE closes a structure from further access. Once the structure is closed, you cannot reference records or add them, and you cannot change field data. *struc_name* is the name associated with the structure, the name assigned with the OPEN statement.

You can use the statement **CLOSE ALL** to close *all* open structures and other files.

14.4 ADD STRUCTURE

FORMAT:

```
ADD STRUCTURE struc_name
      ---
  [LET] struc_name(field) = expr
      ---
END ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  INPUT 'Enter ID number': id%
    INPUT 'Enter last name': last$
    INPUT 'Enter first name': first$
    INPUT 'Enter state': state$
    INPUT 'Enter phone': phone$
30  ADD STRUCTURE cl
      PRINT
      PRINT 'Adding '; last$; ', '; first$
      LET cl(id) = id%
      LET cl(last) = last$
      LET cl(first) = first$
      LET cl(state) = state$
      LET cl(phone) = phone$
    END ADD
40  CLOSE STRUCTURE cl
50  END
```

```
RNH
Enter ID number? 12233
Enter last name? Jones
Enter first name? Tom
Enter state? NV
Enter phone? 2345556161

Adding Jones, Tom
```

DESCRIPTION:

ADD STRUCTURE adds a record to a structure. The ADD STRUCTURE statement begins the add block. *struc_name* is the name associated with the structure that the record is going to be added to. **END ADD** marks the end of the block.

When ADD STRUCTURE is executed, INTOUCH executes the block of code between the ADD STRUCTURE statement and END ADD. This block of code with **LET** statements is used to put data into the fields.

LET assigns a value to the field specified. *struc_name(field)* specifies a field in the structure. *expr* is an expression. When INTOUCH executes the LET statement, it evaluates the expression and assigns the value of this expression to the field specified. END ADD writes out the new record.

14.4.1 CANCEL ADD

FORMAT:

```
CANCEL ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
    ADD STRUCTURE cl
      INPUT 'Client ID': cl(id)
      IF  _EXIT  THEN  CANCEL ADD
      INPUT 'Last name': cl(last)
      IF  _EXIT  THEN  CANCEL ADD
      INPUT 'First name': cl(first)
      IF  _EXIT  THEN  CANCEL ADD
      PRINT 'Adding client'
    END ADD
20  END

RNH
Client ID? 14422
Last name? WHITE
First name? EXIT
```

DESCRIPTION:

CANCEL ADD cancels the current adding of a record to a structure and transfers control to the next statement after the END ADD statement.

This statement can *only* be used within an ADD block—that is, between an ADD STRUCTURE and an END ADD pair of statements.

14.4.2 EXIT ADD

FORMAT:

```
EXIT ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
    ADD STRUCTURE cl
      INPUT 'Client ID ': cl(id)
      INPUT 'Last name ': cl(last)
      INPUT 'First name': cl(first)
      INPUT 'Contact   ': cl(contact)
      IF  _EXIT  THEN  EXIT ADD
      INPUT 'Salesman  ': cl(salesman)
      INPUT 'Mother    ': cl(mother)
      INPUT 'Comment   ': cl(comment)
    END ADD
    PRINT 'Client added'
20  END

RNH
Client ID ? 11111
Last name ? Hollerith
First name? Herman
Contact   ? EXIT
Client added
```

DESCRIPTION:

EXIT ADD transfers control immediately to the corresponding END ADD statement and **performs the add**.

This statement is useful when you want to add a record but do not have all the data. You can enter data into the first few fields and skip the rest of the fields.

14.5 DELETE STRUCTURE

FORMAT:

```
DELETE STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
30  ! Delete all clients from California
    FOR EACH cl
      PRINT 'Deleting '; cl(first); ' '; cl(last) ;'...';
      DELETE STRUCTURE cl
      PRINT 'record deleted'
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
Deleting Keith Kent...record deleted
Deleting Paul Johnson...record deleted
Deleting Wayne Waters...record deleted
Deleting Earl Errant...record deleted
Deleting Cathy Cass...record deleted
Deleting Pete Porter...record deleted
Deleting Dale Derringer...record deleted
```

DESCRIPTION:

DELETE STRUCTURE deletes the current record from the specified structure. *struc_name* is the structure name associated with the structure that the record is going to be deleted from.

14.6 LOCK | UNLOCK STRUCTURE

FORMAT:

```
[LOCK | UNLOCK] STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
30  FOR EACH cl
      PRINT
      PRINT cl(first); ' '; cl(last)
      LOCK STRUCTURE cl   ! Give us exclusive access
      LINE INPUT DEFAULT cl(phone), PROMPT 'Enter new phone ': phone$
      IF  _EXIT  THEN  EXIT FOR
      cl(phone) = phone$
      UNLOCK STRUCTURE cl ! Put the record out to disk
                          ! and release it
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH

Keith Kent
Enter new phone 6199675021

Paul Johnson
Enter new phone EXIT
```

DESCRIPTION:

You can use **LOCK/UNLOCK STRUCTURE** to lock or unlock the data record currently being accessed. INTOUCH automatically locks and unlocks data when you work with it. However, you can use LOCK and UNLOCK if you want to do this manually.

Normally, INTOUCH locks and unlocks data as needed. LOCK and UNLOCK override INTOUCH's automatic locking features. LOCK STRUCTURE locks the data currently being accessed, giving the program exclusive access to the current record. No one else can access the data until it is unlocked.

UNLOCK unlocks the current record or data. The record is put to disk (if needed) and can again be accessed by others. *struc_name* is the structure name associated with the structure.

14.7 EXTRACT STRUCTURE

FORMAT:

```
EXTRACT STRUCTURE struc_name
        ---  block of code
  [INCLUDE | EXCLUDE] cond_expr
  [SORT [ASCENDING | DESCENDING] BY expression]
        ---
END EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  PRINT 'List of Clients'
    PRINT
30  EXTRACT STRUCTURE cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    END EXTRACT
40  CLOSE STRUCTURE cl
50  END

RNH
List of Clients

Earl Errant          (408) 844-7676
Al Abott             (202) 566-9892
Bud Brock            (218) 555-4322
Cathy Cass           (619) 743-8582
Dale Derringer       (818) 223-9014
Fred Farmer          (305) 552-7872
```

DESCRIPTION:

When INTOUCH does an extract, it examines each record in the structure. For each record, INTOUCH executes the code between the **EXTRACT STRUCTURE** and **END EXTRACT** statements. For instance, here is a structure with client information:

ID #	LAST	FIRST	CITY	STATE	PHONE
80543	Cass	Cathy	San Diego	CA	6197438582
80542	Brock	Bud	Duluth	MN	2185554322
80522	Errant	Earl	Monterey	CA	4088447676
80561	Derringer	Dale	Los Angeles	CA	8182239014
80531	Abott	Al	New York	NY	2025669892
80573	Farmer	Fred	Miami	FL	3055527872

EXTRACT creates a list of clients. The above program example extracts information from each record in the structure and displays each client's name and phone number.

When INTOUCH does an extract, it examines each record of the structure in order. If the **KEY** option is specified, only those records with a key matching the KEY expression are examined. For each examined record, the following is done:

1 Each **INCLUDE** and **EXCLUDE** statement is checked in turn. The examined record is *not* extracted if an INCLUDE statement evaluates to *FALSE*, or an EXCLUDE statement evaluates to *TRUE*.

2 If any **SORT** specifications are given, a sort key is built using the SORT expression as the key. If no SORT specifications are given, the record is immediately extracted.

3 When all records have been examined, the sort keys, if any, are sorted.

INTOUCH then builds a list of extracted records. This list can be accessed later with a **FOR EACH** loop. You can have up to 16 sort keys and 32 extract criteria.

14.7.1 INCLUDE Statement

FORMAT:

```
INCLUDE cond_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
    PRINT 'List of California Clients'
    PRINT
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(state)
    NEXT cl
    CLOSE STRUCTURE cl
30  END
```

```
RNH
List of California Clients

Keith Kent          CA
Paul Johnson        CA
Wayne Waters        CA
Earl Errant         CA
Cathy Cass          CA
Pete Porter         CA
Dale Derringer      CA
```

DESCRIPTION:

The **INCLUDE** statement includes records depending on the value of a conditional expression.

cond_expr is a conditional expression that INTOUCH evaluates. If the expression is **TRUE**, INTOUCH **includes** the record in the extract list. If the expression is **FALSE**, INTOUCH excludes the record from the list. For example, the program above creates an extract list containing only those clients located in California.

NOTE: The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

14.7.2 EXCLUDE Statement

FORMAT:

```
EXCLUDE cond_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      EXCLUDE cl(phone)[1:3] = '619'
    END EXTRACT
30  PRINT 'List of Clients'
    PRINT
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
    CLOSE STRUCTURE cl
40  END

RNH
List of Clients

Earl Errant         (408) 844-7676
Al Abott            (202) 566-9892
Bud Brock           (218) 555-4322
Dale Derringer      (818) 223-9014
Fred Farmer         (305) 552-7872
```

DESCRIPTION:

The **EXCLUDE** statement excludes records from the extract list, depending on the value of a conditional expression.

cond_expr is a conditional expression. INTOUCH evaluates this expression. If the expression is **TRUE**, INTOUCH **excludes** the current record from the extract list. For example, the program above creates an extract list of all the clients in the client structure—except those with an area code of 619.

NOTE: The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

14.7.3 SORT Statement

FORMAT:

```
SORT [ASCENDING | DESCENDING] BY expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      SORT ASCENDING BY cl(last)
    END EXTRACT
    PRINT 'List of Clients'
    PRINT
30  FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of Clients

Al Abott            (202) 566-9892
Bud Brock           (218) 555-4322
Cathy Cass          (619) 743-8582
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
Fred Farmer         (305) 552-7872
```

DESCRIPTION:

The **SORT** statement sorts the records in an extract list in either **ASCENDING** or **DESCENDING** order. *expr* is an expression whose value determines how to order the list. INTOUCH evaluates the expression for each record and stores the value. When all the records have been extracted, INTOUCH orders the list according to these values.

You can sort in either ASCENDING or DESCENDING order. ASCENDING creates a list in ascending order (lowest to highest). DESCENDING creates a list in descending order (highest to lowest). The default is ascending order. String values are sorted according to the ASCII character set.

INTOUCH does sorts in order. Therefore, you can use multiple sorts to order the list more and more specifically. For example, the following program creates a list of clients. The clients are sorted first by state and within each state by last name.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
      SORT ASCENDING BY cl(state)
      SORT ASCENDING BY cl(last)
    END EXTRACT
20  PRINT 'List of Clients'
    PRINT
    FOR EACH cl
      PRINT cl(last); ', '; cl(first), cl(state)
    NEXT cl
30  CLOSE STRUCTURE CL
40  END

RNH
List of Clients

Cass, Cathy            CA
Derringer, Dale        CA
Errant, Earl           CA
Farmer, Fred           FL
Brock, Bud             MN
Abott, Al              NY
```

When you sort fields that are filled with nulls (no data was ever stored in them), the fields are sorted as though they were space filled.

14.7.4 FOR EACH/NEXT

FORMAT:

```
FOR EACH struc_name
        ---
        ---  block of code
        ---
NEXT struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
    PRINT 'List of California Clients'
    PRINT
30  FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50 END

RNH
List of California Clients

Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

DESCRIPTION:

You can use the **FOR EACH** statement to execute a block of code for each record in the extract list. This allows you to manipulate structure information in your programs.

The FOR EACH statement begins a loop that executes a block of code for each record in the extract list. *struc_name* is the structure name associated with the structure. **NEXT struc_name** marks the end of the loop.

You can use the **REPEAT**, **ITERATE** and **EXIT FOR** statements in the FOR EACH loop.

14.7.5 EXTRACT STRUCTURE: KEY

FORMAT:

```
EXTRACT STRUCTURE struc_name: KEY field = expr1 [TO expr2]
          ---
          ---  block of code
          ---
END EXTRACT
```

struc_name is the structure name associated with the structure. *field* is the name of the field that contains the key. *expr* is an expression that tells what key(s) to extract. INTOUCH evaluates the expression, checks the structure's index for records with matching keys, and extracts these records (if any records are found).

KEY Option

The **KEY** option includes records using the record's *key*. The key is a field which has an index for fast access. The key option can considerably speed up extractions.

The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

For example, we have a structure with the following client information and the ID field is a key field:

ID #	LAST	FIRST	CITY	STATE	PHONE
80543	Cass	Cathy	San Diego	CA	6197438582
80542	Brock	Bud	Duluth	MN	2185554322
80522	Errant	Earl	Monterey	CA	4088447676
80561	Derringer	Dale	Los Angeles	CA	8182239014
80531	Abott	Al	New York	NY	2025669892
80573	Farmer	Fred	Miami	FL	3055527872

In the program below, the KEY option is used to extract the client with the ID number 80561.

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
20  EXTRACT STRUCTURE cl: KEY id = 80561
      PRINT 'Client:',
      PRINT cl(first); ' '; cl(last), cl(id)
    END EXTRACT
    CLOSE STRUCTURE cl
30  END

RNH
Client:            Dale Derringer        80561
```

TO expr Option

You can extract records with keys in a certain range with the **TO** option. *expr1* is the lowest key to check. *expr2* is the highest. INTOUCH extracts any records whose keys are within the range specified.

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
20  INPUT 'Enter the lowest ID to check': lowest
    INPUT 'Enter the highest ID to check': highest
    EXTRACT STRUCTURE cl: KEY id = lowest TO highest
      PRINT cl(id); TAB(10); cl(last)
    END EXTRACT
    CLOSE STRUCTURE cl
30  END
```

```
RNH
Enter the lowest ID to check? 80540
Enter the highest ID to check? 80570
80542    Brock
80543    Cass
80561    Derringer
```

14.7.6 EXTRACT STRUCTURE, FIELD: PARTIAL KEY

FORMAT:

```
EXTRACT STRUCTURE struc_name, FIELD field_expr: PARTIAL KEY str_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
    EXTRACT STRUCTURE cl, FIELD last: PARTIAL KEY 'Ros'
    END EXTRACT
    PRINT 'List of clients with last name starting with Ros'
    PRINT
    FOR EACH cl
      PRINT cl(first); ' '; cl(last)
    NEXT cl
20  CLOSE STRUCTURE cl
30  END
```

```
RNH
Bud Roske
Earl Rost
Dale Rosty
```

DESCRIPTION:

The **PARTIAL KEY** option will search in the EXTRACT STRUCTURE for part of a key value.

Here is a structure with the following client information and the ID is a key field:

ID #	LAST	FIRST	CITY	STATE	PHONE
80543	Roberts	Cathy	San Diego	CA	6197438582
80542	Roske	Bud	Duluth	MN	2185554322
80522	Rost	Earl	Monterey	CA	4088447676
80561	Rosty	Dale	Los Angeles	CA	8182239014
80531	Abott	Al	New York	NY	2025669892
80573	Farmer	Fred	Miami	FL	3055527872

The above example program creates an extract list containing only those clients with a last name starting with "ROS".

14.7.7 CANCEL EXTRACT

FORMAT:

```
CANCEL EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
    EXTRACT STRUCTURE cl
      PRINT 'Client: '; cl(last)
      LINE INPUT 'Press return to continue': z$
      IF  _EXIT  THEN  CANCEL EXTRACT
    END EXTRACT
    PRINT 'Records extracted:'; _EXTRACTED
20  CLOSE STRUCTURE cl
30  END

RNH
Client: Smith
Press return to continue? EXIT
Records extracted: 0
```

DESCRIPTION:

CANCEL EXTRACT cancels the current extract of a record and transfers control to the next statement after the END EXTRACT statement.

This statement can *only* be used within an EXTRACT block—that is, between an EXTRACT STRUCTURE and an END EXTRACT pair of statements.

14.7.8 EXIT EXTRACT

FORMAT:

```
EXIT EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
      PRINT 'Client: '; cl(last)
      LINE INPUT 'Press return to continue': z$
      IF  _EXIT  THEN  EXIT EXTRACT
    END EXTRACT
    PRINT 'Records extracted:'; _EXTRACTED
20  END

RNH
Client: Smith
Press return to continue? <RETURN>
Client: Kent
Press return to continue? EXIT
Records extracted: 1
```

DESCRIPTION:

EXIT EXTRACT passes control to the corresponding END EXTRACT statement, performs final sorting (if any), and creates the extracted collection.

14.7.9 REEXTRACT STRUCTURE

FORMAT:

```
REEXTRACT STRUCTURE struc_name
          ---
   [INCLUDE | EXCLUDE] cond_expr...
   [SORT [ASCENDING | DESCENDING] BY expression...
          ---
END EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
    REEXTRACT STRUCTURE cl
      EXCLUDE cl(phone)[1:3] <> '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California Clients in Area Code 619'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of California Clients in Area Code 619
Cathy Cass          (619) 743-8582
Paul Johnson        (619) 489-5551
Keith Kent          (619) 967-5021
Pete Porter         (619) 778-6709
Wayne Waters        (619) 564-1231
```

DESCRIPTION:

You can use **REEXTRACT STRUCTURE** to do a second extract on a list of structure records you previously extracted. This lets you choose more and more specific records through a series of REEXTRACTs.

REEXTRACT does an extract on the list of records previously extracted. *struc_name* is the structure name associated with an open structure.

END EXTRACT marks the end of the REEXTRACT construct. REEXTRACT operates the same as EXTRACT. However, REEXTRACT operates on a previously extracted list.

Note

Extract operations by key cannot be performed with REEXTRACT.

14.7.10 EXTRACT STRUCTURE: APPEND

FORMAT:

```
EXTRACT STRUCTURE struc_name: APPEND
```

EXAMPLE:

```
10  OPEN STRUCTURE detail: name 'tti_run:detail'
    SET STRUCTURE detail: EXTRACTED 0
20  EXTRACT STRUCTURE detail, FIELD lineid : &
        KEY '10301001' TO '10301999', APPEND
      SORT BY detail(prodnbr)
      SORT BY detail(invnbr)
    END EXTRACT
30  EXTRACT STRUCTURE detail, field lineid : &
        KEY '10311001' to '10311999', APPEND
      SORT BY detail(prodnbr)
      SORT BY detail(invnbr)
    END EXTRACT
    PRINT 'Prod'; TAB(7); 'Line ID'; TAB(17); 'Qty'
40  FOR EACH detail
      PRINT detail(prodnbr); TAB(7); detail(lineid); &
            TAB(17); detail(qty)
    NEXT detail
50  END

RNH
Prod  Line ID   Qty
22800 10301-002      2
22800 10301-004      1
22800 10301-006      2
24100 10311-003      1
24200 10301-003      1
24200 10311-009      1
28400 10311-001      2
28800 10301-009      2
28800 10311-002      9
28800 10311-005      1
28800 10311-006      1
31020 10301-005      1
31040 10311-010      2
31150 10301-001      1
31150 10301-008      8
31150 10311-004      1
31150 10311-008      1
33090 10301-007      2
33090 10311-007      1
```

DESCRIPTION:

The **EXTRACT STRUCTURE: APPEND** statement adds records to the last collection of extracted records rather than creating a new collection.

14.7.11 EXTRACT STRUCTURE, SET, USING

FORMAT:

```
EXTRACT STRUCTURE struc_name1, SET 'set_name', USING struc_name2
```

EXAMPLE:

```
10  OPEN STRUCTURE class: name 'devint:intest_dbms'
20  OPEN STRUCTURE part : name 'devint:intest_dbms'
    EXTRACT STRUCTURE class, SET 'class_part', USING part
    END EXTRACT
30  END
```

DESCRIPTION:

Used for DBMS handling, this statement fetches all structures, (*struc_name1*) owned by another structure (*struct_name2*) within a given set.

14.7.12 EXTRACT STRUCTURE, SET, FIELD: KEY

FORMAT:

```
EXTRACT STRUCTURE struc_name1, SET 'set_name', FIELD field_expr: KEY str_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE class:  name 'devint:intest_dbms'
20  OPEN STRUCTURE part :  name 'devint:intest_dbms'
    EXTRACT STRUCTURE class, SET 'class_part', FIELD class_code: KEY cl_code$
    END EXTRACT
30  END
```

DESCRIPTION:

Used in DBMS handling, this statement allows you to extract all structures specified by *struc_name1* owned by the current record within the given set (*set_name*) with field *field_expr* equal to the given key.

14.8 ASK STRUCTURE

The **ASK STRUCTURE** statement is used to ask about various device and structure characteristics from within your programs.

FORMAT:

```
ASK STRUCTURE struc_name: struc_option [num_var | str_var]
```

DESCRIPTION:

struc_name is the name of a structure whose characteristics are being asked about. *struc_option* can be any of the structure options available. The options are explained in the following sections.

14.8.1 ASK STRUCTURE FIELD: item

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: item var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  ASK STRUCTURE cl: FIELDS num_fields
    FOR i = 1 TO num_fields
      CLEAR
      ASK STRUCTURE cl, FIELD #i: DESCRIPTION b$
      ASK STRUCTURE cl, FIELD #i: PROMPT a$
      ASK STRUCTURE cl, FIELD #i: POSITION a%
      ASK STRUCTURE cl, FIELD #i: LENGTH b%
      ASK STRUCTURE cl, FIELD #i: HEADING f$
      ASK STRUCTURE cl, FIELD #i: PRINTMASK c$, &
                                  SCREENMASK d$, &
                                  HELP e$
      PRINT AT 5,5: ''
      PRINT 'Description   : '; b$
      PRINT 'Prompt        : '; a$
      PRINT 'Position      : '; a%
      PRINT 'Field length  :' ; b%
      PRINT 'Rpt. heading  : '; f$
      PRINT 'Print mask    : '; c$
      PRINT 'Screen mask   : '; d$
      PRINT 'Help          : '; e$
      DELAY 5
    NEXT i
    CLOSE STRUCTURE cl
30  END
```

```
RNH
Description   : Client ID number
Prompt        : Client ID number
Position      :  1
Field length  : 5
Rpt. heading  : CLNT,ID
Print mask    : >####
Screen mask   : digits:#####
Help          : Client ID number
```

DESCRIPTION:

The **FIELD** option allows you to get information about a specific *field* in a structure. *struc_name* is the name of the structure. *field_expr* is the field you are inquiring about. *item* specifies what type of information you are asking. The information is stored in the variable specified.

The following sections provide more information on what you can use for *field_expr* and on the various field *items*.

Note

The *ITEM* information is created when the the SETUP routine is used to define the field. You can refer to Chapter 16, Creating Structures, Field Definitions with SETUP for information on defining fields.

14.8.1.1 FIELD Expressions

The **field_expr** used in the **ASK STRUCTURE FIELD: item** statement can be either a *constant* or a *string or numeric expression.*

You can use a string constant to specify the field name. To use a string constant, just enter the field name, *without quotes*. INTOUCH will then use the string constant as the field name:

```
ASK STRUCTURE TTI_RUN:CLIENT, FIELD LAST: DESCRIPTION A$
                                    /
          the field is specified by its field name
```

You can also specify a field name with an expression. To use an expression, precede the expression with a pound sign (#). The pound sign tells INTOUCH that the following characters are an expression, not the field name. If you do not include a pound sign, INTOUCH will interpret the characters as a field name.

```
ASK STRUCTURE CL, FIELD #FIELDNAME$: DESCRIPTION A$
                               /
  the field is specified by the value of the variable FIELDNAME$

ASK STRUCTURE CL, FIELD #FIELDNUM: DESCRIPTION A$
                              /
  the field is specified by the value of the variable FIELDNUM
```

EXAMPLE:

This example shows how you can access the actual field data using field expressions.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  DO
    CLEAR
    PRINT AT 1, 1:;
      ask_fields
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      show_data
    LOOP
    CLOSE STRUCTURE cl
    STOP
```

```
30  ask_fields:
    DO
      IF  _ERROR  THEN  SET ERROR OFF
      PRINT 'Field List:  '; &
            'ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE'
      PRINT
      LINE INPUT 'Select fields to display': field_list$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      FOR f = 1 TO ELEMENTS(field_list$)
        field$ = ELEMENT$(field_list$, f)
        ASK STRUCTURE cl, FIELD #field$: NUMBER z
        IF  z = 0  THEN
          MESSAGE ERROR: 'Illegal field: '; field$
        END IF
      NEXT f
40  LOOP WHILE _ERROR
    RETURN

50  show_data:
    PRINT
    EXTRACT STRUCTURE cl
      FOR f = 1 TO ELEMENTS(field_list$)
        field$ = ELEMENT$(field_list$, f)
        PRINT cl(#field$),
      NEXT f
      PRINT
60  END EXTRACT
    DELAY
    RETURN
70  END

RNH
Field List:  ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE

Select fields to display? last,first,phone

Smith               Sam                 (809) 555-8789
Kent                Keith               (619) 967-5021
Johnson             Paul                (619) 489-5551
Waters              Wayne               (619) 564-1231
Rodrigues           Homero              (   )    -   0
Donaldson           George              (   )    -   0
Errant              Earl                (408) 844-7676
Abott               Al                  (202) 566-9892
Brock               Bud                 (218) 555-4322
Cass                Cathy               (619) 743-8582
Porter              Pete                (619) 778-6709
Derringer           Dale                (818) 223-9014
Farmer              Fred                (305) 552-7872
```

14.8.1.2 Item: ACCESS

```
ASK STRUCTURE struc_name, FIELD field_name: ACCESS str_var
```

ACCESS retrieves the access (read and write) rules for the specified field. This information tells you if the field can be read and written to. **N** is normal–meaning the field can be read and written to if the structure is also "N"ormal. Depending on whether security levels have been set on the structure and/or field, the letter can be in the range of A thru Z. INTOUCH defaults to N when the structure is created and fields are defined.

```
10  OPEN STRUCTURE inv: NAME 'tti_run:invoice', ACCESS INPUT
20  ASK STRUCTURE inv, FIELD custnbr: ACCESS x$
30  PRINT x$
40  CLOSE STRUCTURE inv
50  END

RNH
READ:N, WRITE:N
```

14.8.1.3 Item: APPLICATION

```
ASK STRUCTURE struc_name, FIELD field_expr: APPLICATION str_var
```

APPLICATION returns a name of an application for the specified field in a string variable. This is optional information the user provides when the field is defined.

```
10  OPEN STRUCTURE cl : NAME 'tti_run:client'
    ASK STRUCTURE cl, FIELD id: APPLICATION str$
    PRINT str$
20  END

RNH
REPORTING_ID
```

14.8.1.4 Item: ATTRIBUTES

```
ASK STRUCTURE struc_name, FIELD field_expr: ATTRIBUTES str_var
```

ATTRIBUTES returns the INTOUCH field semantics (NUM - number, UC - upper-case, etc.) for the specified field in a string variable.

You can refer to Section 16.4.8, Semantics for detailed information on field attributes.

14.8.1.5 Item: CHANGEABLE

```
ASK STRUCTURE struc_name, FIELD field_expr: CHANGEABLE num_var
```

CHANGEABLE returns a value of TRUE or FALSE. If the field specified by *field_expr* can be changed, the value is TRUE. If the field cannot be changed, the value is FALSE.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    ASK STRUCTURE cl, FIELD id: CHANGEABLE z
    ASK STRUCTURE cl, FIELD city: CHANGEABLE z1
    PRINT z
    PRINT z1
    CLOSE STRUCTURE CL
20  END

RNH
 0
 1
```

14.8.1.6 Item: CLASSIFICATION

```
ASK STRUCTURE struc_name, FIELD field_expr: CLASSIFICATION str_var
```

CLASSIFICATION returns the classification (if there is one) for the specified field in a string variable. This is optional information.

```
10  OPEN STRUCTURE cl : NAME 'tti_run:client'
    ASK STRUCTURE cl, FIELD last: CLASSIFICATION str$
    PRINT str$
20  END

RNH
NAME
```

14.8.1.7 Item: DATATYPE

```
ASK STRUCTURE struc_name, FIELD field_expr: DATATYPE str_var
```

DATATYPE returns the field data type, such as CH (character), IN (integer), etc., in a string variable.

You can refer to Section 16.4.4, Data type for detailed information on field data types.

14.8.1.8 Item: DESCRIPTION

```
ASK STRUCTURE struc_name, FIELD field_expr: DESCRIPTION str_var
```

DESCRIPTION returns the description for the specified field in a string variable.

14.8.1.9 Item: HEADING

```
ASK STRUCTURE struc_name, FIELD field_expr: HEADING str_var
```

HEADING returns the report column heading for the specified field in a string variable. This is the heading that would appear in a Guided Query Language (GQS) report column.

14.8.1.10 Item: HELP

```
ASK STRUCTURE struc_name, FIELD field_expr: HELP str_var
```

HELP returns the help text for the specified field in a string variable.

14.8.1.11 Item: KEYED

```
ASK STRUCTURE struc_name, FIELD field_expr: KEYED num_var
```

KEYED returns a value of TRUE or FALSE in a numeric variable. If the specified field is a key field, the value is TRUE. Otherwise, the value is FALSE.

14.8.1.12 Item: LENGTH

```
ASK STRUCTURE struc_name, FIELD field_expr: LENGTH num_var
```

LENGTH returns the length of the specified field in a numeric variable.

14.8.1.13 Item: NAME

```
ASK STRUCTURE struc_name, FIELD field_expr: NAME str_var
```

NAME returns the name of the specified field in a string variable.

14.8.1.14 Item: NULL

```
ASK STRUCTURE struc_name, FIELD field_name: NULL int_var
```

If the specified field is **NULL** (i.e. contains no data), this statement returns TRUE. If the field is not NULL, the statement returns FALSE.

14.8.1.15 Item: NUMBER

```
ASK STRUCTURE struc_name, FIELD field_expr: NUMBER num_var
```

NUMBER returns the field number of the specified field in a numeric variable. Fields are numbered sequentially. If the field does not exist, INTOUCH returns a value of 0.

14.8.1.16 Item: OPTIMIZED

```
ASK STRUCTURE struc_name, FIELD field_expr: OPTIMIZED num_var
```

OPTIMIZED returns a value of TRUE or FALSE in a specified numeric variable. If the key field in *field_expr* is optimized, the value is TRUE. Otherwise, the value is FALSE.

```
10  OPEN STRUCTURE c1: NAME 'tti_run:client'
    ASK STRUCTURE c1, FIELD id: OPTIMIZED z
    PRINT z
    CLOSE STRUCTURE c1
20  END

RNH
 1
```

14.8.1.17 Item: POSITION

```
ASK STRUCTURE struc_name, FIELD field_expr: POSITION num_var
```

POSITION returns the starting position for the specified field in a numeric variable.

14.8.1.18 Item: PRINTMASK

```
ASK STRUCTURE struc_name, FIELD field_expr: PRINTMASK str_var
```

PRINTMASK returns the print mask for the specified field in a string variable.

14.8.1.19 Item: PROMPT

```
ASK STRUCTURE struc_name, FIELD field_expr: PROMPT str_var
```

PROMPT returns the prompt for the specified field in a string variable.

14.8.1.20 Item: SCREENMASK

```
ASK STRUCTURE struc_name, FIELD field_expr: SCREENMASK str_var
```

SCREENMASK returns the screen mask for the specified field in a string variable. This option is not currently used.

14.8.1.21 Item: VRULES

```
ASK STRUCTURE struc_name, FIELD field_expr: VRULES str_var
```

VRULES returns the validation rules for the specified field in a string variable.

You can refer to the "VALID" function item in Section A.2, Other Functions for information on validation rules.

```
10  OPEN STRUCTURE cl : NAME 'tti_run:client'
    ASK STRUCTURE cl, FIELD bday: VRULES str$
    PRINT str$
20  END

RNH
date ymd; minlength 8
```

14.8.2 ASK STRUCTURE: CURRENT

FORMAT:

```
ASK STRUCTURE struc_name: CURRENT str_var
```

EXAMPLE:

```
10  DIM a$(100)
    LET i = 0
20  OPEN STRUCTURE cl: NAME 'tti_run:client'
30  EXTRACT STRUCTURE cl
    END EXTRACT
40  FOR EACH cl
      PRINT cl(last); ', '; cl(first)
      INPUT 'Would you like to see this record (Y/N)': yn$
      IF  yn$ = 'Y'  THEN
        LET i = i + 1
        ASK STRUCTURE cl: CURRENT a$(i)
      END IF
    NEXT cl
50  PRINT
    FOR j = 1 TO i
      SET STRUCTURE cl: CURRENT a$(j)
      PRINT cl(last); ','; cl(first), cl(state), cl(phone)
    NEXT j
60  END
```

```
RNH
Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

DESCRIPTION:

ASK STRUCTURE: CURRENT assigns the current record value to the *str_var*. Once the current record has been assigned with ASK STRUCTURE: CURRENT, you can use the SET STRUCTURE: CURRENT statement to get this record.

INTOUCH returns a zero length string if there is no CURRENT record.

14.8.3 ASK STRUCTURE: FIELDS

FORMAT:

```
ASK STRUCTURE struc_name: FIELDS num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: FIELDS z
    PRINT z
20  END

RNH
 18
```

DESCRIPTION:

You can find out the number of fields in a structure with the **ASK STRUCTURE: FIELDS** statement. The number is assigned to the numeric variable *num_var*.

14.8.4 ASK STRUCTURE: KEYS

FORMAT:

```
ASK STRUCTURE struc_name: KEYS num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: KEYS z
    PRINT z
20  END

RNH
 2
```

DESCRIPTION:

The **ASK STRUCTURE: KEYS** statement returns the number of keys that are accessible by INTOUCH. It returns the value of 0 if no keys are available.

14.8.5 ASK STRUCTURE: CAPABILITY

FORMAT:

```
ASK STRUCTURE struc_name: CAPABILITY str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: CAPABILITY z$
    PRINT z$
20  END

RNH
INDEXED,INPUT
```

DESCRIPTION:

Given a structure expression, **ASK STRUCTURE: CAPABILITY** sets *str_expr* to a comma delimited string containing one or more of the following: INDEXED, RELATIVE, INPUT, OUTPUT

Table 14–1 Structure Types

Structure Type	Description
RELATIVE	You can access a structure, using a relative record number, with statements such as ASK/SET STRUCTURE...RECORD.
INDEXED	The structure is indexed; you can access it by key with statements such as SET STRUCTURE...KEY.
INPUT	You can read from the structure.
OUTPUT	You can write to the structure.
null string	The structure is not currently open.

14.8.6 ASK STRUCTURE: EXTRACTED

FORMAT:

```
ASK STRUCTURE struc_name: EXTRACTED num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
    END EXTRACT
    ASK STRUCTURE cl: EXTRACTED z
    PRINT 'Records found: '; z
20  END

RNH
Records found:  13
```

DESCRIPTION:

ASK STRUCTURE: EXTRACTED asks the operating system for the last extracted count for the structure specified.

14.8.7 ASK STRUCTURE: ID

FORMAT:

```
ASK STRUCTURE struc_name: ID str_var
```

EXAMPLE:

```
10  DECLARE STRUCTURE str
    OPEN STRUCTURE cl: NAME 'tti_run:client'
    ASK STRUCTURE cl: ID cl_id$
    SET STRUCTURE str: ID cl_id$
20  EXTRACT STRUCTURE str
    END EXTRACT
    FOR EACH str
      PRINT str(#1); '  '; str(#2)
    NEXT str
30  END
```

```
RNH
20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

The **ASK STRUCTURE: ID** statement asks the operating system for the ID of a structure and returns it in the string variable *str_var*.

14.8.8 ASK STRUCTURE: POINTER

FORMAT:

```
ASK STRUCTURE struc_name: POINTER num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
    END EXTRACT
    FOR EACH cl
      ASK STRUCTURE cl: POINTER ptr
      PRINT ptr, cl(last)
    NEXT cl
20  END
```

```
RNH
 1                Smith
 2                Jones
 3                Kent
 4                Johnson
 5                Waters
 6                Errant
 7                Brock
 8                Cass
 9                Porter
 10               Derringer
 11               Farmer
```

DESCRIPTION:

From within a FOR EACH...NEXT STRUCTURE_NAME block, **ASK STRUCTURE: POINTER** asks the structure for the number of the current record pointer.

14.8.9 ASK STRUCTURE: RECORD

FORMAT:

```
ASK STRUCTURE struc_name: RECORD num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE ml: NAME 'tti_run:maint_log'
    EXTRACT STRUCTURE ml
      ASK STRUCTURE ml: RECORD ml_rec
      PRINT ml_rec, ml(cost)
    END EXTRACT
20  END

RNH
 1                    $450.00
 2                  $3,200.00
 3                    $370.00
 4                  $1,200.00
 5                    $600.00
 6                    $770.00
```

DESCRIPTION:

ASK STRUCTURE: RECORD asks for the relative record number of the current record and places that number in the numeric variable given.

14.8.10 ASK STRUCTURE: RECORDSIZE

FORMAT:

```
ASK STRUCTURE struc_name: RECORDSIZE int_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  ASK STRUCTURE cl: RECORDSIZE recsize
30  PRINT 'Logical recordsize: '; recsize
40  END

RNH
Logical recordsize:   400
```

DESCRIPTION:

The **ASK STRUCTURE: RECORDSIZE** statement returns the record size of the structure data file.

14.8.11 ASK STRUCTURE: ACCESS

FORMAT:

```
ASK STRUCTURE struc_name: ACCESS str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE inv: NAME 'tti_run:invoice', ACCESS INPUT
20  ASK STRUCTURE inv: ACCESS x$
30  PRINT x$
40  CLOSE STRUCTURE inv
50  END

RNH
SECURITY:N, READ:N, WRITE:N, UPDATE:N, DELETE:N
```

DESCRIPTION:

The **ASK STRUCTURE: ACCESS** statement retrieves the access rules for the specified structure. Security level, data file read, write, update, and delete rules are returned.

14.8.12 ASK | SET STRUCTURE: TIMEOUT

FORMAT:

```
ASK STRUCTURE struc_name: TIMEOUT int_var
SET STRUCTURE struc_name: TIMEOUT int_expr
```

DESCRIPTION:

The **ASK STRUCTURE: TIMEOUT** statement asks for the current TIMEOUT setting.

The **SET STRUCTURE: TIMEOUT** statement is used when a record is locked for over the *int_expr*. INTOUCH then returns a TIMEOUT error.

14.8.13 ASK | SET STRUCTURE #string_expr . . .

FORMAT:

```
ASK STRUCTURE #string_expr. . .
SET STRUCTURE #string_expr. . .
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    str$ = 'CL'
    fld$ = 'ID'
    do_work
    STOP

20  ROUTINE do_work
    ASK STRUCTURE #str$, FIELD #fld$: DESCRIPTION dsc$
    PRINT 'Description is: '; dsc$
    END ROUTINE

30  END

RNH
Description is: Client ID number
```

DESCRIPTION:

You can use a string expression for the structure name in an **ASK STRUCTURE #string_expr** or **SET STRUCTURE #string_expr** statement. This allows you to write generalized code to work for several structures.

14.8.14 ASK STRUCTURE: ENGINE

FORMAT:

```
ASK STRUCTURE struc_name : ENGINE str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE c1 : name 'tti_run:vendor'
20  ASK STRUCTURE  c1 : ENGINE ename$
30  PRINT 'DATABASE ENGINE is: '; ename$
40  END

RNH
DATABASE ENGINE is: RMS
```

DESCRIPTION:

The **ASK STRUCTURE: ENGINE** statement returns the name of the database engine (record management system) being used for the specified structure in *str_var*.

14.9 SET STRUCTURE

The **SET STRUCTURE** statement is used to change various device and structure characteristics from within your programs.

FORMAT:

```
SET STRUCTURE struc_name: struc_option [num_var | str_var]
```

SET STRUCTURE sets characteristics of structures. *struc_name* is the name of the structure whose characteristics are being set. *struc_option* is the option you are setting. The options are explained in the following sections.

14.9.1 SET STRUCTURE: CURRENT

FORMAT:

```
SET STRUCTURE struc_name: CURRENT str_expr
```

EXAMPLE:

```
10  DIM a$(100)
    LET i = 0
20  OPEN STRUCTURE cl: NAME 'tti_run:client'
30  EXTRACT STRUCTURE cl
    END EXTRACT
40  FOR EACH cl
      PRINT cl(last); ', '; cl(first)
      INPUT 'Would you like to see this record (Y/N)': yn$
      IF  yn$ = 'Y'  THEN
        LET i = i + 1
        ASK STRUCTURE cl: CURRENT a$(i)
      END IF
    NEXT cl
50  PRINT
    FOR j = 1 TO i
      SET STRUCTURE cl: CURRENT a$(j)
      PRINT cl(last); ','; cl(first), cl(state), cl(phone)
    NEXT j
60  END
```

```
RNH
Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

DESCRIPTION:

The **CURRENT** option sets the current record to that specified by the *str_expr*. The str_expr contains the information for the current record for the record management system you are using. You can use ASK STRUCTURE: CURRENT to store a current record value into a string variable.

When INTOUCH executes a SET STRUCTURE: CURRENT statement, it uses the structure name and sets the current record to the value specified by the string variable. The structure must be open and str_expr must contain a value stored with the **ASK STRUCTURE: CURRENT** statement.

If you use a null string for the value, INTOUCH sets the structure to no current record and sets _EXTRACTED to zero.

```
SET STRUCTURE struc_name: CURRENT ''
```

14.9.2 SET STRUCTURE, FIELD: KEY

FORMAT:

```
SET STRUCTURE struc_name, FIELD field_expr: KEY str_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    LINE INPUT 'Enter an ID': id$
    SET STRUCTURE cl, FIELD id: KEY id$
    IF  _EXTRACTED = 0  THEN
       MESSAGE ERROR: 'Not found'
    ELSE
       PRINT cl(id), cl(last)
    END IF
20  END

RNH
Enter an ID? 80561
80561              Derringer
```

DESCRIPTION:

The **FIELD** option lets you get a record by means of a key field in a structure. You can use SETUP's "SHOW FIELDS" menu option (see Chapter 16, Creating Structures, Field Definitions with SETUP) to see the field names. The FIELD option is currently used only with the **KEY** or **PARTIAL KEY** option. The KEY option specifies the key to look for. The key is contained in *str_expr*.

The above example shows how to look in the CLIENT structure for an ID.

_EXTRACTED contains the number of records extracted. If the operation fails, _EXTRACTED will be 0 and an error message will be displayed.

14.9.3 SET STRUCTURE, FIELD: PARTIAL KEY

FORMAT:

```
SET STRUCTURE struc_name, FIELD field_expr: PARTIAL KEY str_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    INPUT 'Name': name$
    SET STRUCTURE cl, FIELD last: PARTIAL KEY name$
    PRINT cl(id); ' '; cl(last)
20  END

RNH
Name? D
80561 Derringer
```

DESCRIPTION:

This statement retrieves the first record matching the partial key in *str_expr*.

14.9.4 SET STRUCTURE: ID

FORMAT:

```
SET STRUCTURE struc_name: ID str_expr
```

EXAMPLE:

```
10  DECLARE STRUCTURE str
    OPEN STRUCTURE cl: NAME 'tti_run:client'
    ASK STRUCTURE cl: ID cl_id$
    SET STRUCTURE STR: ID cl_id$
20  EXTRACT STRUCTURE str
    END EXTRACT
    FOR EACH str
      PRINT str(#1); ' '; str(#2)
    NEXT str
30  END

RNH
20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

SET STRUCTURE: ID sets a structure to a structure ID that you have stored previously into a string variable with the **ASK STRUCTURE: ID** statement. Once you have used the SET STRUCTURE: ID statement, you can access the structure with the new structure name (STR in the example). By using these statements, you can write generalized routines when you do not know which structure you are going to access until run time.

14.9.5 SET STRUCTURE: POINTER

FORMAT:

```
SET STRUCTURE struc_name: POINTER num_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
    END EXTRACT
    SET STRUCTURE cl: POINTER 3
    PRINT cl(id); ' ' ; cl(last)
20  END

RNH
20000 Kent
```

DESCRIPTION:

This statement sets the structure to the *nth* record extracted. The statement is useful after you have done an extract, because it provides random access to any record extracted. There is no error message if there are no records extracted, or if the number given is out of range. If the number is valid, _EXTRACTED is set to 1; otherwise, it is set to 0.

14.9.6 SET STRUCTURE: RECORD

FORMAT:

```
SET STRUCTURE struc_name: RECORD num_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE ml: NAME 'tti_run:maint_log'
    SET STRUCTURE ml: RECORD 3
    PRINT ml(cost)
20  END

RNH
    $370.00
```

DESCRIPTION:

SET STRUCTURE: RECORD sets the structure to the record number given. The statement works only for structures that support relative record access.

14.9.7 SET STRUCTURE: EXTRACTED 0

FORMAT:

```
SET STRUCTURE struc_name: EXTRACTED 0
```

EXAMPLE:

```
10  OPEN STRUCTURE vend: NAME 'tti_run:vendor'
    SET STRUCTURE vend: EXTRACTED 0
20  END
```

DESCRIPTION:

Setting the number of records extracted to zero causes a new collection to be started. The **SET STRUCTURE struc_name : EXTRACTED 0** statement is used in conjunction with the **EXTRACT STRUCTURE struc_name: APPEND** statement.

14.9.8 SET STRUCTURE, SET, USING: EXPRESSION 'owner'

FORMAT:

```
SET STRUCTURE struc_name1, SET 'set_name', USING struc_name2: EXPRESSION 'owner'
```

EXAMPLE:

```
10  OPEN STRUCTURE class: name 'devint:intest_dbms'
20  OPEN STRUCTURE part : name 'devint:intest_dbms'
    SET STRUCTURE class, SET 'class_part', USING part: EXPRESSION 'owner'
30  END
```

DESCRIPTION:

This statement used for DBMS handling, fetches the owner of the structure specified by *struc_name1*. The structure specified by *struc_name2* is the owner of *struc_name1* with the set given in *set_name*.

14.9.9 SET STRUCTURE, SET: EXPRESSION 'owner'

FORMAT:

```
SET STRUCTURE struc_name1, SET 'set_name': EXPRESSION 'owner'
```

EXAMPLE:

```
10  OPEN STRUCTURE class: name 'devint:intest_dbms'
20  OPEN STRUCTURE part : name 'devint:intest_dbms'
    SET STRUCTURE class, SET 'class_part': EXPRESSION 'owner'
30  END
```

DESCRIPTION:

This statement is used for DBMS handling.

14.9.10 SET STRUCTURE, SET, USING, FIELD: KEY

FORMAT:

```
SET STRUCTURE struc_name1, SET 'set_name', USING 'struc_name2',
        FIELD field_expr: KEY str_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE class:  name 'devint:intest_dbms'
20  OPEN STRUCTURE part :  name 'devint:intest_dbms'
    SET STRUCTURE class, SET 'class_part', USING part, FIELD &
      class_code: KEY cl_code$
30  END
```

DESCRIPTION:

This statement fetches the first *struc_name1* record using the key given in *str_expr* within the set, *set_name*, where *struc_name2* is the owner.

14.9.11 SET STRUCTURE, SET, USING

FORMAT:

```
SET STRUCTURE struc_name1, SET 'set_name', USING struc_name2
```

EXAMPLE:

```
10  OPEN STRUCTURE class: name 'devint:intest_dbms'
20  OPEN STRUCTURE part : name 'devint:intest_dbms'
    SET STRUCTURE class, SET 'class_part', USING part
30  END
```

DESCRIPTION:

This statement is used for DBMS handling and fetches the first *struc_name1* within a set. The record referred to by *struc_name2* is the owner.

15 File Handling

Files are places where information is stored. You can access files from within your INTOUCH programs. Files exist outside of your program. Therefore, when your program ends, the file and the information in it still exists. The next time you run the program, you can access the file, remove old information from it and store new information in it.

Files are stored on devices: disks, tapes, etc.. They are stored under file specifications, which include a device name. For information on file names and file specifications, see Section 17.1, File Specifications and the *Command Language and DCL User's Guide* of the OpenVMS documentation set.

The following pages describe the INTOUCH statements used to manipulate files.

15.1 OPEN #chnl_num: NAME ...

FORMAT:

```
OPEN #chnl_num: NAME 'file_spec'
       [, ACCESS INPUT| OUTPUT | OUTIN ] [, UNFORMATTED]
       [, UNIQUE] [, OPTIMIZE OFF]
```

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS OUTPUT
    PRINT #1: 'This is the first line of text.'
    PRINT #1: 'This is the second line of text.'
20  CLOSE #1
30  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
    LINE INPUT #1: line_1$, line_2$
    PRINT line_1$
    PRINT line_2$
40  CLOSE #1

RNH
This is the first line of text.
This is the second line of text.
```

PURPOSE:

OPEN opens a file so you can read from it and write to it. You can also use OPEN to create a file.

DESCRIPTION:

OPEN either opens an existing file or creates a new one. *#chnl_num* is the channel number associated with the file. *chnl_num* can be any integer number in the range of 1 to 95. (0 is the channel number associated with the terminal. Channel number 0 cannot be opened or closed.) The channel number is used to refer to the file. The channel number must be unique. If a channel number is already associated with an open file, an exception is generated.

file_spec gives the file specification of the file being opened. The file specification can be any string expression.

15.1.1 OPEN Options

The OPEN statement has several options. Multiple options are separated with commas.

15.1.1.1 ACCESS Option

The **ACCESS** option specifies one of three input/output options. These options tell INTOUCH whether you want to input (read) data, output (store) data or input and output data.

ACCESS INPUT

- The file is opened for input only. Data can only be read from the file.
- If the file does not exist, an exception is generated.
- Access INPUT is the default if no ACCESS is specified.

ACCESS OUTPUT

- The file is opened for output only. Data can be stored in the file. Data cannot be read from the file.
- When a text file is opened, the default line length is 132. If you need the line length to be other than 132, you must use the SET #chnl_num: MARGIN num_expr statement.
- If the file exists, INTOUCH creates a new version of the file and uses that version. The old versions of the file are left untouched.
- If the file does not exist, INTOUCH creates a file using the specifications provided.

ACCESS OUTIN

- The file is opened for input and output. Data can be read, deleted and added to the file.
- If the file exists, INTOUCH opens the latest version.
- If the file does not exist, INTOUCH creates a file using the specifications provided.

15.1.1.2 UNFORMATTED Option

When you write to a file opened as **UNFORMATTED**, the writes are done without any carriage control. This allows various character sequences to be sent to the channel without having CR/LF (carriage return/line feed) sequences sent as well.

```
10  OPEN #1: NAME 'tt:', ACCESS OUTPUT, UNFORMATTED
20  FOR i = 1 TO 10
      PRINT #1: i;
    NEXT i
30  END

RNH 1  2  3  4  5  6  7  8  9  10
```

15.1.1.3 OPTIMIZE OFF Option

When you specify **OPTIMIZE OFF**, the file is opened without any of the INTOUCH I/O optimizations.

```
10  OPEN #2: NAME 'report.txt', OPTIMIZE OFF
20  END
```

15.1.1.4 UNIQUE Option

When the **UNIQUE** option is used, the file is created with a unique name. These are usually temporary work or holding files for listings, etc.

```
10  OPEN #12: UNIQUE, ACCESS OUTPUT
20  ASK #12: NAME x$
30  PRINT x$
40  CLOSE #12
50  END

RNH
SYS$SCRATCH:TMP_B90001.TMP
```

The following example illustrates how to create uniquely named files based on *file_spec*.

```
10  OPEN #12: NAME 'payroll', UNIQUE, ACCESS OUTPUT
20  ASK #12: NAME x$
30  PRINT x$
40  CLOSE #12
50  END

RNH
SYS$SCRATCH:PAYROLL_AE0001.TMP
```

15.2 CLOSE #chnl_num

FORMAT:

```
CLOSE [#chnl_num | ALL]
```

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.lis', ACCESS OUTPUT
20  PRINT #1: 'This is the first line.'
30  PRINT #1: 'Here is the second line.'
40  CLOSE #1
50  END
```

PURPOSE:

CLOSE #chnl_num closes a file. **CLOSE ALL** closes all files. You should close your files before your program ends.

DESCRIPTION:

CLOSE closes a file from further access. Once a file is closed, data cannot be stored in it or read from it. *chnl_num* is the channel number associated with the file. (The channel number is assigned in the OPEN statement.) The channel number can be given as any numeric expression.

15.3 PRINT #chnl_num

The **PRINT #chnl_num** statement writes data to a file so the data can be referenced at a later time.

FORMAT:

```
PRINT #chnl_num [, USING print_mask]: [TAB(col){, | ;}] [expr {, | ;}
                                      [TAB(col){, | ;}] expr...]
```

EXAMPLE:

```
10  OPEN #1: NAME 'test.tmp', ACCESS OUTPUT
    PRINT #1, USING '{UCASE}?': 'first line'
    PRINT #1: TAB(5); 'second line'
    CLOSE #1
20  OPEN #1: NAME 'test.tmp', ACCESS INPUT
    LINE INPUT #1: record_1$, record_2$
    PRINT record_1$
    PRINT record_2$
    CLOSE #1
30  END

RNH
FIRST LINE
    second line
```

DESCRIPTION:

PRINT writes data to a file. The file must be open and have a channel number associated with it. *chnl_num* is the channel number and it can be any numeric expression. *expr* is the expression being stored in the file. When INTOUCH executes a PRINT statement, it evaluates the expression and stores its value in the file. The expression is optional. A PRINT statement without an expression writes a blank line to the file.

15.4 INPUT #chnl_num: var, var...

FORMAT:

```
INPUT #chnl_num: var, var...
```

EXAMPLE:

```
10  OPEN #1: NAME 'number.dat', ACCESS OUTPUT
    PRINT #1: 1; 2; 3
    CLOSE #1
20  OPEN #1: NAME 'number.dat', ACCESS INPUT
    INPUT #1: a, b, c
    PRINT a; b; c
30  CLOSE #1
40  END

RNH
 1  2  3
```

DESCRIPTION:

The **INPUT #chnl_num** statement is used to read the data stored in a file. The file must be open and have a channel number associated with it. The simplest version of the INPUT statement is:

```
INPUT #chnl_num: var
```

chnl_num is the channel number associated with the file. The channel number can be any integer expression. *var* is the variable the data is assigned to. Each time data is input from a file, it must be assigned to a variable. The data input must match the variable's data type. INTOUCH inputs data sequentially starting at the beginning of the file.

15.4.1 Inputting Multiple Variables

One INPUT statement can be used to input data into a number of variables. The variables in the INPUT list must be separated by commas.

```
INPUT #chnl_num: var, var, var...
```

INTOUCH inputs data sequentially, starting from the beginning of the file. INTOUCH continues inputting data until all the variables in the list have values.

```
10  DIM num(10)
    OPEN #1: NAME 'number.dat', ACCESS OUTPUT
20  PRINT #1: 1
    PRINT #1: 2
    PRINT #1: 3
30  CLOSE #1
40  OPEN #1: NAME 'number.dat', ACCESS INPUT
50  FOR i = 1 TO 3
      INPUT #1: num(i)
      PRINT num(i);
    NEXT i
60  CLOSE #1
70  END

RNH
 1  2  3
```

If the variable and data types don't match, an exception will be generated.

If an attempt is made to input more data than the file contains, an exception is generated.

15.5 LINE INPUT #chnl_num: str_var, str_var...

FORMAT:

```
LINE INPUT #chnl_num [, EOF num_var]: str_var, str_var...
```

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
20  LINE INPUT #1: line_1$, line_2$
    PRINT line_1$, line_2$
30  CLOSE #1
40  END

RNH
This is the first line of text.         This is the second line of text.
```

DESCRIPTION:

LINE INPUT #chnl_num reads a line of text from a file. Everything on the line including commas, quotation marks, semi-colons, etc., is accepted as part of the input line.

The file must be open and have a channel number associated with it. The simplest version of the LINE INPUT statement is:

```
LINE INPUT #chnl_num: str_var
```

chnl_num is the channel number associated with the file. The channel number can be any integer expression. *str_var* is the variable to which data is being assigned. When INTOUCH executes the LINE INPUT statement, it reads a line from the file and assigns it to the string variable specified.

LINE INPUT accepts as the input data, everything from the beginning of the line up to the first line terminator. The contents of the line—including commas, quotation marks, tabs, leading and trailing spaces, etc.—are assigned to the string variable specified. A string variable must be specified. If a numeric variable is specified, an error will result. If the line being read is empty, INTOUCH assigns a null string to the string variable.

15.5.1 EOF Option

The **EOF** option of LINE INPUT causes INTOUCH to return a TRUE/FALSE value indicating when the end-of-file has been reached. This eliminates the need for an error handler when reading files. The format is:

```
LINE INPUT #chnl_num, EOF num_var: str_var
```

```
10  ven_ch = _CHANNEL
    OPEN #ven_ch: NAME 'test_file.tmp'
    DO
      LINE INPUT #ven_ch, EOF endfile?: datafile$
      IF  endfile?  THEN  EXIT DO
      PRINT 'line was: '; datafile$
    LOOP
    CLOSE #1
20  END
```

```
RNH
line was: This is the first line of text.
line was: This is the second line of text.
```

_CHANNEL is the next available channel number.

15.5.2 Multiple Variables

LINE INPUT can be used to read several lines of data from a file. To read more than one item, separate the string variables with commas. Lines are read sequentially, starting from the beginning of the file, and assigned to the variables listed.

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
20  LINE INPUT #1: line_1$, line_2$
    PRINT '1  '; line_1$
    PRINT '2  '; line_2$
30  CLOSE #1
40  END

RNH
1  This is the first line of text.
2  This is the second line of text.
```

If an attempt is made to input more data than the file contains, INTOUCH generates an exception.

15.6 ASK #chnl_num:

FORMAT:

```
ASK #chnl_num: [NAME str_var][, ZONEWIDTH num_var] [, MARGIN num_var]
               [, CURRENT str_var]
```

EXAMPLE:

```
10  OPEN #3: NAME 'storage.dat', ACCESS OUTPUT
20  ASK #3: ZONEWIDTH x
    PRINT 'The current print zone width is'; x
30  CLOSE #3
40  END

RNH
The current print zone width is 20
```

PURPOSE:

Use **ASK #chnl_num** to find what various characteristics a device is set to.

DESCRIPTION:

ASK returns the characteristic of the device specified and stores the value in a *num_var* or *str_var*. *chnl_num* is an optional channel number. If no channel number is specified, INTOUCH checks the default device. If a channel number is specified, INTOUCH checks the device associated with that channel number.

15.6.1 ASK Options

An option must be included in the **ASK #chnl_num** statement. The ask options currently available are described below.

ZONEWIDTH num_var

ASK #chnl_num: ZONEWIDTH finds the print zone width of the device specified and assigns the value to the numeric variable *num_var*.

```
10  OPEN #3: NAME 'storage.dat', ACCESS OUTPUT
20  ASK #3: ZONEWIDTH x
    PRINT 'The current print zone width is'; x
30  CLOSE #3
40  END

RNH
The current print zone width is 20
```

MARGIN num_var

ASK MARGIN finds the right margin of the device specified and assigns its value to the numeric variable *num_var*.

```
10  OPEN #3: NAME 'test_file.tmp', ACCESS OUTPUT
20  ASK #3: MARGIN marg
    PRINT 'The current margin is'; marg
30  CLOSE #3
40  END

RNH
The current margin is 132
```

CURRENT str_var

ASK #chnl_num: CURRENT is used to store a current record value into the str_var.

```
10  OPEN #1: NAME 'temp.tmp', ACCESS OUTPUT
    FOR z = 1 TO 20
      PRINT #1: 'This is line number '; z
    NEXT z
    CLOSE #1
20  OPEN #1: NAME 'temp.tmp'
    FOR i = 1 TO 5
      LINE INPUT #1: a$
    NEXT i
    ASK #1: CURRENT c$
    PRINT '5th item was: '; a$
30  FOR i = 1 TO 5
      LINE INPUT #1: a$
    NEXT i
    PRINT '10th item was: '; a$
40  SET #1: CURRENT c$
    LINE INPUT #1: a$
    PRINT 'Back to 5th item again: '; a$
50  CLOSE #1
60  END
```

```
RNH
5th item was: This is line number  5
10th item was: This is line number  10
Back to 5th item again: This is line number  5
```

NAME str_var

ASK #chnl_num: NAME asks the INTOUCH operating system for the file specification of the file open on channel *#chnl_num* and stores the value into *str_var*.

```
10  out_ch = 12
20  OPEN #out_ch: NAME 'sys$scratch:minutes.lis', &
        ACCESS OUTPUT
30  ASK #out_ch: NAME x$
40  PRINT x$
50  CLOSE #out_ch
60  END

RNH
USER:[TESTER]MINUTES.LIS
```

15.7 SET #chnl_num: expr

FORMAT:

```
SET # chnl_num: [ ZONEWIDTH num_expr ] [, MARGIN int_expr]
                [, CURRENT str_expr]
```

EXAMPLE:

```
10  OPEN #3: NAME 'storage.dat', ACCESS OUTPUT
20  ASK #3: ZONEWIDTH x
    PRINT 'The current print zone width is'; x
    SET #3: ZONEWIDTH 35
    ASK #3: ZONEWIDTH x
    PRINT 'The new print zone width is'; x
30  CLOSE #3
40  END

RNH
The current print zone width is 20
The new print zone width is 35
```

DESCRIPTION:

SET #chnl_num sets various device characteristics. *chnl_number* is a channel number. If no channel number is specified, INTOUCH sets the default device (the terminal). If a channel number is specified, INTOUCH sets the device associated with that channel number.

When a device characteristic is SET, it remains set until the device is closed. Therefore, if the terminal is SET, it will remain SET until you exit from the INTOUCH environment or the SET statement is used again.

15.7.1 SET #chnl_num: Options

An option must be included with the **SET #chnl_num** statement. The set options currently available are described below:

ZONEWIDTH num_expr

SET #chnl_num: ZONEWIDTH sets the print zone width of the device specified to the number designated. *num_expr* indicates the width to set the device's print zones. See above example.

MARGIN int_expr

SET #chnl_num: MARGIN sets the right margin on the device specified to the number indicated. *int_expr* specifies the column to set the margin to. The margin must be greater than the zonewidth.

```
10  OPEN #3: NAME 'storage.dat', ACCESS OUTPUT
20  SET #3: MARGIN 45
    PRINT #3: REPEAT$('1234567',10)
    CLOSE #3
30  OPEN #3: NAME 'storage.dat', ACCESS INPUT
    DO
      LINE INPUT #3, EOF endfile?: item$
      IF  endfile?  THEN  EXIT DO
      PRINT item$
    LOOP
40  CLOSE #3
50  END

RNH
123456712345671234567123456712345671234567123
456712345671234567123456
```

CURRENT str_expr

SET #chnl_num: CURRENT sets the current record to that specified by *str_expr*. The *str_expr* contains the information for the record you want to make current.

```
10  OPEN #1: NAME 'temp.tmp', ACCESS OUTPUT
    FOR z = 1 TO 20
      PRINT #1: 'This is line number '; z
    NEXT z
    CLOSE #1
20  OPEN #1: NAME 'temp.tmp'
    FOR i = 1 TO 5
      LINE INPUT #1: a$
    NEXT i
    ASK #1: CURRENT c$
    PRINT '5th item was: '; a$
30  FOR i = 1 TO 5
      LINE INPUT #1: a$
    NEXT i
    PRINT '10th item was: '; a$
40  SET #1: CURRENT c$
    LINE INPUT #1: a$
    PRINT 'Back to 5th item again: '; a$
50  CLOSE #1
60  END

RNH
5th item was: This is line number  5
10th item was: This is line number  10
Back to 5th item again: This is line number  5
```

15.8 Deleting Files

15.8.1 KILL

FORMAT:

```
KILL str_expr
```

EXAMPLE:

```
5   ! DANGER -- Executing this program will DELETE a file

10  INPUT 'What file do you want to delete': file$
20  KILL file$
30  END

RNH
What file do you want to delete?
```

PURPOSE:

Use **KILL** to delete a file from within your program.

DESCRIPTION:

KILL searches for and deletes the file specified in *str_expr*, the string expression. You must include the full file name with extension. If you do not include a version number, KILL will delete the most recent version only.

16 Creating Structures, Field Definitions with SETUP

16.1 SETUP–Overview

INTOUCH works with data in any record management system. SETUP tells INTOUCH where data files are located and what record management systems are used. Once you have defined your structures with SETUP, you can use INTOUCH's structure statements to manipulate the data.

SETUP now supports the following database engines:

RMS
Rdb
DBMS
USERBASE
POISE
FASTFILE
S1032
INGRES
dBASE III
ADABAS
ORACLE

Procedures

The following procedures are available in SETUP:

- **DEFINE**

  Define fields in a structure. Use DEFINE when you want to add new fields, modify existing fields or delete fields.

- **SHOW**

  Show field definitions or structure information. Use SHOW to display or print lists of the fields defined for this structure or to display information on this structure.

- **MODIFY**

  Modify the structure file. Use MODIFY to change the structure file and/or the structure security and access levels.

- **CREATE**

  Create a DATA File.

- **EXIT**

  Exit the INTOUCH SETUP routine. EXIT will save the work you have done on the structure and exit you back to the INTOUCH prompt.

Each of the SETUP procedures is described in detail on the following pages.

16.1.1 Entering SETUP

To enter SETUP, type **CALL SETUP** at the INTOUCH prompt.

Example 16–1: To Use SETUP

```
INTOUCH
CALL SETUP
```

SETUP will ask you a number of questions including:

- structure name
- database engine (record system being used)
- dataset (data file name)
- data dictionary (definition file name)

Example 16–2: SETUP Prompts

```
SETUP V4.2              Set Up INTOUCH 4GL Data Structures

Structure name :
Database engine:
Dataset        :
Data dictionary:




Structure name? _____________________________________________________________

EXIT = Exit                                              \ = Back  HELP = Help
```

You can create a new structure by entering new file names or you can modify existing structures by selecting them at the prompts.

Once you have specified the structure you want to work with, you can define fields in the structure or display field definitions.

16.2 SETUP Prompts

This section describes the SETUP prompts and your options for answering them.

When you are using SETUP and are at any of the prompts, you can enter **HELP** or press the Help key to get information about the prompt.

Default answers are provided for many of the prompts. If you want to accept the answer provided, press the Return key at the prompt. If you do not want to use the default, type your answer over the default.

If you want to go back to a previous prompt, use the "\" (backslash key) to back up.

16.2.1 Structure Name

Enter a structure name. This can be a new name or the name of an existing structure. If you are creating a new structure, this will be the name of the structure file. The default extension for the file is .STR. The structure file contains:

- the name of the database engine (record management system).
- the name of the dataset (data file).
- the name of the data dictionary (definition file).

Your structure name can be any VMS file specification. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

Structure name? CUSTOMER

16.2.2 Database engine

Use the arrow keys to select the database engine (record management system) used by this structure. INTOUCH supports these systems:

RMS
RDB
DBMS
USERBASE
POISE
FASTFILE
S1032
INGRES
DBASE3
ADABAS
ORACLE

(You can refer to the *INTOUCH Interfaces to Database Engines* manual for more information.)

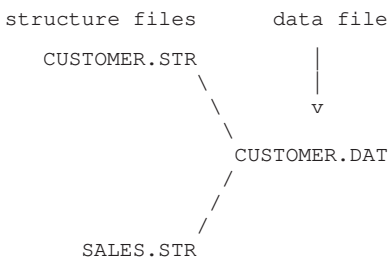
If you are creating a new structure, the default will be RMS. If you are modifying an existing data structure, the default will be the database engine (record management system) specified in the structure file. FASTFILE is an INTOUCH record management system designed for speed.

Database engine? RMS

16.2.3 Dataset

Enter the name of the data file (dataset) you will be using with this structure. If this is a new structure, the default data file name will be the same as the structure name. The default extension is .DAT.

You can use the same data file for a number of structures. For instance, you might use the CUSTOMER data file for both the CUSTOMER and the SALES structures:



Your dataset name can be any VMS file specification. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

Dataset? CUSTOMER.DAT

Rdb dataset

If the database engine is **Rdb**, the input format will be different. An Rdb database consists of tables. Each structure corresponds to one table. For Rdb, the dataset is the name of a table in the Rdb database.

```
FORMAT:         table_name IN rdb_database
EXAMPLE:        employee in rdb_data:personnel
```

DBMS dataset

For a **DBMS** dataset, you need to specify the DBMS record name and the name of the DBMS root file. Optionally, you can add the name of the subschema.

If the subschema is not given, the DEFAULT_SUBSCHEMA is assumed.

```
FORMAT:         record_name IN [subschema_name FOR ] root_file
EXAMPLES:       part in datadisk:[parts]dbmparts
                part in ss0001 for datadisk:[parts]dbmparts
```

INGRES dataset

For an **INGRES** dataset, you need to specify the INGRES table name. The INGRES database to use is specified by the symbol INT_INGRES_DBNAME.

```
FORMAT:         table_name
EXAMPLE:        employee
```

ADABAS dataset

For an **ADABAS** dataset, you need to specify the file number and the database number.

```
FORMAT:         file_number IN database_number
EXAMPLE:        6 IN 1
```

ORACLE dataset

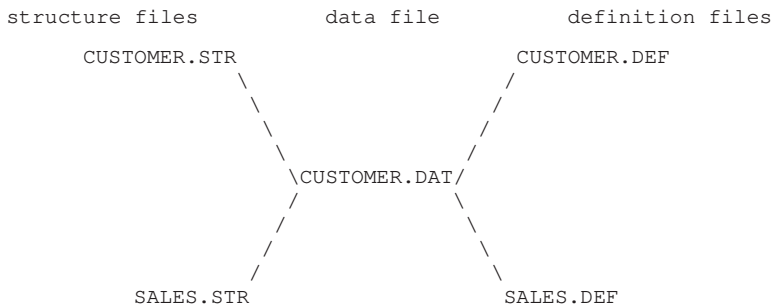
For an **ORACLE** dataset, you need to specify the ORACLE table name.

```
FORMAT:         table_name
EXAMPLE:        emp
```

16.2.4 Data dictionary

Enter the name of the definition file you will be using with this structure. If you are creating a new structure, the default definition file name will be the same as the structure name. The default extension is .DEF.

As with the data file, you can use the same definition file for a number of structures. In fact, you can create any combination of definition, data and structure files:



Your definition file name can be any VMS file specifications. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

Data dictionary? CUSTOMER

There are three special definition file names allowed. If the database engine for this structure can supply the field information and you want to use these native definitions, enter **NONE** as the definition file name and INTOUCH will use the definitions embedded in the database engine with the fields.

Data dictionary? NONE

These database engines can supply the field definitions:

Rdb
DBMS
dBASE III
INGRES
ADABAS
ORACLE

If you are using a database engine that supports native field definitions (Rdb, for example) and you would like to augment the native definitions with additional definitions, you can enter a definition file name prefixed with **AUG>**. This will create an augmented definition file. The native field names will be available along with any fields defined in the augmented definition file. Typically, augmented definitions are used to supply report headings, prompt text and print masks.

For example, if you have an Rdb table which contains of the following native fields:

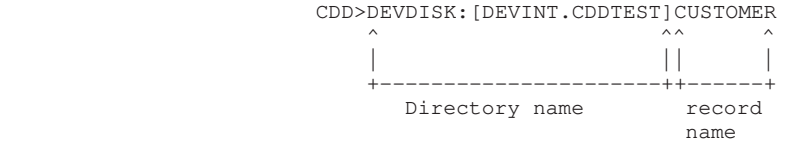
```
Name            Start           Length
-----           -----           ------
fielda            1               10
fieldb           11               5
fieldc           16               10
```

and you want to define a field (an index) consisting of "fielda" and "fieldb", you can create an augmented data dictionary and define the index you want.

Data dictionary? AUG>RDBTEST.DEF

If you are using **CDD**, you can enter **CDD>** and a name of a CDD/Plus Directory and recordname:

Data dictionary? CDD>DEVDISK:[DEVINT.CDDTEST]CUSTOMER



CDD/Plus access is read only at this time.

After the structure information has been entered, the SETUP Procedure Menu will be displayed and you can select the SETUP procedure you want to perform.

16.3 SETUP Procedures

Example 16–3: SETUP Procedures

```
SETUP V4.2              Set Up INTOUCH 4GL Data Structures

Structure name :  CUSTOMER.STR
Database engine:  RMS
Dataset        :  CUSTOMER.DAT
Data dictionary:  CUSTOMER.DEF


                        +---Setup Procedure---+
                        |  Define fields      |
                        |  Show             [>|
                        |  Modify structure [>|
                        |---------------------|
                        |  Create data file   |
                        |  EXIT               |
                        +---------------------+

EXIT = Exit                                        \ = Back  HELP = Help
```

The following procedures are available in SETUP:

- **Define fields** - Define the fields in the data record. Define allows you to add, modify and delete field definitions.
- **Show** - Show field definitions in either a full or brief format, list the defined fields or show structure file information.
- **Modify structure** - Change the structure file or change the structure security and access levels.
- **Create data file** - Creates the data file.

Note

Depending on the database engine chosen and the definition file selected, you might not have all of these options.

To select the SETUP procedure you want to perform, use the arrow keys to highlight your selection and then press Return. You can also enter in the first few characters and press Return.

16.4 DEFINE Procedure

You can use the DEFINE procedure to define the fields in a structure. The DEFINE procedure is also used to modify and delete field definitions.

When the **Define fields** menu option is selected, the field definition input screen is displayed.

Example 16–4: Field Definition Input Screen

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :
Classification  :
Description     :
Data type       :
First position  :          length:          Occurs:
Semantics       :

Prompt text     :
Report heading  :
Print mask      :
Screen mask     :
Help text       :
Access rules    :
Validation rules:


Field name? ____________________________________

                        Enter -fieldname to delete a field
EXIT = Exit                                          \ = Back  HELP = Help
```

You will be asked first for the name of the field you wish to add, modify or delete. You will then be prompted for information about the field.

16.4.1 Field name

Enter the field name you wish to add, modify or delete. If you are adding a new field, the field name will be used to identify this field. If there are existing fields, you can use one of the **SHOW** options to get a list of the current field names.

If you wish to delete a field, enter a minus sign (**-**) followed by the field name. For example:

```
Field name? -phone
```

The field name:

- must begin with a letter.
- can consist of numbers, letters, underscores and dollar signs.
- can be up to 32 characters long.

- The last character of the field name can be a "$" or "%".

If you are modifying a field, the defaults for the rest of the DEFINE prompts will be your original responses. You can keep these defaults by pressing the Return key at the prompt. You can change the defaults by typing over the responses.

16.4.2 Classification

Enter the field class or press Return to accept the default of **none**.

The classification is optional.

Classification refers to a field class in the CLASSIFICATION data dictionary. Field classes are pre-defined commonly used data fields.

Using Classifications

There are cases when you use the same field type in a record or in several datasets. When this occurs, setting up a field class will save time in defining fields and help to ensure data integrity.

If, for example, you had a home phone field, work phone field and alternate phone field in your record, you would need to define the three phone fields using different field names but the same characteristics. The phone fields would all be 10 numeric characters in length and be printed in the format (123) 456-7890.

To ensure that all the phone numbers are defined exactly the same, you could define a phone *class* in the CLASSIFICATION data dictionary. The class name could be PHONE and be defined with the length (10), semantics (num), print mask ((###~)~ ###~-####) and validations (digits; minlength 10).

Then, whenever you want to define a phone number type field, you would enter the field name, HOME_PHONE for example, with PHONE as the classification and the classification data would automatically become the default answers to the remaining DEFINE prompts. If you did not want to use a particular default, you can simply type in a different answer at the prompt.

You would also be able to use the PHONE classification when defining phone fields in other dataset records.

Before you can use field classifications, you must first define the field classes in the CLASSIFICATION data dictionary.

Creating Field Classes

Field classes are defined the same way as any other data field. However, these fields are located in a specific data file called TTI_RUN:CLASSIFICATION. To create a field class, perform the following steps:

1. Get into SETUP.
2. Enter **TTI_RUN:CLASSIFICATION** as the structure name. The following information will be displayed:

```
Structure name :  TTI_RUN:CLASSIFICATION.STR
Database engine:  RMS
Dataset        :  NONE
Data dictionary:  TTI_RUN:CLASSIFICATION.DEF
```

3. Select the *Define fields* procedure.
4. At the field name prompt, enter the field class name you want to use (i.e. PHONE).
5. Go through each of the prompts and set up the characteristics you want this field class to have.

You can set up as many field classes as you wish.

Using an Alternate Classification File

If you do not want to use the provided TTI_RUN:CLASSIFICATION data file, you can create one of your own. For example, you could create a structure such as CLASS_NAMES for your field classes. However, if you choose to do this, you **MUST** define the file as follows:

DEFINE TTI_CLASSIFICATION device:[directory]CLASS_NAMES.DEF

SETUP will then retrieve classification data from CLASS_NAMES instead of TTI_RUN:CLASSIFICATION.

Generated Defaults

After the classification prompt is answered, SETUP displays either the class defaults or defaults generated by SETUP to assist you in defining the field.

Example 16–5: DEFINE Screen showing field class generated defaults

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :  HOME_PHONE
Classification  :  PHONE
Description     :  Home phone
Data type       :  CH
First position  :  14      length:  10      Occurs:  1
Semantics       :  NUM, RJ

Prompt text     :  Home phone
Report heading  :  Home Phone
Print mask      :  (### )  ### -####
Screen mask     :  none
Help text       :  none
Access rules    :  NORMAL
Validation rules:  digits; minlength 10


Description? Home phone_____________________

EXIT = Exit                                          \ = Back  HELP = Help
```

Example 16–6: DEFINE Screen showing SETUP generated defaults

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :  HOME_PHONE
Classification  :  none
Description     :  Home phone
Data type       :  CH
First position  :  14      length:          Occurs:  1
Semantics       :  UC

Prompt text     :  Home phone
Report heading  :  Home Phone
Print mask      :  none
Screen mask     :  none
Help text       :  none
Access rules    :  NORMAL
Validation rules:  none


Description? Home phone_____________________

EXIT = Exit                                          \ = Back  HELP = Help
```

16.4.3 Description

Enter a description of the field. The description can consist of any characters on the keyboard and be up to 30 characters long.

If you are adding this field, the default description is the field name with any "$", "%", "_" characters changed to spaces.

The description will be used as the default prompt text.

16.4.4 Data type

Enter the data type for this field. The data type is the internal representation of the data. The most common type is **CH** (character) which is the default. The data type will be one of the following:

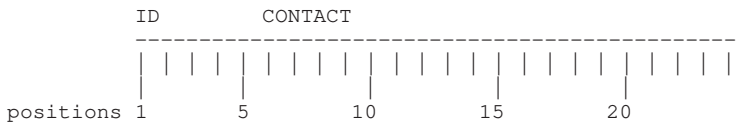
AC	ASCII counted string
AP	Pointer to ASCII counted
C3	COBOL comp-3, also known as packed decimal; the C3 data type also supports the VAX DECIMAL data type
CH	Character
DS	Date stamp (length=8)
EB	EBCDIC
FL	Floating point
GF	G-Float
IN	Integer (signed)
IU	Integer (unsigned)
PF	Packed floating
PZ	Packed zipcode
QS	Quadword (signed)
RO	Right overpunch
RS	Right sign separate
UN	Undefined
ZE	Zoned EBCDIC
ZN	Zoned numeric; used by DIBOL

The DATE (DS) attribute is currently only used by applications and has no effect within INTOUCH.

16.4.5 First position

Enter the position at which this field will start. Character positions are numbered sequentially. For example:

Figure 16–1 Character Positions



In the above diagram, the ID field starts at position 1 and ends at position 5. The CONTACT field starts at position 6, etc.

Fields generally follow each other directly. Therefore, the first position of a new field is usually one more than the last position used. If the last position used was ten, the next field would start at eleven. INTOUCH automatically defaults to the correct sequential position.

Defining existing RMS file fields

If you are defining fields for an existing RMS file, you must keep in mind that the starting position of an RMS file record is 0 and in INTOUCH, the starting position is 1. (RMS is 0 oriented, whereas INTOUCH is 1 oriented.) For example, if the RMS file field starts in position 0, when you define this field in INTOUCH, you must define the field as starting in position 1. If the RMS field starts in position 20, you would use 21 as the starting position, etc.

16.4.6 Field length

Enter the length of this field. The length should be the number of character positions this field will contain. Fields can be any length. The length must be given as an integer number.

You can also enter the ending position for the field length. You enter "P" followed by the ending position. For example: P5

Variable Length Fields

If a CH or UN data type field has a first position that starts at or before the end of a logical record, that field will have its length adjusted upon reference. The new length will reflect the actual end of the logical record.

For example, a field could start in position one, and have its length set to the maximum record length. When the field is referenced, its length will be adjusted to the actual record length. This field is variable length.

16.4.7 Number of occurrences

Enter the number of times that this field occurs in the data record. The default number of occurrences is "1".

If the field being defined is an array, enter the number of elements in the array.

16.4.8 Semantics

Enter the semantics (attributes) that apply to this field. The semantics describe what type of data will be in the field. For example, if this is a dollar amount field you would select NUM:2 which means that this is a numeric field with two decimal places.

The semantics are entered as a list of descriptors separated by commas. When you have completed your list, enter GOLD/F (i.e. press [PF1] and then the [F] key).

The valid semantics are:

NUM	Indicates a numeric field which allows only numbers. The semantic *RJ*, right-justified, is automatically appended when NUM is selected.
NUM:scale	SCALE specifies the scale factor—how many decimal places to use. Scale factors can be positive or negative (from -99 to 99).
RJ	The data will be right-justified in the field.
UC	All letters will be stored in upper-case format. Additionally, field comparison data will be upper-cased prior to any comparison.
ZF	The data will be zero filled.
ZS	Causes all leading zeros to be suppressed
DATE[:YMD|MDY]	This field contains a date. The field must be six or eight digits long. The default date format is YMD. You can enter DATE or optionally, specify the date format. The date format is currently only used by applications. INTOUCH does NOT automatically switch date values into or out of the format specified.
FULLTIME	Indicates that the field contains a date-time stamp and the time value is to be preserved. By default, INTOUCH stores/retrieves only the date portion of a date stamp field.
APPLICATION:name	This is a user defined semantic and is not used by INTOUCH. A user can assign a name to indicate that the field is related to a specific application (i.e. accounting). Application programs can then ask for the application name and perform some action based on the data.
DBFLD:fieldname	This is a special semantic for augmented definition files only. The DBFLD semantic is used to associate an augmented field definition with an actual database field. Fieldname is the actual name of the field in the database.

If you do not want any of the semantics, you can enter NONE.

If you are modifying an existing field, the default will be the semantics you have previously specified.

16.4.9 Prompt text

Enter the prompt text for this field. The prompt text will be used when data is entered for this field. The prompt text:

- can be any characters on the keyboard.
- can be up to 30 characters long.

If you are adding this field, the default will be the *description* you entered for this field.

16.4.10 Report heading

Enter the heading you want to appear on reports. This heading will appear at the top of the report column containing this field's data. If you are adding a new record, the default report heading is the *field name* you entered.

The heading can be up to 30 characters long.

To make a multi-line heading, separate the heading for each line with a comma (i.e. Cust,No).

16.4.11 Print mask

Enter a print mask for this field. The print mask is used to display information entered into this field. The print mask will be used whenever you use INTOUCH's PRINT statement to print this field's information, or when you use one of the INTOUCH programs such as Guided Query Language.

Print masks are created with the #, @ and % characters. SETUP will automatically provide the tilde (~) character for any literal (non-masking) characters found in the entered mask. Some examples:

```
                          Example 1              Example 2
                          ----------------       -------------
Data to be masked:        AB1234XY               AB1234XY
Data to print as :        (AB-1234-XY)           AB 1234 XY
Mask you enter   :        (##-####-##)           ## #### ##
SETUP displays   :        (##~-####~-##~)        ##~ ####~ ##
```

You can refer to the chapter "Printing and Displaying Data" for more information on print masks.

16.4.12 Screen mask

The screen mask is used by applications to enter data into this field. A program can ask for the screen mask and use it in whatever manner that is desired. The format should be the same as for the print mask.

16.4.13 Help text

HELP text can be entered for this field. The help text can be displayed during data entry if the user types "HELP" or presses the Help key at this field prompt when in **MAINTAIN**. Custom applications can also display this help text.

The help text:

- can be up to 60 characters long.
- can consist of any characters on the keyboard.

16.4.14 Access rules

Enter the access rules for this field. The access rules determine who will have access to this field. NORMAL allows users to read, add and change field information.

Access rules have the following formats:

READ:read_access, WRITE:write_access

NORMAL

where *read_access* and *write_access* are each represented by a single letter from (A-Z).

NORMAL denotes READ:N, WRITE:N.

A is the most restricted access level. **Z** is the least restricted access level.

The access_rules are used in conjunction with the structure security level to determine whether or not a user can access the field.

When a user attempts to access a field, the structure security level is compared to the read or write access level defined for the field. If the field's access level is lower than the structure security level, access is allowed. For example:

> If the structure READ access is J and the field READ access is T, the field data CAN be read. However, if the structure WRITE access is J and the field WRITE access is E, the field CANNOT be written to or changed.

16.4.15 Validation rules

You can enter validation rules for this field. The validation rules will be used to validate data entered by a data entry program.

The data entry program can access these validation rules to make sure that the data entered is valid prior to storing the data into the field.

Any rule that is valid to the INTOUCH VALID function can be entered here.

The validation rules are entered as a list of items separated by semicolons. For example:

```
required; digits; minlength 10
```

When you have completed your list, enter GOLD/F (i.e. press PF1 and then the F key).

Please refer to Appendix A of this manual for detailed information on the VALID function.

When you have answered all the field prompts, the field definition will be added to the data dictionary. The screen will clear and you can define the next field or enter "\" or "EXIT" to return to the SETUP procedures menu.

16.4.16 Field Definitions and Key Fields — General Information

Normally, when you define the record fields, each defined field represents a single piece of data in the record. This allows your programs access to all of the record data.

There are times when you want to process two or more adjoining fields. For example, if you have a last name field, first name field and initial field and you want to sort the records by name, you would sort the three separate fields–last name, first name and initial.

To simply this process, you can define an additional field, FULL_NAME, which can consist of the three fields. When you define the FULL_NAME field, the starting position would be the first position of the last name and the length would be the total number of characters in the three fields. Then, when you want to sort or print the full name, you need only refer to one field, the FULL_NAME field.

Redefining fields becomes very useful when you need to set up unique key fields for RMS indexed files.

Key Fields

If you are creating an RMS indexed file, you will be designating one or more fields as *key fields*. RMS indexed files require at least one key field. When you use the CREATE procedure to create an RMS indexed file, you will be prompted for a *primary key* field and *alternate key* fields.

Key fields contain unique data which enables programs to locate records very quickly. For example, the customer number field could be a key field because all the customer numbers in all the records would be different. This type of key field is called a *unique key field*.

Duplicate key fields contain unique types of data but the data might be found in more than one record in the file.

Any defined or redefined field can be designated as a key field. However, it is best to use fields that normally will contain unique data as key fields. Common practice is to set up the first field as a key field.

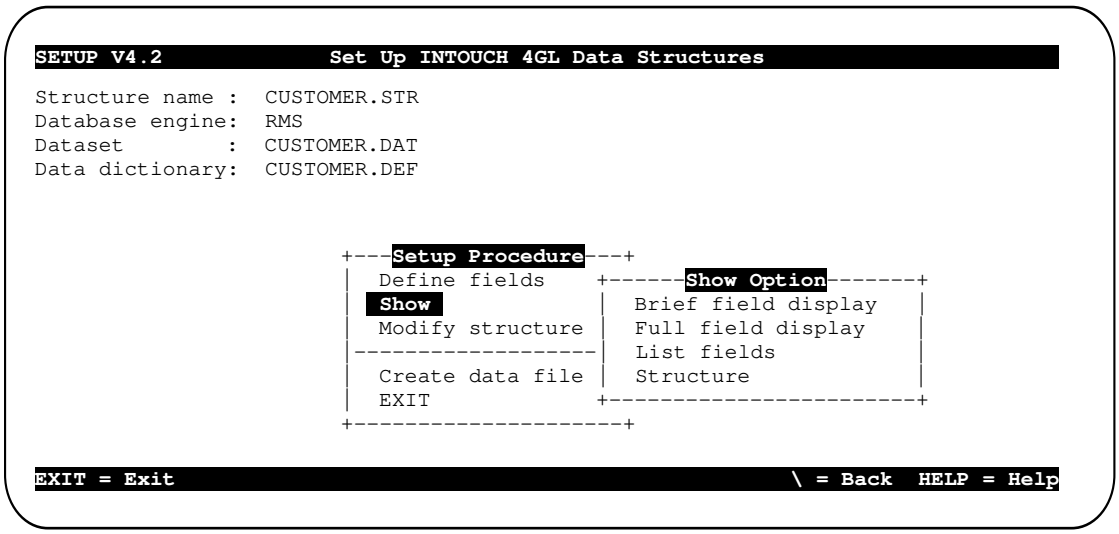
A key field can be defined which consists of two or more fields. If the fields which make up this key field are not adjoining (i.e. one after the other), this is called a *segmented key field*. Key fields consisting of two or more fields, whether segmented or not, must have a length that equals the total length of all the segments that make up the key field.

16.5 SHOW Procedure

The SHOW procedure displays information about the fields defined in the data dictionary and also displays structure information.

You can use the arrow keys to select the **Show** procedure from the SETUP procedures menu.

:



The SHOW procedure has four options:

- The **Brief field display** creates a listing of the fields showing some of the definition information.
- The **Full field display** creates a listing of the fields showing all of the definition information.
- The **List fields** option creates a listing of the defined fields.
- The **Structure** option displays information about the structure file.

SETUP creates the listings in report format so that you can display the field and structure information on your screen, print a hardcopy or save the information on disk.

After a report is created, you can select an output option from the Output Options menu. The default is **Screen** which will display the report to your screen. The line under the Output Options menu tells you the name of the temporary report file that was created.

```
+--------------------- Output Options ---------------------+
| Screen          |  System Printer   |  Laser Printer   |
| Printer Port    |  Save to Disk     |  EXIT            |
+--------------------------------------------------------+

Ready to print user:[tester]setup_show_definition_210006.tmp
```

If you are in the middle of displaying a report and want to exit out of the display, enter CTL/Z and you will return to the Output Options menu.

To return to the SETUP procedures menu, you can use the arrow keys to select the EXIT option or press "\" (backslash key).

16.5.1 Brief Field Display

The Brief Field Display shows the field name, description, first position of the field, field length, number of occurrences, last field position, data type, print mask and semantics.

Example 16–7: SHOW - Brief Field Display Example

```
08-Mar-1995            Definitions for: CUSTOMER.DEF                 Page:   1

Field Name      Description       First  Length  Occurs  Last  Dtype  Print Mask
--------------- ----------------- -----  ------  ------ -----  -----  ----------
CUSTNBR         Customer number       1       5      1      5  CH     none
      Semantics  : none

NAME            Customer name         6      25      1     30  CH     none
      Semantics  : none

ADDR1           Address line 1       31      25      1     55  CH     none
      Semantics  : none

ADDRESS         Address array        31      25      1     55  CH     none
      Semantics  : none
```

16.5.2 Full Field Display

The Full Field Display shows all of the definition information.

Example 16–8: SHOW - Full Field Display Example

```
08-Mar-1995             Definitions for: CUSTOMER.DEF                  Page:   1

Field Name      Description       First  Length  Occurs  Last  Dtype  Print Mask
--------------- ----------------- -----  ------  ------ -----  -----  ----------
CUSTNBR         Customer number       1       5       1      5  CH     none
      Semantics  : none
      Prompt text: Customer number
      Heading    : Cust#
      Help       : none
      Screen mask: none
      Validations: none

NAME            Customer name         6      25       1     30  CH     none
      Semantics  : none
      Prompt text: Customer name
      Heading    : Customer name
      Help       : none
      Screen mask: none
      Validations: none
```

16.5.3 List Fields Display

The List Fields Display shows the field name, description, first position of the field, field length and data type.

Example 16–9: SHOW - Fields List Display Example

```
08-Mar-1995                 CUSTOMER.DEF - FIELDS                    Page:   1

Field Name                        Description                     First Length  DT
--------------------------------- ------------------------------- ----- ------  --
CUSTNBR                           Customer number                     1      5  CH
NAME                              Customer name                       6     25  CH
ADDR1                             Address line 1                     31     25  CH
ADDRESS                           Address array                      31     25  CH
ADDR2                             Address line 2                     56     25  CH
CITY                              City                               81     20  CH
STATE                             State                             101      2  CH
ZIP                               Zip                               103      5  CH
```

16.5.4 Structure Display

The Structure option of SHOW displays information about the structure and security access levels.

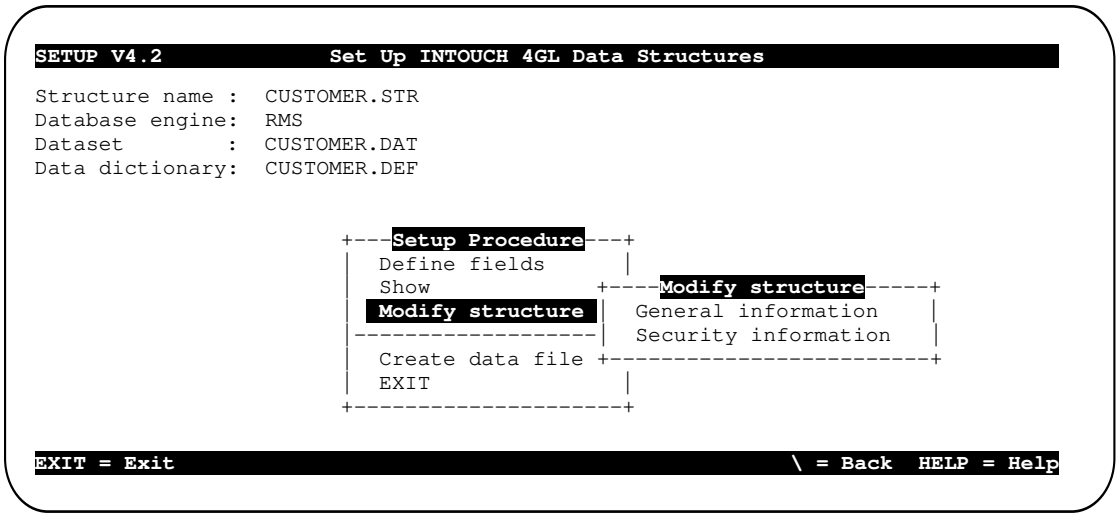
Example 16–10: SHOW - Structure Display Example

```
08-Mar-1995                Structure: CUSTOMER.STR                      Page: 1

Structure name : CUSTOMER.STR
Dataset        : CUSTOMER.DAT
Record system  : RMS
Data dictionary: CUSTOMER.DEF
Access levels  : SECURITY:N, READ:N, WRITE:N, UPDATE:N, DELETE:N
```

16.6 MODIFY STRUCTURE Procedure

The MODIFY STRUCTURE procedures allow you to:

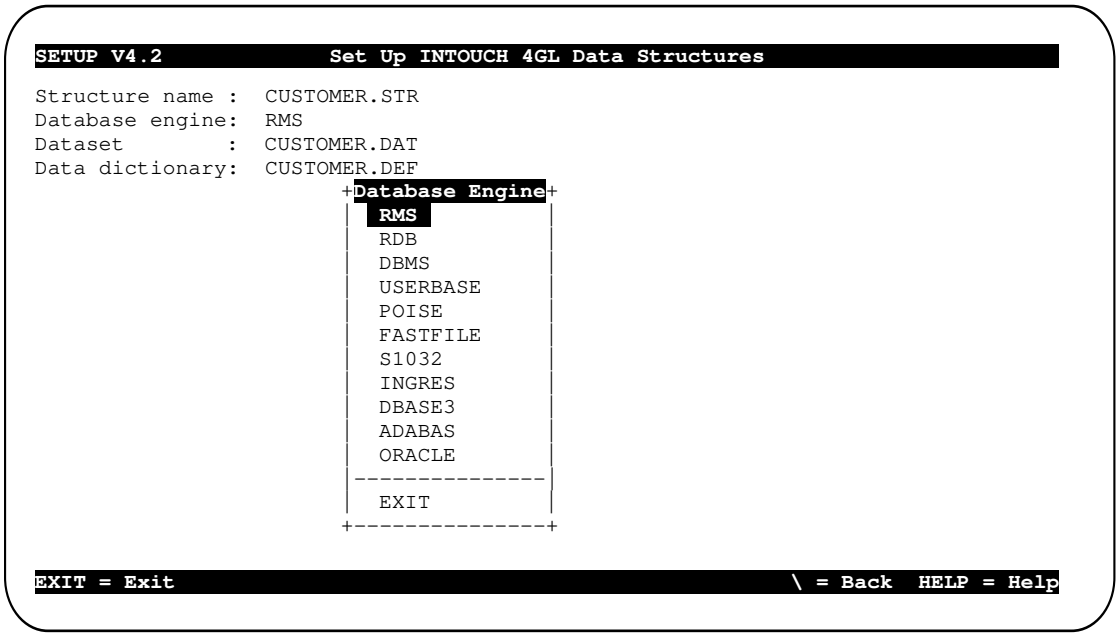
- change the general structure information.
- change the structure security level information and the read, write, update and delete access levels.



You can use the arrow keys to select the **Modify structure** procedure from the SETUP procedures menu and then select one of the two modify structure options.

16.6.1 Modify Structure General Information

When you select the modify structure **General information** option, SETUP will display the current structure information.



You will be prompted for:

- Database engine
- Dateset
- Data dictionary

You can either press Return at the prompts to keep the current information or enter new information. If you need clarification on any of the prompts, you can refer back to the **SETUP Prompts** section in this chapter.

After the last prompt is answered, the **STR** file will be updated with the new information.

You will be returned to the SETUP procedures menu where you can select another option or exit SETUP.

16.6.2 Modify Structure Security Information

When you select the modify structure **Security information** option, SETUP will display the current security information.

```
SETUP V4.2            Structure Security and Access Levels

Structure security level:  N
Read   access level     :  N
Update access level     :  N
Write  access level     :  N
Delete access level     :  N




Structure security level? N

                           Security levels are A to Z
EXIT = Exit                                          \ = Back  HELP = Help
```

You will be prompted for:

- Structure security level information
- Read, update, write and delete access level information

You can either press Return at the prompts to keep the current information or enter new information.

Structure security level

Enter the security level associated with this structure.

The security level is a single letter in the range of **A** to **Z**.

Security level **A** is the highest level of security. Security level **Z** is the lowest level of security.

Read access level

Enter the access level required to *read* a data record in this structure.

The *read access level* is a single letter in the range **A** to **Z**. **A** is the most restricted access level. **Z** is the least restricted.

When an attempt is made to read a record from this structure, INTOUCH will compare the structure security level to the read access level. If the read access level is lower than the structure security level, the read request will be allowed.

For example:

Structure level	Read access level	Access
K	R	Allowed
K	E	Denied
K	K	Allowed

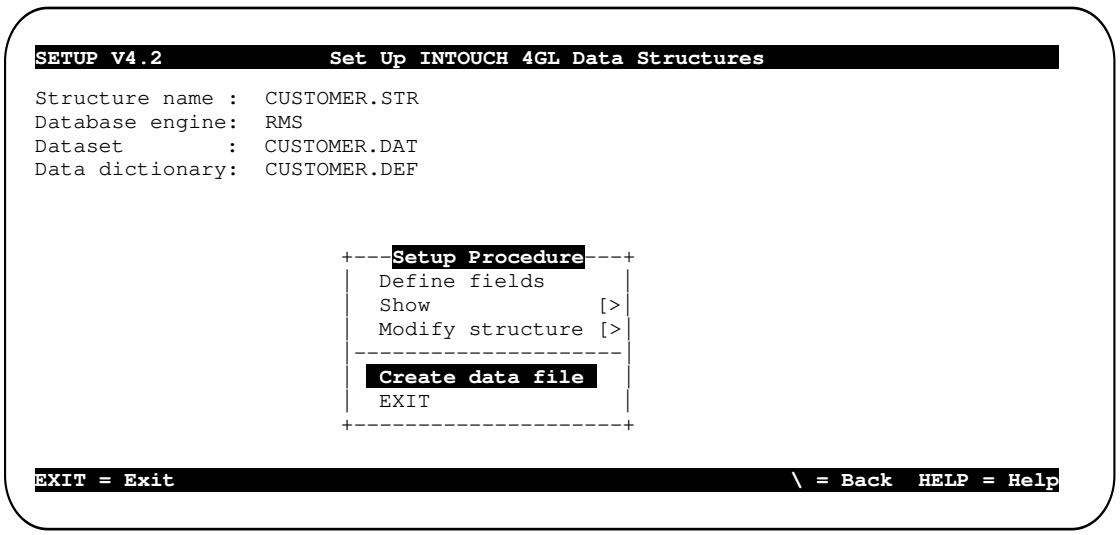
Update, Write, Delete access levels

Enter the access levels required to *update, write, delete* data records in this structure. Answer each of the prompts with a letter from **A** to **Z**.

After the last prompt is answered, the **STR** file will be updated with the new security information.

You will be returned to the SETUP procedures menu where you can select another option or exit SETUP.

16.7 CREATE Procedure



Selecting the **Create data file** option from the SETUP procedures menu will allow you to create a data file.

After the CREATE option has been selected, the create screen will be displayed and you will be prompted for some information.

Note

If the database engine is RMS, the following screen is displayed. If the database engine is not RMS, a similar screen will be displayed. The screen heading shows the database engine and name of the file that will be created.

Example 16–11: CREATE Procedure RMS Input Screen

```
SETUP V4.2              Creating RMS data file: CUSTOMER.DAT

File organization:
Recordsize       :


                      +RMS File Organization+
                      |  Indexed            |
                      |  Sequential         |
                      +---------------------+


EXIT = Exit                                          \ = Back  HELP = Help
```

RMS File Organization

Select how you want your file to be organized. The options are:

- **INDEXED** - the data file record contains one or more *key* fields
- **SEQUENTIAL** - the data file record contains *no* key fields

If you are going to be sorting and/or extracting data from the RMS file you are creating, the indexed file organization is more efficient than the sequential organization. This is especially true if you have large files.

An *indexed file* contains one or more indexed key fields which allow programs to locate and access file data very quickly without having to read through the file to locate records.

A *sequential file* has no key fields and accessing file data can be very slow and time consuming. Locating data in a sequential file requires reading every record in the file to find all occurrences of the data.

If you select **Indexed** file organization, you will be asked for the *key* fields in later prompts.

Record size

Enter the record size that you want for this data file.

The default record size is 25% larger than the actual length of the fields defined. This allows room for adding new fields to the record without having to expand the record size.

The legal record sizes depend on the database engine (file management system) you use. For RMS, sizes range from 1 to approximately 32000.

Note

Depending on the database engine and file organization, SETUP might ask for additional information.

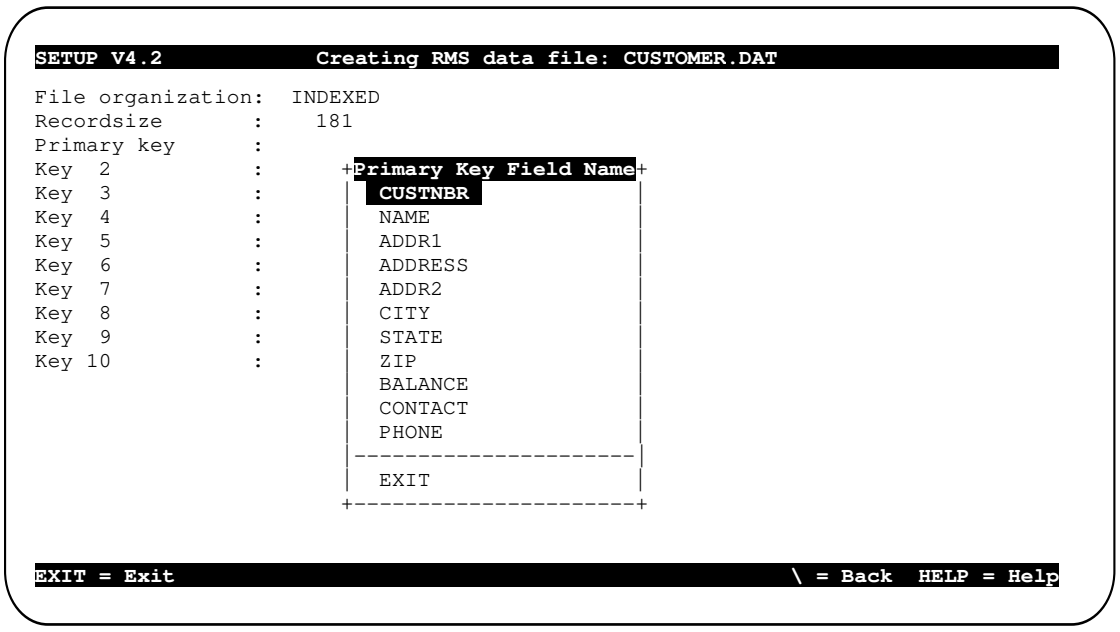
If no additional information is required to create this data file, you can enter "Y"es at the *Proceed with file creation* prompt and your data file will be created using the appropriate database engine.

Primary key field

Select the field from the list of defined fields, that you want to use as the primary key field. The first field is the default.

RMS indexed files require at least one key field, the *primary* key field.

You can refer back to the "DEFINE Procedure" section for information on key fields.



Allow duplicates

You can control whether the data to be stored in the RMS key field will be *unique* (only one occurrence of the data in the key field) or allow *duplicates* (multiple occurrences of the data in the key field).

Data access works most efficiently with unique key fields.

Normally, the PRIMARY key field is set up to allow only unique data in the field (i.e. NO duplicates) and the other key fields (ALTERNATE key fields) can be unique or not.

If you want more than one key field, select the fields you want to use as ALTERNATE key fields. Selecting **All keys defined** ends key field selection.

You will be asked if you want to proceed with file creation. Enter **Y** to create the data file.

After the file has been created, you will be returned to the SETUP procedures menu.

16.8 Defining COBOL Record Fields

The **TTI_RUN:COBOL_DEF.INT** program allows COBOL users to define COBOL record fields. This program processes a text file that contains a COBOL record definition statement and creates an INTOUCH data dictionary (definition file).

The program expects the text file to contain a *single* COBOL record definition statement. The file must be in one of the following formats:

— An ANSI formatted file which contains line numbers in columns 1-6 and comments in columns 72-80.

— A TERMINAL formatted file which has no reserved columns at the beginning and end of the lines.

The file name or file specification that is to be defined can be up to 63 characters in length if the screen margin is set at 80. If the margin is set to 132, 115 characters are allowed.

16.8.1 Define Example

Here is an example of how to use the **TTI_RUN:COBOL_DEF** program. This example shows the input file and the created file.

The sample text file COBOL_TEST.RECORD contains the following record definition statement:

```
01  test_rec.
  05  code pic x.
  05  desc pic x(20).
  05  amount pic s9(5)v99 comp.
  05  tbl  occurs 3 times.
      10  tbl-code pic x(5).
      10  tbl-desc pic x(25).
```

To run the program, enter the following at the "$" prompt:

$ **INTOUCH/SOURCE TTI_RUN:COBOL_DEF**

The program screen will be displayed:

```
COBOL_DEF V4.2   Create INTOUCH Definition from COBOL Definition

COBOL copy-lib:
ANSI format   :




COBOL copy-lib? ____________________________________________________________

EXIT = Exit                                              \ = Back  HELP = Help
```

At the "COBOL copy-lib?" prompt, enter the file name. In this example, the file name is **COBOL_TEST.RECORD**.

At the "Is file in ANSI format (Y/N)? NO__" prompt, press the Return key to take the "NO" default because the COBOL_TEST.RECORD file is not in ANSI format, it is in TERMINAL format.

To proceed, enter "Y" at the "Proceed with definition creation (Y/N)?" prompt.

The program processes the COBOL_TEST.RECORD file and creates the COBOL_TEST.DEF file.

You are returned to the "COBOL copy-lib?" prompt. Enter EXIT to exit the program.

Checking the Output

To see what the COBOL_TEST.DEF file looks like, you can:

1. Get into INTOUCH by entering **INTOUCH**
2. Enter **CALL SETUP** to activate the SETUP routine.
3. Answer the following prompts:

   Structure name : **COBOL_TEST**
   Database engine: **RMS**
   Dataset : **COBOL_TEST**
   Data dictionary: **COBOL_TEST**

   After the prompts are answered, a COBOL_TEST.STR file is created.
4. Use the arrow keys to select the **Show** menu option and the **Brief field display** option

The following definition listing is created:

```
11-Apr-1995              Definitions for: COBOL_TEST.DEF              Page:   1

Field Name       Description      First  Length  Occurs  Last  Dtype  Print Mask
---------------- ---------------- -----  ------  ------ -----  -----  ----------
TEST_REC         Test rec             1     115       1    115  CH     none
      Semantics  : UC

CODE             Code                 1       1       1      1  CH     none
      Semantics  : UC

DESC             Desc                 2      20       1     21  CH     none
      Semantics  : UC

AMOUNT           Amount              22       4       1     25  IN     none
      Semantics  : NUM:2, RJ, ZF

TBL_A1           Tbl a1              26      30       1     55  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A1      Tbl code a1         26       5       1     30  CH     none
      Semantics  : UC

TBL_DESC_A1      Tbl desc a1         31      25       1     55  CH     none
      Semantics  : UC

TBL_A2           Tbl a2              56      30       1     85  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A2      Tbl code a2         56       5       1     60  CH     none
      Semantics  : UC

TBL_DESC_A2      Tbl desc a2         61      25       1     85  CH     none
      Semantics  : UC

TBL_A3           Tbl a3              86      30       1    115  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A3      Tbl code a3         86       5       1     90  CH     none
      Semantics  : UC

TBL_DESC_A3      Tbl desc a3         91      25       1    115  CH     none
      Semantics  : UC
```

17 INTOUCH Running Under VMS

17.1 File Specifications

Every file has a file specification. The file specification is used to access the file. In INTOUCH, a legal file specification looks like this:

node::device:[directory]filename.ext;version

When you use a file name in an INTOUCH statement or command, it must be given as a string expression. If the file name is given as a constant, it must be enclosed in quotes. For example:

OLD 'ACCOUNT'

You can specify any legal OpenVMS file specification with INTOUCH. Simply name the file according to the rules described in the *Command Language and DCL User's Guide* of the OpenVMS documentation set.

Note

The default extension for INTOUCH programs is .INT. The default for include files is .INC.

A Built-in Functions

INTOUCH has numerous built-in functions. This appendix describes the system and other built-in functions.

A.1 System Functions

The following functions return information about the system and user actions:

_BACK

Returns a TRUE or FALSE value. TRUE if the backslash key was pressed at the last prompt.

_CHANNEL

Returns the next available channel number.

```
10  out_ch = _CHANNEL
    OPEN #out_ch: NAME 'sys$scratch:test.lis'
    PRINT #out_ch: 'This is a test'
    CLOSE #out_ch
20  END
```

_DEBUG

Returns a TRUE or FALSE value. TRUE if DEBUG is on. FALSE if DEBUG is off.

_EXIT

Returns a TRUE or FALSE value. TRUE if EXIT was entered at the last prompt.

_EXTRACTED

Tells how many records were extracted in the last extract.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
30    INCLUDE cl(state) = 'CA'
40  END EXTRACT
50  PRINT 'Number of California clients:'; _EXTRACTED
60  END
```

_HELP

Returns a TRUE or FALSE value. TRUE if HELP or a question mark (?) was entered or the Help key is pressed at the last prompt.

_INTEGER

Sometimes when the user does a CALL, an integer value is returned as a result of the CALL. _INTEGER returns any resulting integer data.

```
10  LIBRARY 'LIBRTL'
20  CALL LIB$ICHAR('Q')
30  PRINT 'The integer value of "Q" is: '; _INTEGER
40  END
```

_REAL

Some routines return a real value as the result of the CALL. The _REAL function returns any resulting real data.

```
10  LIBRARY 'MTHRTL'
20  CALL MTH$ALOG(45.3)
30  PRINT 'The log is:'; _REAL
40  END
```

_REPLY

Returns the user's reply to the last prompt. The reply is returned as a string.

_STATUS

Returns the value given to INTOUCH by the operating system, for the last operating system request. This function is useful when debugging system level errors. The _STATUS function is often used with the SYSTEXT$ function.

_STRING

Upon the completion of an INPUT MENU statement, the concept _STRING contains the menu path taken by the user when selecting the menu item. (i.e. "#2;#3" means the 3rd item of the 2nd submenu.)

_TERMINATOR

The _TERMINATOR function returns the name of the key that terminated the last INPUT statement. The values returned are:

UP	Up arrow
DOWN	Down arrow
RETURN	Return
FIND	Find (find line, word)
INS	Insert
SEL	Select
PREV	Prev Screen
NEXT	Next Screen
REM	Remove
HELP	Help
DO	Do

PF1, PF2, PF3, PF4

KPO, KP1, KP2, KP3, KP4, KP5, KP6, KP7, KP8, KP9, MIN, COM, ENT, PER

F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, HELP, DO, F17, F18, F19, F20

A.2 Other Functions

The other non-system functions are:

ABS(num_expr)

ABS returns the absolute value of a specified numeric expression.

ACOS(num_expr)

ACOS(x) returns the angle whose COS is x. The angle is returned in radians.

ANGLE(num_expr, num_expr)

Given X and Y coordinates, returns the angle from 0,0. The angle is returned in radians.

ASCII(str_expr)

The ASCII function returns the decimal ASCII value of a string's first character. It is returned as an integer.

ASIN(num_expr)

ASIN(x) returns the angle whose SIN is x. The angle is returned in radians.

ATN(num_expr)

ATN returns the angle, in radians, of a specified tangent.

CEIL(num_expr)

CEIL(x) returns the ceiling of x. That is, the smallest integer that is not less than x.

CHANGE$(str_expr1, str_expr2, str_expr3)

CHANGE$ changes specified characters in str_expr1. Str_expr1 is the source string. Str_expr2 contains the target characters, and str_expr3 specifies the substitution characters. CHANGE$ returns the changed string.

CHANGE$ searches for the target characters within the source string and replaces these characters with the substitution characters. The substitution characters are mapped onto the target characters.

```
10  LET a$ = 'bdbdbdbd'
20  LET b$ = 'b'
30  LET c$ = 'c'
40  LET changed$ = CHANGE$(a$, b$, c$)
50  PRINT a$
60  PRINT changed$
70  END

RNH
bdbdbdbd
cdcdcdcd
```

CHARSET$[(str_expr)]

Returns the character set specified. The optional string expression can be used to specify the character set to return. The available character sets are:

UCASE	all upper-case letters (A-Z)
LCASE	all lower-case letters (a-z)
CONTROL	all control characters (ASCII 0-31)
ASCII	the ASCII character set, in order (0-255)

ASCII is the default character set for CHARSET$.

```
10  LINE INPUT 'Enter your text': text$
20  ! Change upper-case to lower-case
    ct$ = CHANGE$(text$, &
         CHARSET$('UCASE'), &
         CHARSET$('LCASE'))
    PRINT 'Lower-case version is:'; ct$
30  END
```

CHR$(int_expr1[, int_expr2])

CHR$ returns a string with the specified ASCII value (int_expr1) repeated the specified number of times (int_expr2). If no count is specified, a default count of one is used.

CONVERT(str_expr)

Given a string containing a mapped integer, this function returns the integer value.

```
10  a$ = 'AB'
    b = CONVERT(a$)
20  END

RNH
16961
```

CONVERT$(int_expr1 [[, int_expr2], int_expr3])

Given an integer (int_expr1) and an optional length (int_expr2), which defaults to four, this function returns a string mapping of the integer.

If the optional data type (int_expr3) is 17, the returned string will be a packed floating (PF).

The following data types are supported:

Table A–1 CONVERT$ Function - Supported Data Types

Data Type	Conversion Result
1	Integer (2 or 4 byte)
7	COBOL comp-3 (C3 packed decimal)
17	Packed floating (PF)

```
10  a$ = CONVERT$(16961)
    PRINT a$
20  END

RNH
AB
```

The CONVERT and CONVERT$ functions can be used in situations such as building segmented keys consisting of multiple data types.

COMPARE(str_expr1, str_expr2)

The COMPARE function compares two strings and returns a numeric value ranging from 0 (no match) to 100 (an exact match).

```
10  options$ = 'LEFT,RIGHT,UP,DOWN'
    best = 0
    best$ = ''
    INPUT 'Enter an option': usr_opt$
20  FOR idx = 1 TO ELEMENTS(options$)
      opt$ = ELEMENT$(options$, idx)
      score = COMPARE(opt$, usr_opt$)
      IF  score > best  THEN
        best = score
        best$ = opt$
      END IF
    NEXT idx
30  SELECT CASE best
    CASE 0
      PRINT 'Unknown option: '; usr_opt$
    CASE 100
      PRINT 'Option okay, it was: '; usr_opt$
    CASE ELSE
      PRINT 'Misspelled option: '; usr_opt$
      PRINT USING 'Did you mean ? (## percent)': best$, best
    END SELECT
40  END

RNH
Enter an option? dwn
Misspelled option: dwn
Did you mean DOWN (92 percent)
```

COS(num_expr)

COS returns the cosine of an angle the user specifies in radians.

COSH(num_expr)

COSH returns the hyperbolic cosine of a passed real number.

COT(num_expr)

COT(X) is equivalent to 1/TAN(X).

CPAD$(text_str, size [, pad_str])

CPAD$ returns a new string, padded on the left and on the right with pad characters. Text_str is the string to be centered, size is the size of the new string. The default pad character is a space.

```
10  PRINT CPAD$('123', 9, '0')
20  END

RNH
000123000
```

CSC(num_expr)

CSC(x) is the cosecant of X. It is shorthand for 1/SIN(x).

DATE

Returns today's date in YYDDD format.

DATE$[(int_expr1, [int_expr2])]

The DATE$ function returns the date in image format. Int_expr1 is a given Julian day number, the default is today's date. Int_expr2 indicates the desired output format for the date. The Julian day—the number of days since January 1, 1600.

Table A–2 DATE$ Function - Integer Values

Value (int_expr2)	Output Date Format
0	YYYYMMDD format
1	MMDDYYYY format
2	DDMMYYYY format
3	dd-Mon-yyyy format
4	Month dd, yyyy format

```
PRINT DATE$(123456) gives you

    19380104
    |   | |
   year mo day

PRINT DATE$ gives today's date

    19910417
    |   | |
   year mo day

PRINT DATE$(123456,1) gives you

    01041938
    | |   |
   mo day year

PRINT DATE$(123456,2) gives you

    04011938
    | |   |
  day mo  year

PRINT DATE$(123456,3) gives you

    04-Jan-1938
```

```
PRINT DATE$(123456,4) gives you
    January 4, 1938
```

To print today's date in a suitable format for a letter heading, you can use:

```
PRINT DATE$(DAYS(DATE$),4) gives you
    April 17, 1995
```

DAYS(str_expr [, int_num])

Given a date in CCYYMMDD or YYMMDD format, returns the number of days since January 1, 1600 (this date is day 1). This number is called the Julian day.

```
10  PRINT DAYS('19930403')
20  PRINT DAYS('930403')
30  END

RNH
 143634
 143634
```

Int_num indicates the desired input format for the date. The default input format is zero. If the century is not included, it assumes 1900 as the century.

Table A–3 DAYS Function - Integer Values

Value (int_num)	Input Date Format
0	CCYYMMDD or YYMMDD
1	MMDDCCYY or MMDDYY
2	DDMMCCYY or DDMMYY
3	DD-Mon-CCYY or DD-Mon-YY
4	Month DD, CCYY

```
10  PRINT DAYS('19930403',0)
    PRINT DAYS('04031993',1)
    PRINT DAYS('03041993',2)
    PRINT DAYS('03-Apr-1993',3)
    PRINT DAYS('April 3, 1993',4)
20  END

RNH
 143634
 143634
 143634
 143634
 143634
```

DAY$[(int_expr)]

Given an integer expression specifying the number of days since January 1, 1600, DAY$ returns the day of the week. If no integer expression is given, DAY$ returns the day of the week for today's date. The day is returned as a string expression (Friday, Saturday, etc.).

DECODE(str_expr, int_expr)

Given the string representation of a number and the base that the value is in (int_expr), INTOUCH returns the value in base 10. The number is returned as a real number.

DEG(num_expr)

Given an angle in radians, returns the number of degrees.

DIV0(num_expr1, num_expr2)

The DIV0 function divides num_expr1 by num_expr2. If num_expr2 (divisor) is 0, 0 is returned.

```
10  PRINT DIV0(0.8, 0.000004)
    PRINT DIV0(0.8, 0.0)
    PRINT DIV0(6, 3)
    PRINT DIV0(6, 0)
20  END

RNH
 200000
 0
 2
 0
```

DTYPE(expr)

The DTYPE function returns as an integer value, the data type of an expression or dynamic variable: 1 = string, 2 = integer, 3 = real.

```
10  DECLARE DYNAMIC x
20  x = 45.6
30  PRINT DTYPE(x)
40  END

RNH
3
```

EDIT$(str_expr,int_expr)

EDIT$ performs one or more editing operations, depending on the value of the integer expression, on the supplied string argument. The integer expression is one of the integers below, or a sum of integers below for the desired edit functions:

Table A–4 EDIT$ Function - Operation Values

Value	Edit Operation
1	Trim parity bits.
2	Discard all spaces and tabs.
4	Discard characters: CR, LF, FF, ESC, RUBOUT and NULL.
8	Discard leading spaces and tabs.
16	Reduce spaces and tabs to One space.
32	Convert lower case to upper case.
64	Convert "[" to "(" and "]" to ")".
128	Discard trailing spaces and tabs.
256	Do not alter characters inside quotes.

ELEMENTS(str_expr1 [, str_expr2])

Returns the number of elements in a string expression which contains a list of elements. Str_expr1 is the string containing the list of elements. Str_expr2 is the separator. A comma is the default separator.

- Leading and trailing spaces are removed from the text.
- Quoted data is treated as one element.

ELEMENT$(str_expr1, num_expr [, str_expr2])

ELEMENT$ returns the element from str_expr1 which is specified by the num_expr. Str_expr1 contains a set of elements with separators between them. The default separator is a comma:

```
10  LET a$ = ELEMENT$('ADD,DEL,EXIT',2)
20  PRINT a$
30  END

RNH
DEL
```

You can specify a separator other than the comma with str_expr2.

```
10  LET sentence$ = 'This is a test.'
20  LET a$ = ELEMENT$(sentence$,2,' ')
30  PRINT a$

RNH
is
```

More than one separator in a row returns a null for the corresponding element.

```
10  LET sentence$ = 'This,, is, a, test'
    PRINT ELEMENT$(sentence$, 2)
20  END

RNH
```

ENCODE$(num_expr, num_int)

Returns a string containing a number converted to the base specified. Num_expr is the value to convert. Num_int is the base to convert. For instance, '2' indicates binary, etc.

EPS(num_expr)

INTOUCH carries 15 digits of precision. The EPS(number) returns the smallest value, which when added to "number" yields the next number.

EVAL(str_expr)

This function evaluates the expression described in str_expr, and returns the result. If the variable being assigned the result is dynamic, the function puts the result in whatever data type the expression result was in. If the variable is NOT dynamic, and the result is the wrong data type, a "Data type mismatch" error is returned.

```
10  LINE INPUT 'Enter an expression': a$
    PRINT 'The result is '; EVAL(a$)
20  END

RNH
Enter an expression?  5 + 4
The result is  9
```

EXLABEL$

EXLABEL$ returns the label and line number executing when the last exception occurred, e.g., DO_INPUT.4

EXLINE

EXLINE returns the line number executing when the last exception occurred.

EXP(num_expr)

EXP function returns the value of the mathematical constant, "e", raised to a specified power.

EXTEXT$(int_expr)

EXTEXT$ returns explanatory text associated with a specified exception number.

EXTYPE

EXTYPE returns the number of the last exception that occurred. It is returned as an integer.

FALSE

Returns the constant 0. It is returned as an integer.

FILESPEC$(str_expr1 [, str_expr2 [, str_expr3]])

FILESPEC$ parses a file specification and returns either a full file specification or specific file specification fields.

Str_expr1 is the file specification to be parsed. If no file specification is given, the device and directory you are currently running from are returned.

Str_expr2 is a list of field names, separated by commas, which are to be returned. The field names are:

DEVICE	device name
DIRECTORY	directory name
NAME	file name
TYPE	type or extension name
VERSION	file version number
LOCATION	device and directory names
ALL or ""	full file specification

Str_expr3 is the default file specification. This parameter is optional.

FILESPEC$ can be used in various formats.

```
10  PRINT FILESPEC$('x.y', 'ALL')
    PRINT FILESPEC$('', 'ALL')
20  END
```

```
RNH
USER:[FRED]X.Y;
USER:[FRED].;
```

```
10  x$ = 'TTI_RUN:CLIENT'
    PRINT FILESPEC$(x$, 'ALL', '.dat')
20  END
```

```
RNH
STORAGE:[INTOUCH]CLIENT.DAT;
```

```
10  PRINT FILESPEC$('tti_run:client', 'ALL', '.dat')
    PRINT FILESPEC$('tti_run:client', 'LOCATION')
    PRINT FILESPEC$('tti_run:client', 'LOCATION, NAME')
    PRINT FILESPEC$('tti_run:client.dat;1')
20  END
```

```
RNH
STORAGE:[INTOUCH]CLIENT.DAT;
STORAGE:[INTOUCH]
STORAGE:[INTOUCH]CLIENT
STORAGE:[INTOUCH]CLIENT.DAT;1
```

FINDFILE$(str_expr [, int_expr])

Given a file name to find, returns the complete file specification of the first file found that matches the name given. If no file is found, the function returns a null string.

FINDFILE$ calls can be nested if the inner call has only one argument (i.e., the file specification, but no index number).

str_expr	The name of the file to search for. This can be just part of the full file specification.
int_expr	Which file specification to return if multiple files are found. This parameter is optional. The default is to return the first file found.
Result	The complete file specification of the file found.

```
10  PRINT FINDFILE$('tti_run:*.int')
```

```
RNH
DUA0:[INTOUCH]ALIENS.INT;5
```

```
10  DO
      LINE INPUT 'File.Spec': spec$
      IF  _EXIT  THEN  EXIT DO
      FOR i = 1 TO 9999
        file$ = FINDFILE$(spec$, i)
        IF  file$ = ''  THEN  EXIT FOR
        PRINT file$
      NEXT i
    LOOP
20  END

RNH
File specification? TTI_RUN:CLIENT.*
DUA0:[INTOUCH]CLIENT.DAT;7
DUA0:[INTOUCH]CLIENT.DEF;3
DUA0:[INTOUCH]CLIENT.INT;1
DUA0:[INTOUCH]CLIENT.SAV;1
DUA0:[INTOUCH]CLIENT.STR;16
```

FORMAT$(expr, str_expr)

Given an expression and a format, returns the result of the expression in the format indicated.

The '@' format character causes the character not to be translated by the formatter. The '<' and '>'are treated like an '@' character. You can justify a character string, but avoid zero suppression and zero insertion.

The FORMAT$ function takes an expression of any data type for the data to format (the first argument), including string expressions.

```
10  z$ = FORMAT$('1234567', '###~-####')
    PRINT 'Phone number: '; z$
20  END

RNH
Phone number: 123-4567
```

The FORMAT$ function returns all asterisks "*" in the case of overflow.

```
10  z$ = FORMAT$(12.23,'#.##')
    PRINT z$
20  END

RNH
****
```

FORMAT$( ) returns the same string data as given by the following:

```
PRINT USING str_expr: expr
```

The FORMAT$ function supports the DATE format and date arguments. Given a date in YYMMDD or CCYYMMDD format, FORMAT$ returns the date in the date format requested.

```
FORMAT$(z$, '{DATE [argument]}?')
```

The **?** can be replaced with a mask. If no date argument is provided, the default is MDCY.

```
10  z1$ = FORMAT$('950401', '{DATE MDCY}?')
    z2$ = FORMAT$('950401', '{DATE MDCY}##/##/####')
    z3$ = FORMAT$('19950401', '{DATE MDCY}?')
    z4$ = FORMAT$('19950401', '{DATE MDCY}##/##/####')
20  PRINT z1$, z2$
    PRINT z3$, z4$
30  END

RNH
04011995            04/01/1995
04011995            04/01/1995
```

Table A–5 FORMAT$ Function - Date Arguments

DATE Argument	YYMMDD Input	Result	CCYYMMDD Input	Result
none	950207	02071995	19950207	02071995
YMD	950207	950207	19950207	950207
CYMD	950207	19950207	19950207	19950207
MDY	950207	022195	19950207	022195
MDCY	950207	02071995	19950207	02071995
DMY	950207	070295	19950207	070295
DMCY	950207	07021995	19950207	07021995
DMONY	950207	07-Feb-95	19950207	07-Feb-95
DMONCY	950207	07-Feb-1995	19950207	07-Feb-1995
MONTHDY	950207	February 7, 95	19950207	February 7, 95
MONTHDCY	950207	February 7, 1995	19950207	February 7, 1995

The FORMAT$ function supports character rotation. The ROTATE option rotates the last nn characters of a string to the first position in the string.

```
FORMAT$(z$, '{ROTATE n}?')
```

The **?** can be replaced with a mask.

```
10  z1$ = FORMAT$('5552527800', '{ROTATE 3}?')
    z2$ = FORMAT$('5552527800', '{ROTATE 3}###~ ###~-####')
    z3$ = FORMAT$('TuneTommy', '{ROTATE 5}?')
    z4$ = FORMAT$('TuneTommy', '{ROTATE 5}#####~ ####')
20  PRINT z1$, z2$
    PRINT z3$, z4$
30  END

RNH
8005552527            800 555-2527
TommyTune             Tommy Tune
```

The FORMAT$ function supports the TIME format. Given a military standard time in HHMM, HH:MM, HHMMSS or HH:MM:SS format, FORMAT$ returns the time as HH:MM AM/PM or HH:MM:SS AM/PM.

```
FORMAT$(z$, '{TIME}?')
```

```
10  t1$ = FORMAT$('1022', '{TIME}?')
    t2$ = FORMAT$('19:45:36', '{TIME}?')
20  PRINT t1$
    PRINT t2$
30  END
```

```
RNH
10:22 AM
07:45:36 PM
```

The FORMAT$ function supports the ZIPCODE format. Given a 5, 6 or 9 digit zipcode, FORMAT$ returns a formatted zipcode.

```
FORMAT$(z$, '{ZIPCODE}?')
```

```
10  z1$ = FORMAT$('92126', '{ZIPCODE}?')
    z2$ = FORMAT$('K8A3P9', '{ZIPCODE}?')
    z3$ = FORMAT$('931327845', '{ZIPCODE}?')
20  PRINT '5 character zipcode : '; z1$
    PRINT '6 character zipcode : '; z2$
    PRINT '9 character zipcode : '; z3$
30  END
```

```
RNH
5 character zipcode : 92126
6 character zipcode : K8A 3P9
9 character zipcode : 93132-7845
```

FP(num_expr)

Given a number, returns the fractional part of the number.

FULLTIME$[(float_expr, [int_var])]

Given the number of seconds since the INTOUCH base date, this function returns the date and time in one of the formats given below.

Where float_expr is the number of seconds since the INTOUCH base date. The default is the current date and time. January 1, 1600 00:00:00 is considered the second 0.

Table A–6 FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
0	CCYYMDD HHMMSS
1	MMDDCCYY HHMMSS
2	DDMMCCYY HHMMSS

Table A–6 (Cont.) FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
3	DD-Mon-CCYY HH:MM:SS
4	Month DD, CCYY HH:MM:SS

```
10  PRINT FULLTIME$
20  sec = SECONDS('19910621 115042')
30  PRINT FULLTIME$(sec, 0)
40  PRINT FULLTIME$(sec, 1)
50  PRINT FULLTIME$(sec, 2)
60  PRINT FULLTIME$(sec, 3)
70  PRINT FULLTIME$(sec, 4)
80  END

RNH
19910621 123756
19910621 115042
06211991 115042
21061991 115042
21-Jun-1991 11:50:42
June 21, 1991 11:50:42
```

HASH$(str_expr1 [, str_expr2 [, int_expr]])

This function changes the plain text in str_expr1 into a hashed eight-byte string value. It can be used to develop one-way hashed passwords. The optional text in str_expr2 and optional int_expr can be used to further make the hashed value unique.

```
10  password$ = HASH$('TRUTH')
    INPUT 'Password': pwd$
    IF  HASH$(pwd$) = password$  THEN
      PRINT 'That was the correct password.'
    ELSE
      PRINT 'That was not the correct password.'
    END IF
20  END

RNH
Password?  MONEY
That was not the correct password.
```

INT(num_expr)

INT returns the whole portion of a real number as a real number.

INTEGER(num_expr)

INTEGER changes any numeric expression into an integer value and assigns the integer value to the variable specified.

IP(num_expr)

IP truncates the value of a real number at the decimal point and returns the integer portion.

ITEM(str_expr1, str_expr2)

The ITEM function returns the number of the first item that matches the whole or partial item name given. A negative number is returned if more than one item matches.

```
10  z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'MOD')
    PRINT z
20  END

RNH
4


10  z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'EX')
    PRINT z
20  END

RNH
-2
```

LBOUND(array_name [,int_expr])

Given an array and a dimension number, returns the low bound for that dimension. The default dimension is 1.

LCASE$(str_expr)

LCASE returns a string expression with all letters in lower case.

LEFT[$](str_expr, int_expr)

LEFT$ returns the leftmost **nn** characters from a string. Int_expr is the last character position to be included in the resulting string.

LEN(str_expr)

LEN returns the length of a string. It returns an integer.

LOG(num_expr)

LOG returns the natural logarithm of a specified number.

LOG2(num_expr)

LOG2 returns a number's base 2 logarithm.

LOG10(num_expr)

LOG10 returns a number's common logarithm.

LPAD$(text_str, size [, pad_str])

LPAD$ pads a string on the left with pad characters. The default pad character is a space.

```
10  PRINT LPAD$('123', 6, '0')
20  END
RNH

000123
```

LTRIM$(str_expr)

Removes all leading spaces (those on the left side of the string).

MATCH(str_expr1, str_expr2)

Str_expr1 contains a list of elements separated by commas. Str_expr2 contains a string. MATCH compares str_expr2 with each of the elements in str_expr1 and gives the number of the element that matches.

Quoted data is treated as one element; the quotes are not removed.

```
10  INPUT 'Which procedure (ADD,DEL,QUIT)': pro$
20  LET x = MATCH('ADD,DEL,QUIT', pro$)
30  ON x GOSUB 50, 70, 90
40  GOTO 10
50  PRINT 'Adding...'
60  RETURN
70  PRINT 'Deleting...'
80  RETURN
90  END
```

MAX(num_expr, num_expr)

MAX(x,y) returns the larger of the two values x and y.

MAXLEN(str_var)

Returns the maximum number of characters that a string variable can contain. Since all string variables are variable length, with a maximum of 65535, this function always returns 65535.

MAXNUM

Returns the largest number available in this implementation of INTOUCH.

MAXSIZE(array_name)

Returns the total number of elements that can be contained in an array.

MID[$](str_expr, int_expr1 [,int_expr2])

MID$ or MID returns a substring from the middle characters of a specified string, leaving the string unchanged. Int_expr1 is the starting position of the substring, and int_expr2 is the length of the substring. MID$(str_expr,int_expr1) will return the rest of the string.

```
10  a$ = 'beginmiddleend'
    middle$ = MID$(a$, 6, 6)
    end$ = MID$(a$, 6)
    PRINT middle$, end$
20  END

RNH
middle     middleend
```

MIN(num_expr1, num_expr2)

MIN(x,y) returns the lesser of the values x and y.

MOD(num_expr1, num_expr2)

Gives the remainder of one number divided by another.

ORD(str_expr)

Given the ASCII name of a character, returns the location of that character in the ASCII character table. It returns an integer number.

ORDNAME$(int_expr)

Given the location of a character in the ASCII character table, returns the name of that character.

PARSE$(str_expr)

PARSE$ splits a string into tokens and returns each token separated by a space. Letters are upper-cased, except within quotes. Tail comments are ignored. Embedded "$" characters are allowed. The maximum line length is 1024 characters.

```
a$ = 'company$ = abc$ +"and sons" !rnn'
PRINT PARSE$(a$)

RNH
COMPANY$ = ABC$ + "and sons"
```

PATTERN(str_expr1, str_expr2)

Matches any characters in text (str_expr1) with the pattern (str_expr2). Str_expr1 is the text to search and str_expr2 is the pattern being searched for. Returns the location of the first character in the text that contains the pattern. If the characters cannot be found, returns zero.

PATTERN Options and Examples

Pattern options can be mixed and matched with unlimited complexity.

?

Matches any character in the text.

```
10  IF  PATTERN('The quick brown fox', &
       'f?x') > 0  THEN
       PRINT 'Found'
    END IF
20  END

RNH
Found
```

*****

Matches zero or more of a character that precedes the asterisk.

```
10  IF  PATTERN('aaa   03/26/92', 'a* *03/26/92') > 0  THEN
      PRINT 'The date is found'
    END IF
20  END

RNH
The date is found
```

{}

Used to define a group of characters. The characters in the enclosed group can be ranges such as {a-z} or individual characters such as {AQX}.

```
10  text$ = 'A1N5V7N0'
    rule_str$ = '{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}'
    IF  PATTERN(text$, rule_str$) > 0  THEN
      PRINT 'Driver's licence is correct'
    END IF
20  END

RNH
Driver's licence is correct
```

{^}

Looks for characters that are NOT {^A-Z}. The result of line 30 below shows the difference between using '?' and {^}.

```
10  PRINT PATTERN('Mary J. Smith','{^Mar}')
20  PRINT PATTERN('Mary J. Smith','J. {^S}')
30  PRINT PATTERN('Mary J. Smith','J. S?')
40  END

RNH
4
0
6
```

~

The '~' (quote) character looks for a pattern of text that IS an * (stands for itself).

```
10  text$ = '$4,670.00'
    IF  PATTERN(text$, '$~4, 670.00') > 0  THEN
      PRINT 'Your text is correct'
    END IF
20  END

RNH
Your text is correct
```

{<cc|ccc|c>}

Looks for text in str_expr1 that matches the enclosed groups.

```
10  text$ = 'The area code is 619'
    IF  PATTERN(text$, 'is {<619|714|916>}') > 0  THEN
      PRINT 'Your area code is on the list'
    END IF
20  END

RNH
Your area code is on the list
```

PI

Returns the value 3.141592653589793.

PIECES(str_expr1 [,str_expr2])

and

PIECE$(str_expr1,num_expr[,str_expr2])

PIECE$ returns an element from str_expr1 specified by num_expr. Str_expr1 contains a list of elements. The separator can be indicated by str_expr2. The default separator is a carriage-return line-feed pair.

These two functions are similar to the ELEMENTS( ) and ELEMENT$( ) functions except that:

- no leading or trailing spaces are removed from the text.
- quoted data is not skipped over.
- the default separator is the two-character sequence. CHR$(13)+CHR$(10), the CR/LF sequence used by WRAP$( ).

```
10 CLEAR
    MESSAGE 'Press GOLD/F when done'
    LINE INPUT AREA 5, 10, 8, 60: text$
    PRINT AT 10, 1: 'Rewrapped text'
    wt$ = WRAP$(text$, 1, 30)
    PRINT 'Number of lines: '; PIECES(wt$)
    PRINT wt$
    PRINT
    PRINT 'First line was: '; PIECE$(wt$, 1)
20  END
```

POS(str_expr1, str_expr2[, int_expr])

POS searches for a substring within a string. It returns the substring's starting character position. Str-exp1 is the string to be searched, str-exp2 is the substring to locate, and int-exp is the OPTIONAL character position at which to begin the search. (The default is the start of the string.)

PRETTY$(str_expr)

PRETTY$ converts text so that the text displays on any terminal. Named control characters show up with their names. Other control characters show up as {X} where "X" is the letter to press or as {XX} where "XX" is the hexadecimal value of the character.

```
10  a$ = 'Hello' + CHR$(5) + CHR$(161) + CHR$(7)
20  PRINT PRETTY$(a$)
30  END

RNH
Hello{^E}{A1}{bel}
```

QUOTE$(str_expr)

The QUOTE$ function encloses a string expression in double quotes. If the string expression is already enclosed in double quotes, QUOTE$ leaves it alone. If the string expression is already wrapped in single quotes, QUOTE$ replaces them with double quotes. Elements double quoted within the string expression are given another pair of double quotes (see following example). Elements single quoted within the string expression are ignored.

```
10  DO
      CLEAR
      PRINT AT 1,1:
      MESSAGE 'Enter a line of text to be quoted'
      PRINT 'Text:'
      INPUT '', LENGTH 30: line$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  line$ = ''  THEN REPEAT DO
20    PRINT
      PRINT 'Quoted text using the QUOTE$ function...'
      PRINT quote$(line$)
      DELAY
    LOOP
30  END

RNH
Text:
? The little boy cried "wolf!"

Quoted text using the QUOTE$ function...
"The little boy cried ""wolf!"""
```

RAD(num_expr)

Given a measurement in degrees, returns the number of radians.

REAL(num_expr)

REAL changes any numeric expression into a real or floating-point value and assigns the real value to the variable specified.

```
10  INPUT 'Your age': age%
    LET decimal_age = REAL(age%)
    PRINT 'Your number is'; decimal_age
20  END
```

REMAINDER(num_expr1, num_expr2)

REMAINDER(x,y) returns the remainder when X is divided by Y. It differs subtly from MOD. MOD(-4,3) = 2 while REMAINDER(-4,3) = -1.

REPEAT$(str_expr, int_expr)

Creates a string composed of the specified string repeated the specified number of times.

REPLACE$(str_expr1, str_expr2 [,str_sep1][,str_sep2])

Searches for a list of patterns in the str_expr1 and replaces them with the output string from str_expr2. Returns the replaced string expression.

Str_expr1 is a list of patterns to search for.

Str_expr2 is the replacement list.

Str_sep1 is the optional separator for replacement items. The default is a comma.

Str_sep2 is the optional separator between the input and output text in items. Default is =.

```
10  text$ = '01-Mar-1981'
    PRINT REPLACE$(text$, 'r=y 8=9' , ' ')
20  END

RNH
01-May-1991
```

RIGHT[$](str_expr, int_expr)

RIGHT$ returns the rightmost characters from a string. The int_exp is the character position of the last character to be included in the substring COUNTING FROM THE RIGHT.

```
10  ans$ = RIGHT$('Daniel', 2)
    PRINT 'rightmost char.= ', ans$
20  END

RNH
rightmost char.= el
```

RND

or

RND(num_expr)

RND returns a random number greater than or equal to zero and less than one. If a numeric expression (num_expr) is given, RND returns a whole number between one and the numeric expression.

ROUND(num_expr [, int_expr])

ROUND rounds a num_expr to the specified number of decimal places (int_expr). The default int_expr is 0.

RPAD$(text_str, size[,pad_str])

RPAD$ pads a string on the right with pad characters. The default pad character is a space.

```
10  PRINT RPAD$('123', 6, '0')
20  END

RNH
123000
```

RTRIM$(str_expr)

Returns a string without any trailing spaces (those on the right side).

SCAN(str_expr1, str_expr2 [, int_expr])

Scans str_expr1 for the characters in str_expr2 and returns the position at which str_expr2 begins. Int_expr specifies a character position at which the search is to begin.

The characters in str_expr2 need not appear contiguously in str_expr1.

```
10  LET a$ = 'Cameron Whitehorn'
    LET b$ = 'amr Wtor'
    LET position = SCAN(a$, b$)
    PRINT position
20  END

RNH
2
```

SEC(num_expr)

Returns a secant of a given angle (1/COS(num_expr)). Num_expr is a passed angle.

SECONDS(str_expr)

Given a full-time string in CCYYMMDD HHMMSS, YYMMDD HHMMSS, HHMMSS or HHMM format, returns the number of seconds since the INTOUCH base date (January 1, 1600 00:00:00).

The number of seconds is returned as a floating point number.

```
10  z  = SECONDS('19931201 103050')
    z1 = SECONDS('931201 103050')
    z2 = SECONDS('103050')
    z3 = SECONDS('1030')
20  PRINT 'Seconds CYMDHMS ='; z
    PRINT 'Seconds  YMDHMS ='; z1
    PRINT 'Seconds     HMS ='; z2
    PRINT 'Seconds     HM  ='; z3
30  END

RNH
Seconds CYMDHMS = 12430837850
Seconds  YMDHMS = 12430837850
Seconds     HMS = 37850
Seconds     HM  = 37800
```

SEG$(str_expr, int_expr1, int_expr2)

Extracts the substring given a first and last character position.

SGN(num_expr)

Returns the sign of a number. It returns a +1 if the expression is positive, a -1 if the expression is negative, and 0 if the expression is zero.

SIN(num_expr)

SIN returns the sine of an angle specified in radians.

SINH(num_expr)

SINH(X) returns the hyperbolic sine X.

SIZE(array_name [,int_expr])

Returns the number of elements in one dimension of an array.

array-name	Array to examine
int-expr	Dimension to get the size of. The default dimension is 1.

SKIP(str_expr1 [, str_expr2] [, int_expr])

SKIP returns the position of the character following the last skipped character.

Str_expr1 is the text string to be searched.

Str_expr2 contains the list of characters which are to be skipped. If only one argument is given, SKIP will skip over spaces, tabs and nulls.

Int_expr contains the search start position. This parameter is optional.

```
10  a$ = '31415 hello'
    z = SKIP(a$, '1234567890 ')
20  PRINT mid(a$, z)
30  END

RNH
Hello
```

SORT$(str_expr1 [,str_expr2])

Sorts the elements from a str_expr1 in ASCII value order. Returns a list of the sorted elements.

Str_expr1 contains the list of elements to be sorted.

Str_expr2 is an optional separator. Default is a comma.

```
10  a$ = 'code area is'
    a_sort$ = SORT$(a$, ' ')
    PRINT a_sort$
20  END

RNH
area code is
```

SPACE$(num_expr)

Returns the number of spaces indicated by num_expr.

SQR(num_expr)

SQR returns the square root of a number.

STR$(num_expr)

STR$ changes a number to a numeric string. The string that is created does not have any extra leading or trailing spaces.

SYSTEXT$

or

SYSTEXT$(int_expr)

SYSTEXT$ returns the text associated with the operating system status specified. If no int_expr is supplied, INTOUCH returns the text for the last system status.

TAB(int_expr)

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to the right to a specified column.

TAN(num_expr)

TAN returns the tangent of an angle that is specified in radians.

TANH(num_expr)

Returns the hyperbolic tangent of the numeric expression given.

TIME(int_expr)

The value returned by the TIME function depends on the value of int_expr.

If int_expr = 0, TIME returns the number of seconds since midnight.

If int_expr = 1, TIME returns the CPU time of the process in tenths of a second.

If int_expr = 2, TIME returns connect time of the current process in minutes.

TIME$

or

TIME$(num_expr)

If num_expr is NOT specified, TIME$ returns the current time of day in HH:MM:SS format.

Num_expr is the number of seconds since midnight. The result is returned in HH:MM format.

```
10  PRINT TIME$(1800)
    PRINT TIME$(54178)
20 END

RNH
00:30
15:02
```

TIME(5)

Returns the number of seconds since INTOUCH was invoked. This function can be used to time events to the nearest 100th/sec.

TRIM$(str_expr)

Returns the string specified stripped of any leading or trailing spaces and tabs.

```
10  LET a$ = '    HELLO    '
20  PRINT '*'; a$; '*'
30  LET stripped$ = TRIM$(a$)
40  PRINT '*'; stripped$; '*'

RNH
*    HELLO    *
*HELLO*
```

TRUE

Returns the constant 1. It is returned as an integer.

TRUNCATE(num_expr, int_expr)

Truncates a real number to a given number of decimal places.

UBOUND(array_name [, int_expr])

Given an array and a dimension number, returns the upper bound for that dimension. It returns an integer value. The default dimension is 1.

UCASE$(str_expr)

UCASE returns a string expression with all letters in upper case.

UNQUOTE$(str_expr)

The UNQUOTE$ function removes one set of quotes from a quoted string expression. If the string expression is not quoted, UNQUOTE$ leaves the string alone. UNQUOTE$ does not affect internally quoted elements.

```
10  DO
      CLEAR
      PRINT AT 1,1:
      MESSAGE 'Enter a line of text to be unquoted'
      PRINT 'Text:'
      INPUT '', LENGTH 50: line$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  line$ = ''  THEN  REPEAT DO
20    PRINT
      PRINT 'Quotes removed using the UNQUOTE$ function...'
      PRINT UNQUOTE$(line$)
      DELAY
    LOOP
100 END

RNH
Text:
? "I will not take these 'things' for granted"
```

```
Quotes removed using the UNQUOTE$ function...
I will not take these 'things' for granted
```

VAL(num_str)

VAL returns the floating-point value of a numeric string.

VALID(text_str, rule_str)

VALID is used to validate user responses.

Text_str is the text to be validated.

Rule_str is the list of validation rules.

Multiple validation rules are separated by a semi-colon. If given characters are NOT between quotes, they are upper-cased.

VALID returns an error if there is an invalid validation rule.

```
'Illegal validation rule' (-4021)
```

VALID returns TRUE or FALSE according to the following validation rules:

- **ALLOW text1, text2, text3 [to text4]**

  ALLOW compares the text_str with each element on the rule_str list (defaults to CHARACTER) and returns TRUE if found.

  For numeric comparison, "NUMBER" or "INTEGER" *must* be used, otherwise, VALID( ) will do a string-compare validation...left justified.

```
10  IF  VALID('ann', 'ALLOW ann, dan, tom')  THEN  PRINT 'true'
20  IF  NOT VALID('10',  'NUMBER; ALLOW 1 to 6; MAXLENGTH 2') &
          THEN PRINT 'false'
30  IF  VALID('12', 'NUMBER; ALLOW 1 TO 24,99') THEN  PRINT 'true'
40  END

RNH
true
false
true
```

- **DISALLOW text1, text2, text3 [to text4]**

  Returns TRUE if text_str cannot be found on the rule_str list.

```
10  IF VALID('10', 'DISALLOW 01,03,05; MINLENGTH 2') &
      THEN PRINT 'true'
20  END

RNH
true
```

- **MINLENGTH nn**

  Minimum of nn characters long.

- **MAXLENGTH nn**

  Maximum of nn characters long.

- **LENGTH nn**

  Exactly nn characters long.

- **CHARACTERS ccc**

  Returns TRUE if the given characters (letters or numbers) are on the rule_str list.

- **NOCHARACTERS ccc**

  Returns TRUE if the given characters are not on the rule_str list.

- **LETTERS**

  Returns TRUE if given text_str consists only of A - Z, a - z or space.

- **LCASE**

  Returns TRUE if given text_str consists of lower-case letters.

- **UCASE**

  Returns TRUE if given text_str consists of upper-case characters.

- **DIGITS**

  Returns TRUE if text_str consists of the numbers 0 - 9.

- **DECIMALS nn**

  Specifies the maximum number of decimal places.

- **NUMBER**

  Indicates that text_str is numeric (validates that number).

- **INTEGER**

  Returns TRUE if given integer number is up to 32 bits long (i.e. 2147483647).

- **INTEGER WORD**

  Returns TRUE if given integer number is up to 16 bits long (i.e. 32767).

- **DATE**

  Validates a given date in: YYMMDD or CCYYMMDD format.

DATE	YMD	YYMMDD format
DATE	DMY	DDMMYY format
DATE	MDY	MMDDYY format
DATE	MDCY	MMDDCCYY format

Note

For all dates, use 'LENGTH nn' to control whether a 6- or 8-character date is required.

- **DATE DMONY**

  Checks if text_str is a legal date in DD-Mon-YY format and returns TRUE or FALSE.

```
10  PRINT VALID('01-May-93', 'DATE DMONY')
20  END

RNH
1
```

- **DATE DMONCY**

  Checks if text_str is a legal date in DD-Mon-CCYY format and returns TRUE or FALSE.

```
10  PRINT VALID('01-May-1993', 'DATE DMONCY')
20  END

RNH
1
```

- **FULLTIME**

  Checks if text_str is a legal date and time in CCYYMMDD HHMMSS or YYMMDD HHMMSS format and returns TRUE or FALSE.

```
10  PRINT VALID('19931231 010101', 'FULLTIME')
    PRINT VALID('931231 010101', 'FULLTIME')
20  END

RNH
1
1
```

- **CODE**

  Checks for VALID INTOUCH code syntax.

```
10  a$ = 'PRINT 'Hello, ' name$'
    IF  NOT VALID(a$, 'CODE')  THEN  PRINT 'false'
30  END
```

- **PATTERN**

  Matches text_str with PATTERN's rule_str. For details see the PATTERN$ function.

- **REQUIRED**

  Returns TRUE if text_str is not null or space(s).

- **YES/NO**

  Returns TRUE if text_str consists only of the following: YES, NO, Y or N.

- **VRULES**

  Checks if a text_str is a legal set of validation rules, and returns TRUE or FALSE.

```
10  PRINT VALID('integer', 'vrules')
20  END

RNH
1
```

- **PRINTMASK**

  Checks the format of text_str and returns TRUE if it is legal or FALSE otherwise.

```
10  text_str$ = '##.##'
    PRINT VALID(text_str$, 'PRINTMASK')
20  END

RNH
1
```

- **EXPRESSION**

  Validates a text_str$ and returns TRUE if it is a legal INTOUCH expression or FALSE otherwise.

```
10  text_str$ = 'total = a% + 30'
    PRINT VALID(text_str$, 'EXPRESSION')
    PRINT VALID('##~-###', 'EXPRESSION')
20  END

RNH
1
0
```

- **MENU**

Validates a text_str$ and returns TRUE if it is a legal menu description for an INPUT MENU statement, or FALSE otherwise.

```
10  text_str$ = '%multi,a,b,c'
    PRINT VALID(text_str$, 'MENU')
20  END

RNH
1
```

- **FILTER**

  Filters the text_str before the validation. The following filters are allowed:

  — **LTRIM$** - Remove leading spaces.

  — **RTRIM$** - Remove trailing spaces.

  — **TRIM$** - Remove both leading and trailing spaces.

  — **UCASE$** - Convert to upper-case characters.

  — **LCASE$** - Convert to lower-case characters.

  — **REPLACE** - If text_str contains a few 'old_t1' characters, all of them will be replaced by 'new_t1'.

  ```
  old_t1=new_t1, old_t2=new_t2, ....
  ```

  — **REMOVE** - Removes any characters from the text_str.

  — **CHANGE** - Changes specified characters. For details see the CHANGE$ function.

  ```
  10  IF  VALID('abcd10', &
            'FILTER REMOVE "10"; LETTERS'  THEN  PRINT 'true'
      IF  VALID("ab1cd1", "FILTER REPLACE '1'='e';LETTERS") &
            THEN  PRINT 'true'
      IF  VALID(" 1234  ","FILTER TRIM; NUMBER") &
            THEN  PRINT 'true'
  30  END

  RNH
  true
  true
  true
  ```

  — **RESTORE** - Restores data to pre-filtered state for further validations. Can be used for multiple validation rules.

```
10  IF  VALID("123", &
            "FILTER REPLACE '1'='a';RESTORE;NUMBER") &
    THEN  PRINT "true"
20  END

RNH
true
```

WRAP$(str_expr, int_expr1, int_expr2)

Returns a word-wrapped text string, given left and right margins. Each line of the string is separated with a CR/LF.

Where string_expr = text string to wrap, int_expr1 = left margin, int_expr2 = right margin.

XLATE$(str_expr1, str_expr2)

The XLATE$ function translates one string to another by referencing a table you supply. For example, the XLATE$ function can translate from EBCDIC to ASCII. The first str_expr is the string to be translated. The second str_expr is the translation table.

B Reserved Words

Reserved words are words that cannot be used as identifiers. INTOUCH's reserved words are:

DATE
DATE$
ELSE
EXLINE
EXTYPE
MAXNUM
NOT
PI
PRINT
REM
RND
TIME
TIME$

The following words will be reserved in future versions:

CON
IDN
NUL$
TRANSFORM
ZER

C INTOUCH's Error and Exception Messages

Errors are returned either immediately after typing a syntactically incorrect line, or when the RUN command is given and your program is syntactically incorrect.

C.1 Compile Errors

ACCESS mode not INPUT, OUTPUT, or OUTIN
ADD without matching END ADD
Already declared as other than array
Array name expected
Array not yet dimensioned
Array or structure reference is missing "( )"
Bad line number
CANCEL ADD without matching ADD
CANCEL EXTRACT without matching EXTRACT
Can't find relation in database
Can't GO
Can't store into an internal field
Can't use this option with FIELD clause
CASE expression is of different type than SELECT CASE
CASE statement missing 'TO'
CASE without matching SELECT CASE
CASE OF not within SELECT block
Compilation errors
Compiler doesn't handle this yet
Database engine discovered bad binary information buffer
Datatype mismatch
DO without matching LOOP
Dots not allowed in variable names
Dynamic variables may not be used as arrays
ELSE without matching IF
END ADD without matching ADD
END EXTRACT without matching EXTRACT or REEXTRACT
END IF without matching IF
END HANDLER without matching HANDLER
END SELECT not within SELECT block
END WHEN without a previous USE
EXIT ADD without matching ADD
EXIT EXTRACT without matching EXTRACT
EXPECTED KEY keyword
EXTRACT without matching END EXTRACT
Expected a line number or label
Expected a relational operator
Expected an expression

Expected ELSE clause
Expected end-of-line
Expected EXCEPTION keyword
Expected FIELD keyword
Expected file name expression
Expected IN or USE keywords
Expected INPUT keyword
Expected 'ON' or 'OFF'
Expected quoted string
Expected RECORD keyword
Expected STEP keyword
Expected STRUCTURE keyword
Expected THEN clause
Expected ','
Expected ':' in substring expression
Expected ',' or ':'
Expected ',' or ';'
Expected ',' or ')'
Expected '('
Expected ')'
Expected '='
Expression poorly formed
Expression too complex
EXTRACT without matching END EXTRACT
INPUT missing variable list
Invalid file name
Invalid relation name
Invalid routine name
Invalid workspace name
FOR EACH may not appear within and EXTRACT
FOR index must be real or integer simple variable
FOR without matching NEXT
Function or array not declared
GOTO or GOSUB expected
HANDLER name not found
HANDLER or FUNCTION name cannot be used as a label
HANDLER without matching END HANDLER
IF without matching END IF
Illegal ADD STRUCTURE nesting
Illegal character in source code
Illegal delimiter
Illegal EXTRACT nesting
Illegal integer
Illegal structure field name
Illegal use of dynamic variable/structure in CASE
Illegally nested error handlers
Illegally nested PROGRAM statement
INCLUDE, EXCLUDE, or SORT outside EXTRACT block
Internal END-ON without preceding ON
Invalid NEXT variable
Leading white space not within a multiple statement line
Line number or label not found
LOOP without matching DO

LSET, CSET, and RSET require string assignments
Missing operand
Missing '=' in FOR statement
Missing ':' in substring expression
Missing '('
Missing ')'
More than 32 dimensions not allowed
Multiply defined HANDLER
Multiply defined label
Nested FOR loops with same index variable
NEXT without matching FOR
No END statement terminating PROGRAM block
Not existing HANDLER name
Only one OPTION BASE allowed per program
Only one OPTION integer/real allowed per program
OPEN statement missing NAME keyword
PARTIAL only allowed for character fields
PROGRAM statement not first in program
Relation name expected
REPEAT DO or ITERATE DO not inside DO loop
REPEAT FOR, ITERATE FOR, or EXIT FOR not in FOR loop
Reserved word
RESUME, RETRY, CONTINUE, or EXIT HANDLER not within handler
SELECT CASE without matching END SELECT
SELECT has no matching END SELECT
Statement following LOOP is not WHILE or UNTIL
Statement is only legal in immediate mode
Statement must be inside a program
Structure name expected
Syntax error
Syntax error in specified expression
This option needs a FIELD clause
Too many relations declared
Too many structures in the relation
Tried to DECLARE an illegal or typed variable
Unexpected number of dimensions
Unknown special variable
Unrecognized COMPILE option
Unrecognized keyword
Unrecognized OPEN keyword
Unrecognized relational operator
Unrecognized SET/ASK keyword
Unrecognized statement
Unterminated ELSEIF statement
USE block has no END WHEN
Use of '()' not valid in this context
USE without preceding WHEN EXCEPTION IN...
Using '()' with array name not valid in this context
Variable already declared
Variable currently exists with a different data type
Variable names may not contain embedded dots
Valid OPTION BASE parameters are 0 and 1
Warning: Missing quote assumed

	WHEN EXCEPTION IN has no USE block
	WHEN EXCEPTION USE has no END WHEN
	WHEN may not be inside of a HANDLER or USE block
	WHILE/UNTIL or EXIT DO not inside DO loop
	Wrong number of arguments
	Wrong number of subscripts in array reference

C.2 Exceptions

Exceptions only occur during execution. The runtime exceptions are:

	Too many GOSUBs
	User CAUSED exception
	Improper entry to error handler detected
	Dynamic variable used but not assigned
	Upper bound is less than lower bound
	File already exists (use REPLACE)
	Datatype mismatch
	Statistics are not enabled
-1001	Program interrupted
-1002	Source code changed
-3101	Dynamic variable datatype mismatch
-3102	Nonexistent structure field referenced
-4001	Illegal digit in decoded string
-4002	Numeric base in DECODE/ENCODE$ not between 2 and 16
-4003	Illegal argument to ORDNAME$( )
-4004	Illegal argument to date$
-4005	Illegal argument to DAYS
-4006	Invalid logical value
-4007	Invalid input screen format
-4008	Illegal memory address to MEM function
-4009	Illegal argument to ENCODE$( )
-4010	Illegal argument to CHARSET$( )
-4016	Illegal length argument
-4017	Expression or statement invalid
-4021	Illegal validation rule
-7011	Must close structure before reusing structure name
-7012	Error in directory name
-7013	Invalid attempt to update a key field
-7014	Deadlock detected
-7015	Database engine rejected add or update

-5001	Program interrupted
-5002	Exceeded virtual memory
-7001	Miscellaneous file error
-7002	Error in structure file
-7003	Error in data dictionary
-7004	Error in data file
-7005	No privilege for operation
-7006	No such record
-7007	Not a key field
-7008	Unknown DBMS in structure
-7009	Duplicate key
-7010	File full
-8001	Structure reference of nonexistent field
-8002	Numeric conversion error
-8003	Unsupported field data type
-8004	Disk quota exceeded
-8102	Syntax error in DATA
-10001	Illegal passing mechanism
-10002	Can't call external routine
-10003	Error in external routine
-10004	Dispatch routine not found
-10005	Illegal symbol name
-10006	Setting symbol failed
-10007	Ambiguous symbol definition
1001	Illegal number
1002	Integer overflow
1002	Floating overflow
1003	Overflow in built-in function
1004	Overflow in evaluating VAL
1006	Overflow on READ number
1007	Overflow in numeric input from terminal
1051	Overflow in evaluating string expression
1053	Overflow on READ string
1054	String too long on input
2001	Subscript out of bounds
3001	Division by zero
3002	Negative number raised to a nonintegral power
3003	Zero raised to negative power
3004	LOG( ) of zero or negative number
3005	SQRT( ) of negative number

3006	MOD or REM had zero divisor
3007	Argument to ACOS or ASIN not in range -1<=X<=1
3008	Attempt to evaluate ANGLE(0,0)
4001	Illegal argument to VAL( )
4002	Illegal argument to CHR$( )
4003	Illegal argument to ORD( )
4004	Illegal argument to SIZE
4005	Index in TAB less than one
4006	SET MARGIN value less than zonewidth
4007	SET ZONEWIDTH value greater than margin
4008	LBOUND index out of range
4009	UBOUND index out of range
4010	Second argument to REPEAT$ < 0
5001	REDIM overflow
6005	REDIM first index greater than second
7001	Bad channel number
7002	Attempt to open or close channel zero
7003	Channel number already in use
7004	File not open
7100	Invalid file attribute
7101	Too many files open
7110	File not found
7112	File locked by another user
7120	Illegal file name
7305	No current record
7322	Delete or update not allowed
8001	Out of DATA
8002	Too few input items
8003	Too many input items
8011	End of file on input
8012	Too few data items in record
8013	Too many data items in record
8101	READ a non-number
8102	Syntax error in input
8103	Non-numeric input when number expected
8105	Invalid file input
8201	Bad format$( ) or using string
8301	Record is larger than record size
8401	Timeout on input
8402	Illegal numeric value for time-expression

10001	ON index out of range
10002	RETURN without GOSUB
10004	SELECT found no matching CASE
10005	Chained program unavailable

D ASCII Character Set

Table D–1 ASCII Character Set

Decimal Value	ASCII character equivalent
0	NUL
1	SOH
2	STX
3	ETX (CTRL/C)
4	EOT
5	ENQ
6	ACK
7	BEL (bell)
8	BS (backspace)
9	HT (horizontal tab)
10	LF (linefeed)
11	VT (vertical tab)
12	FF (form feed)
13	CR (carriage return)
14	SO
15	SI (CTRL/O)
16	DLE
17	DCL (CTRL/Q XON)
18	DC2
19	DC3 (CTRL/S XOFF)
20	DC4
21	NAK (CTRL/U)
22	SYN
23	ETB
24	CAN
25	EM
26	SUB (CTRL/Z)
27	ESC (escape)
28	FS
29	QS
30	RS
31	US

Table D–1 (Cont.) ASCII Character Set

Decimal Value	ASCII character equivalent
32	SP (space)
33	!
34	"
35	#
36	$
37	%
38	&
39	’ (apostrophe)
40	(
41	)
42	*
43	+
44	, (comma)
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E

Table D–1 (Cont.) ASCII Character Set

Decimal Value	ASCII character equivalent
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\ (backslash)
93	]
94	^ (or up arrow)
95	_ (or back arrow)
96	‘ (grave accent)
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k

Table D–1 (Cont.) ASCII Character Set

Decimal Value	ASCII character equivalent
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	| (vertical line)
125	}
126	~ (tilde)
127	del (rubout)

E Creating Libraries for Use with INTOUCH

To create libraries for use with INTOUCH, you will need to follow these steps:

1 Write the source code using an editor. The source code can be in BASIC, FORTRAN or any other VMS provided language that supports the common calling interface. Here is an example of a BASIC routine:

```
10  SUB MULT(IN1%, IN2%, OUT%)
    OUT% = IN1% * IN2%
    END SUB
```

This file has only one subprogram. Your file can contain any number of subprograms.

2 Compile the source code with the appropriate compiler. For instance, the above BASIC routine would have to be compiled with the BASIC compiler:

```
$ BASIC MULT
```

3 Create an options file. Use an editor to create an options file. Give the options file a file extension of ".OPT".

For VAX systems:

Enter the option UNIVERSAL to specify the modules (subprograms and functions) contained in your routine.

```
UNIVERSAL = module [, module...]
```

If your routine contains more than one module, separate multiple module names with commas. The routine described above has only one module:

```
UNIVERSAL = MULT
```

For Alpha systems:

Use the following format to specify the modules (subprograms and functions) contained in your routine.

```
SYMBOL_VECTOR = (module1=PROCEDURE)
SYMBOL_VECTOR = (module2=PROCEDURE)
  . . .
```

If your routine contains more than one module, use a separate line for each module in the .OPT file. The routine described above has only one module:

```
SYMBOL_VECTOR = (MULT=PROCEDURE)
```

4 Link the library.

```
LINK/SHARE routine_name,options_file/OPTIONS
```

For the routine above, if your options file is called MULT.OPT, the link command is:

```
$ LINK/SHARE MULT,MULT/OPTIONS
```

You have now created the shareable image MULT.EXE, containing the routine MULT. You can now use this routine from within INTOUCH.

To use the routine, first name the library with a **LIBRARY** statement, supplying the complete file specification INCLUDING DEVICE AND DIRECTORY. Then, execute the routine with the CALL statement.

```
10  library "DUA1:[STEVE.MODULE]MULT.EXE"
    call mult(8%, 9%, answer%)
    print 'The answer is '; answer%
```

You can have any number of LIBRARY statements in a program. When you call a routine, each library specified is checked in turn for the routine.

F INTOUCH Database Interfaces

F.1 Supported Database Engines

Multiple database engines (record management systems) can be used with INTOUCH. The following database engines are currently supported:

RMS
Rdb
DBMS
USERBASE
POISE
FASTFILE
S1032
INGRES
dBASE III
ADABAS
ORACLE

The SETUP routine tells INTOUCH where data files are located and what record management systems are used. Once you have defined your structures using SETUP, you can use INTOUCH's structure statements to manipulate the data.

For detailed information on INTOUCH interfaces to the various database engines, you can refer to:

- the *INTOUCH 4GL - Interfaces to Database Engines* manual.
- Chapter 16, Creating Structures, Field Definitions with SETUP, in this manual.

Index

A

ABORT command • 2–5
ABS function • A–3
ACCESS - field definition item • 14–24
Accessing system utilities • 1–2
ACCESS INPUT • 14–4
ACCESS option in OPEN #chnl_num • 15–2
ACCESS options in OPEN STRUCTURE • 14–4
ACCESS OUTIN • 14–4
Access rules • 16–16
ACOS function • A–3
ADABAS • 16–1, 16–3, F–1
ADABAS data files • 16–5
ADD STRUCTURE/END ADD • 14–5
Add structure record • 14–5
  cancel add • 14–6
  exit add • 14–7
Alpha AXP systems • 2–14
Alternate key • 16–17, 16–27
Ambiguous constant • 3–5
Ampersand in line continuation • 3–3
ANGLE function • A–3
APPEND option in EXTRACT STRUCTURE • 14–19
APPLICATION - field definition item • 14–25
AREA option in INPUT • 5–12
AREA option of CLEAR statement • 6–20
Arithmetic operators • 3–14
Array
  defining • 10–1
  description • 3–10
  dimension • 10–1
  DIM statement • 10–1
  elements • 3–10
  low bound • 10–4
  OPTION BASE statement • 10–4
  REDIM statement • 10–3
  subscript • 3–10
Array elements • 16–14
ASCII character set • 14–13, D–1
ASCII function • A–4
ASIN function • A–4
ASK #chnl_num options • 15–9
  CURRENT • 15–9
  MARGIN • 15–9
  NAME • 15–10
ASK #chnl_num options (cont'd)
  ZONEWIDTH • 15–9
ASK #chnl_num statement • 15–8
ASK/SET SEED statement • 9–13
ASK ERRORS statement • 9–4
ASK KEYSTROKES statement • 9–7
ASK MARGIN statement • 9–8
ASK MESSAGELINE statement • 9–9
ASK PAGESIZE statement • 9–10
ASK RESPONSES statement • 9–12
ASK STRUCTURE #string_expr... • 14–36
ASK STRUCTURE: ACCESS • 14–35
ASK STRUCTURE: CAPABILITY • 14–31
ASK STRUCTURE: CURRENT • 14–29
ASK STRUCTURE: ENGINE • 14–37
ASK STRUCTURE: EXTRACTED • 14–32
ASK STRUCTURE: FIELDS • 14–30
ASK STRUCTURE: ID • 14–33, 14–40
ASK STRUCTURE: KEYS • 14–31
ASK STRUCTURE: POINTER • 14–33
ASK STRUCTURE: RECORD • 14–34
ASK STRUCTURE: RECORDSIZE • 14–35
ASK STRUCTURE:TIMEOUT • 14–36
ASK STRUCTURE FIELD: item • 14–22
  ACCESS • 14–24
  APPLICATION • 14–25
  ATTRIBUTES • 14–25
  CHANGEABLE • 14–25
  CLASSIFICATION • 14–26
  DATATYPE • 14–26
  DESCRIPTION • 14–26
  field expressions • 14–23
  HEADING • 14–26
  HELP • 14–27
  KEYED • 14–27
  LENGTH • 14–27
  NAME • 14–27
  NULL • 14–27
  NUMBER • 14–28
  OPTIMIZED • 14–28
  POSITION • 14–28
  PRINTMASK • 14–28
  PROMPT • 14–29
  SCREENMASK • 14–29
  VRULES • 14–29

ASK STRUCTURE options • 14–21
ASK SYSTEM, LOGICAL: VALUE statement • 9–16
ASK SYSTEM, PASS: SUCCESS statement • 9–18
ASK SYSTEM, SYMBOL: VALUE statement • 9–20
ASK SYSTEM: COMMENT statement • 9–14
ASK SYSTEM: DIRECTORY statement • 9–15, 9–16
ASK SYSTEM: MODE statement • 9–17
ASK SYSTEM: PARAMETER statement • 9–18
ASK SYSTEM: PROCESS statement • 9–19
ASK SYSTEM: RIGHTS statement • 9–20
ASK SYSTEM: USER statement • 9–21
ASK WINDOW: COLUMN statement • 9–23
ASK WINDOW: CURRENT statement • 9–24
ASK WINDOW: DATA statement • 9–25
ASK WINDOW: KEYMAP statement • 9–26
ASK WINDOW: ROW statement • 9–28
ASK WINDOW: TYPEAHEAD statement • 9–29
ASK WINDOW AREA statement • 9–22
ASK ZONEWIDTH statement • 9–30
Assigning variables • 8–1
ATN function • A–4
AT row, column in PRINT statement • 6–4
%AT row, column menu directive • 5–21
AT row, column option in INPUT • 5–7
ATTRIBUTES - field definition item • 14–25
ATTRIBUTES option in INPUT • 5–8
%AUTODISPLAY menu directive • 5–21
%AUTOVBAR menu directive • 5–21

B

_BACK in SET BACK OFF • 9–3
_BACK in SET BACK ON • 9–2
Backslash during input • 5–4
Backslash key with INPUT SCREEN • 5–19
_BACK system function • A–1
%BAR menu directive • 5–22
Basic program elements • 3–1
Bit manipulation • 3–17
BLINK - screen attribute • 6–21
BOLD - screen attribute • 6–21
BOOLEAN variables • 3–6
BOX option of CLEAR AREA • 6–21
Branching and code blocks • 11–24
Branching - description • 11–24
BREAK statement in debug system • 4–14
Brief field display • 16–19
Built-in functions • A–1

C

Callable routines in libraries • 13–1
Called routine • 13–2
Call routines written in other languages • 13–1
CALL SETUP • 16–2, 16–28
CALL statement • 13–1, 13–2, E–2
CANCEL ADD • 14–6
Cancel adding a structure record • 14–6
CANCEL EXTRACT • 14–17
CASE ELSE statement • 11–20
CASE IS statement • 11–21
CASE statement • 11–18, 11–19
CAUSE EXCEPTION • 12–1
CEIL function • A–4
Centering data, as in a heading • 8–7
Chaining programs • 11–21
CHAIN statement • 11–21
CHANGE$ function • A–4
CHANGEABLE - field definition item • 14–25
Changing an INTOUCH program • 1–6
Channel number • 15–1
Channel number in PRINT statement • 15–4
_CHANNEL system function • 15–7, A–1
CHARSET$ function • A–4
#chnl_num in PRINT statement • 15–4
CHR$ function • A–5
CLASSIFICATION - field definition item • 14–26
CLASSIFICATION file • 16–10
Classification in SETUP • 16–9
CLEAR AREA • 6–20, 6–21
  BOLD, BLINK, REVERSE, UNDERLINE attributes • 6–21
CLEAR AREA BOX • 6–21
  BOLD, BLINK, REVERSE, UNDERLINE attributes • 6–21
Clearing the screen • 6–19
CLEAR statement • 6–19
  AREA option • 6–20
    BOX option • 6–21
CLOSE #chnl_num statement • 15–4
CLOSE ALL • 14–5
CLOSE ALL statement • 15–4
CLOSE STRUCTURE • 14–4
Closing a structure • 14–4

Closing files • 15–4
COBOL
  checking define results • 16–28
  define example • 16–27
  defining record fields • 16–27
COBOL_DEF.INT • 16–27
Column on screen • 6–20
%COLUMNS menu directive • 5–22
Comma in PRINT statement • 6–2
Command completion with the TAB key • 1–11
Command presentation • 2–1
Command recall • 1–3
Commands
  ABORT • 2–5
  COMPILE • 2–13
  EDIT • 2–10
  FRAME ON/OFF • 2–12
  GO • 2–15, 4–16
  IDE • 2–12
  INTOUCH/COMPILE • 2–14
  INTOUCH/DEVELOP • 2–15
  INTOUCH/IMAGE • 2–14
  LIST and LNH • 2–9
  LIST ERROR • 2–10
  NEW • 2–3
  OLD • 2–8
  REFRESH • 2–12
  RENUMBER • 2–11
  REPLACE • 2–5
  RUN and RNH • 2–7
  SAVE • 2–4
  UNSAVE • 2–16
Command - what it does • 2–2
! comment • 4–2
Comments in programs • 4–1
COMPARE function • A–6
COMPILE command • 2–13
Compiled images • 2–14
Compile directives • 7–1
Compile error messages • C–1
Compound expressions • 3–14
  concatenated string expressions • 3–15
  conditional expressions • 3–15
  numeric expressions • 3–14
Concatenated string expressions • 3–15
Conditional expressions • 3–15
Conditional numeric expressions • 3–15
Conditionals • 11–14
Constants
  description • 3–7
  E notation • 3–7
Constants (cont'd)
  numeric - integer • 3–7
  numeric - real • 3–7
  string • 3–8
Constructs - description • 11–1
CONTINUE statement in HANDLER routine • 12–6
Continuing program lines • 3–3
Conventions
  other • xxiv
  program examples • xxiii
  user • xxii
Conventions used in manual • xxii
CONVERT$ function • A–5
CONVERT function • A–5
Coordinates on screen • 6–21
COS function • A–6
COSH function • A–6
COT function • A–6
CPAD$ function • A–6
Create a file using OPEN #chnl_num • 15–1
Create data file procedure in SETUP • 16–24
CSC function • A–7
CSET FILL statement • 8–10
CSET statement • 8–7, 8–9
Ctrl/Z during input • 5–4
Cursor positioning in PRINT statement • 6–6

D

Database engine • 16–3, 16–24, F–1
Database engines - supported • 16–1, 16–3, F–1
  ADABAS • 16–1, F–1
  dBASE III • 16–1, F–1
  DBMS • 16–1, F–1
  FASTFILE • 16–1, F–1
  INGRES • 16–1, F–1
  ORACLE • 16–1, F–1
  POISE • 16–1, F–1
  Rdb • 16–1, F–1
  RMS • 16–1, F–1
  S1032 • 16–1, F–1
  USERBASE • 16–1, F–1
Data dictionary • 16–5, 16–27
DATAFILE option in OPEN STRUCTURE • 14–4
Data management system • 14–1
Dataset • 16–4
DATA statement • 8–10
Data structure statements • 14–1
  ADD STRUCTURE/END ADD • 14–5
  ASK STRUCTURE options • 14–21

Data structure statements (cont'd)
CANCEL ADD • 14–6
CANCEL EXTRACT • 14–17
CLOSE ALL • 14–5
CLOSE STRUCTURE • 14–4
DELETE STRUCTURE • 14–8
EXCLUDE in EXTRACT STRUCTURE • 14–12
EXIT ADD • 14–7
EXIT EXTRACT • 14–18
EXTRACT STRUCTURE ... END EXTRACT • 14–9
FOR EACH ... NEXT structure • 14–14
INCLUDE in EXTRACT STRUCTURE • 14–11
LOCK/UNLOCK STRUCTURE • 14–8
OPEN STRUCTURE • 14–3
REEXTRACT STRUCTURE ... END EXTRACT • 14–18
SET STRUCTURE options • 14–37
SORT in EXTRACT STRUCTURE • 14–12
DATATYPE - field definition item • 14–26
Data type in SETUP • 16–12
Data types
integer number • 3–4, 3–5
real number • 3–4, 3–5
string • 3–4, 3–6
DATE$ function • A–7
DATE directive used with PRINT USING • 6–14
DATE format with FORMAT$ • A–14
DATE function • A–7
DAY$ function • A–9
DAYS function • A–8
dBASE III • 16–1, 16–3, F–1
DBMS • 16–1, 16–3, F–1
DBMS data files • 16–4
DBMS record extraction • 14–21
DBMS structure handling • 14–42, 14–43, 14–44
DCL symbol for INTOUCH images • 2–14
DEBUG ON/OFF • 4–8, 4–9
%DEBUG program directive • 7–3
Debug statements
BREAK • 4–14
DEBUG ON/OFF • 4–8
LIST STATS • 4–13
STATS ON/OFF • 4–12
STEP • 4–15
TRACE ON/OFF • 4–10
Debug system • 4–8
_DEBUG system function • A–1
Debug - using GO • 4–16
Decimal point • 16–14
DECLARE PREFIX • 8–3
DECLARE statement • 8–1
DYNAMIC option • 8–2
INTEGER option • 8–2
REAL option • 8–2
STRING option • 8–2
DECLARE STRUCTURE • 8–2
DECODE function • A–9
Default file extension - .INT • 3–4
DEFAULT option in INPUT • 5–9
Define fields procedure in SETUP • 16–8
Defining arrays • 10–1
Defining COBOL record fields • 16–27
Definition file • 16–5
DEG function • A–9
DEGREE • A–9
DELAY option in MESSAGE statement • 6–18
DELAY statement • 6–18
Delete access level • 16–24
DELETE STRUCTURE • 14–8
Delete structure record • 14–8
Deleting a file • 15–12
DESCRIPTION - field definition item • 14–26
Description in SETUP • 16–12
Developing INTOUCH programs • 2–15
Development cycle time • xxi
Dimension array • 10–1
DIM statement • 3–10, 10–1
Directives in programs
%DEBUG • 7–3
%INCLUDE • 7–2
%INCLUDE CONDITIONAL • 7–2
%MESSAGE • 7–1
%MESSAGE ERROR • 7–1
Directives used in menus • 5–21
Directives used with PRINT USING • 6–13
DISPATCH statement • 11–30
Displaying data • 6–1
Displaying messages • 6–16
DIV0 function • A–9
DO...END DO loop • 11–10
DO loops • 11–7
DO LOOPS
UNTIL option • 11–9
WHILE option • 11–8
DO statement in DO/LOOP • 11–7
DTYPE function • 8–2, A–9
Duplicate key field • 16–17

E

EDIT$ function • A–10
EDIT command • 2–10
Editing input response • 5–4
Editor - INTOUCH • 1–12
Edit string • A–10
EDIT symbol • 2–10
ELEMENT$ function • A–10
ELEMENTS function • A–10
ELSEID option in IF/THEN • 11–17
ELSE OPTION
  in ON...GOSUB • 11–30
ELSE option in IF/THEN • 11–16
ELSE option in ON..GOTO • 11–26
ENCODE$ function • A–11
END ADD • 14–6
END DO statement • 11–10
END EXTRACT with EXTRACT STRUCTURE • 14–9
END HANDLER with HANDLER • 12–4
END IF in IF/THEN statement • 11–14
END IF statement in IF/THEN • 11–15
END ROUTINE statement • 11–28, 11–29
END statement • 4–4
END WHEN with WHEN EXCEPTION IN/USE • 12–3
Engines - database interfaces • F–1
E notation
  in PRINT statement • 6–5
EOF option with LINE INPUT • 15–7
EPS function • A–11
ERASE in PRINT statement • 6–4
ERASE option in INPUT • 5–11
Error counter • 9–5
_ERROR in SET ERROR OFF • 9–4
_ERROR in SET ERROR ON • 9–3
Error message • 1–5
Error messages
  compile errors • C–1
  exceptions • C–4
ERROR option in MESSAGE statement • 6–17
EVAL function • A–11
Evaluation order • 3–18
Exception
  line • A–12
Exception handling • 12–1
  attached handler • 12–1
  CAUSE EXCEPTION • 12–1
  CONTINUE • 12–1
  CONTINUE in HANDLER routine • 12–6
  detached handler • 12–1
Exception handling (cont'd)
  EXIT HANDLER • 12–1
  EXIT HANDLER in HANDLER routine • 12–8
  EXLABEL$ • 12–1
  EXLINE • 12–1
  EXTEXT$ • 12–1
  EXTYPE • 12–1
  RESUME in HANDLER routine • 12–7
  RETRY • 12–1
  RETRY in HANDLER routine • 12–5
  SYSTEXT$ • 12–1
  WHEN EXCEPTION IN/USE/END WHEN • 12–2
  WHEN EXCEPTION USE • 12–3
Exception messages • C–4
Exclamation point with comments • 4–2
EXCLUDE in EXTRACT • 14–10
EXCLUDE within EXTRACT STRUCTURE • 14–12
Excluding extracted records • 14–12
Execute library routines • 13–2
EXECUTE statement • 11–13
EXIT ADD • 14–7
EXIT command • 1–8
EXIT DO statement • 11–10
EXIT during input • 5–4
EXIT EXTRACT • 14–18
EXIT FOR • 14–14
EXIT FOR statement in FOR loop • 11–4
EXIT HANDLER statement in HANDLER routine • 12–8
_EXIT in SET EXIT OFF • 9–6
_EXIT in SET EXIT ON • 9–5
_EXIT internal variable • 5–4
EXIT ROUTINE statement • 11–31
_EXIT system function • A–1
Exit when adding a structure record • 14–7
EXLABEL$ • 12–1
EXLABEL$ function • A–11
EXLINE • 12–1
EXLINE function • A–12
Expanding abbreviatioans with the TAB key • 1–12
EXP function • A–12
Expressions - description • 3–6
EXTEXT$ • 12–1
EXTEXT$ function • A–12
Extract a range of keys • 14–16
Extract criteria • 14–10
_EXTRACTED system function • A–1
Extracting records from a structure • 14–9
EXTRACT STRUCTURE, FIELD: PARTIAL KEY • 14–16

EXTRACT STRUCTURE, SET, FIELD: KEY • 14–21
EXTRACT STRUCTURE, SET, USING • 14–20
EXTRACT STRUCTURE ... END EXTRACT • 14–9
EXTRACT STRUCTURE: APPEND • 14–19, 14–42
EXTRACT STRUCTURE: KEY • 14–10, 14–15
EXTYPE • 12–1
EXTYPE function • A–12

F

Facilities, Debug system • 4–8
FALSE function • A–12
FASTFILE • 16–1, 16–3, F–1
Field attributes • 16–14
Field class • 16–9, 16–10
Field definition general information • 16–17
Field definition items
  ACCESS • 14–24
  APPLICATION • 14–25
  ATTRIBUTES • 14–25
  CHANGEABLE • 14–25
  CLASSIFICATION • 14–26
  DATATYPE • 14–26
  DESCRIPTION • 14–26
  HEADING • 14–26
  HELP • 14–27
  KEYED • 14–27
  LENGTH • 14–27
  NAME • 14–27
  NULL • 14–27
  NUMBER • 14–28
  OPTIMIZED • 14–28
  POSITION • 14–28
  PRINTMASK • 14–28
  PROMPT • 14–29
  SCREENMASK • 14–29
  VRULES • 14–29
Field expressions in ASK STRUCTURE FIELD • 14–23
Field Name in SETUP • 16–8
Field names in ASK STRUCTURE FIELD • 14–23
FIELDS
  Variable length • 16–13
Fields in structures • 3–12
Fields list display • 16–20
Fields with multiple occurrences • 3–13
File description • 15–1
File extension - .INT • 3–4
File line length • 15–2
File organization • 16–25
Files
  asking about • 15–8
  closing • 15–4
  deleting • 15–12
  inputing data • 15–5
  inputting multiple variables • 15–6
  line input • 15–6
  opening • 15–1
  setting characteristics • 15–10
FILESPEC$ function • A–12
File specification • 2–4
File specifications • 17–1
FINDFILE$ function • A–13
FOR/NEXT loop • 11–1
FOR EACH ... NEXT structure • 14–14
FOR EACH loop • 14–14
FOR loop
  index variable • 11–2
  initial expression • 11–2
  limit expression • 11–2
  nesting loops • 11–4
  STEP option • 11–3
FORMAT$ function • A–14
  DATE format options • A–14
  ROTATE option • A–15
  TIME format • A–16
  ZIPCODE format • A–16
Format characters in PRINT USING • 6–8
Format fields in PRINT USING • 6–7
FOR statement in FOR/NEXT loop • 11–1
Fourth-generation database language • xxi
FP function • A–16
FRAME ON/OFF • 6–16
FRAME ON/OFF command • 2–12
Full field display • 16–20
FULLTIME$ function • A–16
Functions - Non-System
  ABS • A–3
  ACOS • A–3
  ANGLE • A–3
  ASCII • A–4
  ASIN • A–4
  ATN • A–4
  CEIL • A–4
  CHANGE$ • A–4
  CHARSET$ • A–4
  CHR$ • A–5
  COMPARE • A–6
  CONVERT • A–5

Functions - Non-System (cont'd)
CONVERT$ • A–5
COS • A–6
COSH • A–6
COT • A–6
CPAD$ • A–6
CSC • A–7
DATE • A–7
DATE$ • A–7
DAY$ • A–9
DAYS • A–8
DECODE • A–9
DEG • A–9
DIV0 • A–9
DTYPE • A–9
EDIT$ • A–10
ELEMENTS • A–10
ELEMENTS$ • A–10
ENCODE$ • A–11
EPS • A–11
EVAL • A–11
EXLABEL$ • A–11
EXLINE • A–12
EXP • A–12
EXTEXT$ • A–12
EXTYPE • A–12
FALSE • A–12
FILESPEC$ • A–12
FINDFILE$ • A–13
FORMAT$ • A–14
FP • A–16
FULLTIME$ • A–16
HASH$ • A–17
INT • A–17
INTEGER • A–17
IP • A–18
ITEM • A–18
LBOUND • A–18
LCASE$ • A–18
LEFT[$] • A–18
LEN • A–18
LOG • A–18
LOG10 • A–19
LOG2 • A–19
LPAD$ • A–19
LTRIM$ • A–19
MATCH • A–19
MAX • A–19
MAXLEN • A–20
MAXNUM • A–20
MAXSIZE • A–20

Functions - Non-System (cont'd)
MID[$] • A–20
MIN • A–20
MOD • A–20
ORD • A–20
ORDNAME$ • A–21
PARSE$ • A–21
PATTERN • A–21
PI • A–23
PIECE$ • A–23
PIECES • A–23
POS • A–23
PRETTY$ • A–23
QUOTE$ • A–24
RAD • A–24
REAL • A–24
REMAINDER • A–25
REPEAT$ • A–25
REPLACE$ • A–25
RIGHT[$] • A–25
RND • A–25
ROUND • A–26
RPAD$ • A–26
RTRIM$ • A–26
SCAN • A–26
SEC • A–26
SECONDS • A–26
SEG$ • A–27
SGN • A–27
SIN • A–27
SINH • A–27
SIZE • A–27
SKIP • A–27
SORT$ • A–28
SPACE$ • A–28
SQR • A–28
STR$ • A–28
SYSTEXT$ • A–28
TAB • A–29
TAN • A–29
TANH • A–29
TIME • A–29
TIME$ • A–29
TIME(5) • A–29
trim$ • A–30
TRUE • A–30
TRUNCATE • A–30
UBOUND • A–30
UCASE$ • A–30
UNQUOTE$ • A–30
VAL • A–31
VALID • A–31

Functions - Non-System (cont'd)
WRAP$ • A–36
XLATE$ • A–36
Functions - System
_BACK • A–1
_CHANNEL • A–1
_DEBUG • A–1
_EXIT • A–1
_EXTRACTED • A–1
_HELP • A–2
_INTEGER • A–2
_REAL • A–2
_REPLY • A–2
_STATUS • A–2
_STRING • A–2
_TERMINATOR • A–3

G

Getting started in INTOUCH • 1–1
GO command • 2–15, 4–16
GOSUB/RETURN statements • 11–27
GOSUB levels • 11–28
GOSUBs nested • 11–28
GOSUB statement • 11–27
GOSUB target limits • 11–29
GOTO command • 1–8
GOTO statement • 11–24
GOTO target limits • 11–26

H

HALT statement • 4–6
HANDLER/END HANDLER with WHEN EXCEPTION USE • 12–4
HASH$ function • A–17
HEADING - field definition item • 14–26
%HEADING menu directive • 5–23
HELP facility • 1–3
HELP - field definition item • 14–27
_HELP in SET HELP OFF • 9–7
_HELP in SET HELP ON • 9–6
Help in SETUP • 16–3
HELP or Help key during input • 5–4
_HELP system function • A–2
Help text • 16–16
HYPHEN directive used with PRINT USING • 6–14

I

IDE command • 2–12
IF/THEN ... END IF • 11–14
IF construct • 11–14, 11–15
IMPLICIT prefix for variables • 8–3
.INC extension • 7–2
.INC - INTOUCH included file extension • 17–1
%INCLUDE CONDITIONAL program directive • 7–2
INCLUDE in EXTRACT • 14–10
%INCLUDE program directive • 7–2
INCLUDE within EXTRACT STRUCTURE • 14–11
Including extracted records • 14–11
Indenting with the TAB key • 1–10
Indexed file • 16–17, 16–25
Index variable in FOR loop • 11–2
INGRES • 16–1, 16–3, F–1
INGRES data files • 16–4
Initial expression in FOR loop • 11–2
INPUT #chnl_num statement • 15–5
Input from a terminal • 5–1
INPUT statement • 5–1, 5–2
AREA option • 5–12
AT row, column option • 5–7
ATTRIBUTES option • 5–8
DEFAULT option • 5–9
default prompt and text • 5–6
ERASE option • 5–11
LENGTH option • 5–9
MENU option • 5–20
options • 5–3
PROMPT option • 5–7
SCREEN option • 5–13
TIMEOUT option • 5–12
VALID option • 5–11
Input styles • 5–2
Inputting multiple variables • 5–5
Inputting strings • 5–5
Input to a file • 15–5, 15–6
INTEGER function • A–17
Integer number • 3–5
_INTEGER system function • A–2
Interrupting an INTOUCH program • 1–6
INT function • A–17
.INT - INTOUCH program extension • 17–1

INTOUCH/COMPILE command • 2–14
INTOUCH/DEVELOP command • 2–15
INTOUCH/IMAGE command • 2–14
INTOUCH commands • 2–1
INTOUCH command - what it does • 2–2
INTOUCH compile errors • C–1
INTOUCH database interfaces • F–1
INTOUCH data management system • 14–1
INTOUCH editor • 1–12, 2–10
INTOUCH environment • xxi
  accessing system utilities • 1–2
  backing up • 1–8
  changing an INTOUCH program • 1–6
  error message • 1–5
  examples of using INTOUCH • 1–4
  exiting • 1–8
  getting help • 1–3
  getting started • 1–1
  interrupting an INTOUCH program • 1–6, 1–7
  INTOUCH editor • 1–12
  message lines • 1–10
  program example • 1–5
  recalling previous commands • 1–3
  resuming INTOUCH program execution • 1–8
  running an INTOUCH program • 1–5
  Tab key features • 1–10
INTOUCH exception messages • C–4
INTOUCH program description • 3–1
INTOUCH program elements • 3–1
INTOUCH prompt • 1–1
INTOUCH reserved words • 3–2, B–1
INTOUCH RUN file • 2–13
INTOUCH screen • 6–16
INTOUCH screen frame • 1–1
INTOUCH structure • 14–1
INTOUCH_EDITOR logical • 2–10
.INT program extension • 3–4
.INT_IMG • 2–14
.INT_IMG_AXP • 2–14
IP function • A–18
ITEM function • A–18
%ITEMS menu directive • 5–23
ITERATE • 14–14
ITERATE DO statement • 11–12
ITERATE FOR statement in FOR loop • 11–6

K

KEYED - field definition item • 14–27
Key field • 16–17
KEY INPUT statement • 5–2, 5–28
KEY option in EXTRACT STRUCTURE • 14–10, 14–15
Key range extraction • 14–16
KILL statement • 15–12

L

Labels in programs • 3–4
LBOUND function • A–18
LCASE$ function • A–18
LCASE directive used with PRINT USING • 6–14
LEFT[$] function • A–18
LEN function • A–18
LENGTH - field definition item • 14–27
LENGTH option in INPUT • 5–9
LET statement in ADD STRUCTURE • 14–6
LET statment • 8–6
Libraries • 13–1
  Creating for INTOUCH • 13–1, E–1
  User-defined library • 13–1
  VMS supplied library • 13–1
Library routine • 13–2
  optional arguments • 13–2
  passing mechanisms • 13–3
  using _INTEGER function with CALL • 13–3
  using _REAL function with CALL • 13–3
LIBRARY statement • 13–1, E–2
Limit expression in FOR loop • 11–2
Line in a program • 3–2
LINE INPUT #chnl_num, EOF • 15–7
LINE INPUT #chnl_num statement • 15–6
LINE INPUT statement • 5–6
LINE INPUT statment • 5–2
Line length of file record • 15–2
Line number range • 3–3
Line terminators • 3–2
  Enter • 3–2
  Return • 3–2
LIST and LNH commands • 2–9
LIST command • 1–7
LIST ERROR command • 2–10
Listing INTOUCH program lines • 1–7
LIST STATS in debug system • 4–13
LNH and LIST command • 2–9
LOCK/UNLOCK STRUCTURE • 14–8

Lock a structure record • 14–8
LOCK option in OPEN STRUCTURE • 14–4
%LOCKSTEP menu directive • 5–24
LOG10 function • A–19
LOG2 function • A–19
LOG function • A–18
Logical INTOUCH_EDITOR • 2–10
Logical operators • 3–16
Loop descriptions • 11–1
LOOP statement in DO/LOOP • 11–7
Low bound in array • 10–4
LPAD$ function • A–19
LSET FILL statement • 8–10
LSET statement • 8–7, 8–8
LTRIM$ function • A–19

M

MATCH function • A–19
MAX function • A–19
MAXLEN function • A–20
MAXNUM function • A–20
MAXSIZE function • A–20
%MAXWIDTH menu directive • 5–24
%MENUBAR menu directive • 5–24
Menu data value • 5–21
Menu descriptions • 5–21
Menu directives • 5–21
  %AT row, column • 5–21
    CENTER keyword • 5–21
  %AUTODISPLAY • 5–21
  %AUTOVBAR • 5–21
  %BAR • 5–22
  %COLUMNS • 5–22
  %HEADING • 5–23
  %ITEMS • 5–23
  %LOCKSTEP • 5–24
  %MAXWIDTH • 5–24
  %MENUBAR • 5–24
  %MESSAGE • 5–25
  %MULTI • 5–25
  %REPLACE • 5–25
  %SIZE number • 5–26
  %SPLIT • 5–26
  %TITLE • 5–26
  %VBAR • 5–26
  %WIDTH • 5–27
Menu elements • 5–20
menu item default • 5–10
Menu items • 5–20
MENU option in INPUT • 5–20
Menu path - tracing • 5–10
Menu user interface • 5–27
MESSAGE ERROR incrementing error counter • 9–5
%MESSAGE ERROR program directive • 7–1
Message lines • 1–10
%MESSAGE menu directive • 5–25
%MESSAGE program directive • 7–1
Messages
  compile errors • C–1
  exceptions • C–4
MESSAGE statement • 6–16
  DELAY option • 6–18
  ERROR option • 6–17
Message trapping • 6–22
MID[$] function • A–20
MIN function • A–20
MOD function • A–20
Modify structure general information • 16–22
Modify structure procedure in SETUP • 16–21
Modify structure security information • 16–23
Monitoring program execution • 4–9
%MULTI menu directive • 5–25
Multiple occurrence fields • 3–13
Multiple statements on a single line • 3–2
Multiple variables in LINE INPUT #chnl_num • 15–8

N

NAME - field definition item • 14–27
Nested DO/LOOP • 11–8
Nesting FOR/NEXT loops • 11–4
NEW command • 2–3
NEXT statement in FOR/NEXT loop • 11–1
NORETURN option of PASS • 11–23
NOWAIT option of PASS • 11–23
NULL - field definition item • 14–27
Null string • 3–8
NUMBER - field definition item • 14–28
Numeric expressions • 3–14
Numeric format characters in PRINT USING • 6–8

O

OLD command • 2–8
ON...GOSUB
  ELSE option • 11–30
ON..GOSUB..[ELSE] statement • 11–28
ON..GOTO..[ELSE] statement • 11–25, 11–26
OPEN #chnl_num options • 15–2
  ACCESS • 15–2
  OPTIMIZE OFF • 15–3
  UNFORMATTED • 15–3
  UNIQUE • 15–3
OPEN #chnl_num statement • 15–1
Opening a structure • 14–3
Opening files • 15–1
OPEN STRUCTURE • 14–3
  ACCESS option • 14–4
  DATAFILE option • 14–4
  LOCK option • 14–4
OPTIMIZED - field definition item • 14–28
OPTIMIZE OFF option in OPEN #chnl_num • 15–3
OPTION BASE statement • 8–5, 10–4
OPTION COMMUNICATE ON/OFF • 6–22
OPTION REQUIRE DECLARE statement • 8–5
ORACLE • 16–1, 16–3, F–1
ORACLE data files • 16–5
Order of evaluation • 3–18
ORD function • A–20
ORDNAME$ function • A–21
Original file • 2–4

P

Page size of screen • 9–10
PARSE$ function • A–21
PARTIAL KEY option in EXTRACT STRUCTURE • 14–16
PASS NORETURN statement • 11–23
PASS NOWAIT statement • 11–23
PASS statement • 11–22
PATTERN function • A–21
PIECE$ function • A–23
PIECES function • A–23
PI function • A–23
POISE • 16–1, 16–3, F–1
POS function • A–23
POSITION - field definition item • 14–28
PREFIXED variable • 8–3
PRETTY$ function • A–23
Primary key • 16–17, 16–26
PRINT #chnl_num • 15–4
Printing data • 6–1
Printing numbers with PRINT statement • 6–5
Printing to a file • 15–4
Printing to a terminal • 6–1
Print mask • 16–15
PRINTMASK - field definition item • 14–28
PRINT statement • 6–1, 16–15
  AT row, column • 6–4
  cursor positioning • 6–6
  ERASE option • 6–4
  item separater • 6–2
    comma • 6–2
    semicolon • 6–2
  printing numbers • 6–5
  TAB function • 6–3
  USING option • 6–7
  WIDE option • 6–5
PRINT USING • 6–7
  directives • 6–13
  format characters • 6–7
  numeric format characters • 6–8
  string format characters • 6–8
PRINT USING directives
  DATE • 6–14
  HYPHEN • 6–14
  LCASE • 6–14
  ROTATE • 6–15
  TIME • 6–15
  UCASE • 6–14
  ZIPCODE • 6–16
Print zones • 6–2
  comma • 6–2
Program description • 3–1
Program directives • 7–1
Program elements • 3–1
Program example • 1–5
Program examples • xxiii
Program labels • 3–4
Program line • 3–2
Program line numbers • 3–3
PROGRAM statement • 4–4
PROMPT - field definition item • 14–29
PROMPT option in INPUT • 5–7
Prompt text • 16–15

Q

QUOTE$ function • A–24

R

RAD function • A–24
RANDOMIZE statement • 4–7
Rdb • 16–1, 16–3, F–1
Rdb data files • 16–4
Read access level • 16–23
READ statement • 8–10
REAL function • A–24
Real number • 3–5
_REAL system function • A–2
Recalling commands when in INTOUCH • 1–3
Record management system • 16–3
Records in structures • 3–12
Record size • 16–26
REDIM statement • 3–10, 10–3
REEXTRACT STRUCTURE ... END EXTRACT • 14–18
REFRESH command • 2–12
Relational operators • 3–15
REMAINDER function • A–25
REM statement • 4–1
RENUMBER command • 2–11
REPEAT • 14–14
REPEAT$ function • A–25
REPEAT DO statement • 11–11
REPEAT FOR statement in FOR loop • 11–5
REPEAT ROUTINE statement • 11–32
REPLACE$ function • A–25
REPLACE command • 2–5
%REPLACE menu directive • 5–25
_REPLY system function • A–2
Report heading • 16–15
REQ • 5–17
Reserved words • 3–2, B–1
RESTORE statement • 8–12
RESUME statement in HANDLER routine • 12–7
Resuming INTOUCH program execution • 1–8
RETRY statement in HANDLER routine • 12–5
Return key with INPUT SCREEN • 5–19
RETURN statement • 11–28, 11–29
REVERSE - screen attribute • 6–21
RIGHT[$] function • A–25
RMS • 16–1, 16–3, F–1
RMS file • 16–24
RMS indexed file • 16–17
RMS interface • xxi
RND function • 4–7, A–25
RNH and RUN commands • 2–7
RNH command • 1–5
ROTATE directive used with PRINT USING • 6–15
ROTATE option with FORMAT$ • A–15
ROUND function • A–26
ROUTINE/END ROUTINE statements • 11–31
ROUTINE statement • 11–28
Row on screen • 6–20
RPAD$ function • A–26
RSET FILL statement • 8–10
RSET statement • 8–7, 8–8
RTRIM$ function • A–26
RUN and RNH commands • 2–7
RUN command • 1–5
RUN file • 2–13
Running an INTOUCH program • 1–5
Running compiled images • 2–14
Runtime exception messages • C–4

S

S1032 • 16–1, 16–3, F–1
SAVE command • 2–4
Scale factor • 16–14
SCAN function • A–26
Screen attributes
  BLINK • 6–21
  BOLD • 6–21
  REVERSE video • 6–21
  UNDERLINE • 6–21
Screen clearing • 6–19
Screen column • 6–20
Screen coordinates • 6–21
Screen format characters • 5–14
Screen format options • 5–16
  AJ - auto jump • 5–17
  AT row, column • 5–17
  BOLD, BLINK, UNDERLINE, REVERSE • 5–19
  DIGITS • 5–17
  ELAPSED time • 5–18
  LCASE • 5–16
  NOECHO • 5–16
  REQ - required • 5–17

Screen format options (cont'd)
UCASE • 5–16
VALID • 5–18
Screen frame • 1–1
Screen mask • 16–16
SCREENMASK - field definition item • 14–29
Screen message trapping • 6–22
SCREEN option in INPUT • 5–13
SCREEN option - user response • 5–19
Screen row • 6–20
SEC function • A–26
SECONDS function • A–26
SEG$ function • A–27
Segmented key field • 16–18
SELECT CASE construct • 11–14
SELECT CASE statement • 11–18
Semicolon in PRINT statement • 6–2
Sequential file • 16–25
SET #chnl_num options • 15–11
CURRENT • 15–11
MARGIN • 15–11
ZONEWIDTH • 15–11
SET #chnl_num statement • 15–10
SET, FIELD: KEY option in EXTRACT STRUCTURE • 14–21
SET, USING option in EXTRACT STRUCTURE • 14–20
SET/ASK statements • 9–1
SET AUTOEXIT statement • 9–1
SET BACK OFF statement • 9–2
SET BACK ON statement • 9–2
SET ERROR OFF statement • 9–4
SET ERROR ON statement • 9–3
SET EXIT OFF statement • 9–6
SET EXIT ON statement • 9–5
SET HELP OFF statement • 9–7
SET HELP ON statement • 9–6
SET MARGIN statement • 9–8
SET MESSAGELINE statement • 9–9
SET PORT OFF statement • 9–11
SET PORT ON statement • 9–10
SET SCROLL statement • 9–12
SET STRUCTURE #string_expr... • 14–36
SET STRUCTURE, FIELD: KEY • 14–39
SET STRUCTURE, FIELD: PARTIAL KEY • 14–39
SET STRUCTURE, SET, USING • 14–43
SET STRUCTURE, SET, USING, FIELD: KEY • 14–43
SET STRUCTURE, SET, USING: EXPRESSION 'owner' • 14–42
SET STRUCTURE, SET: EXPRESSION 'owner' • 14–43
SET STRUCTURE: CURRENT • 14–37
SET STRUCTURE: EXTRACTED 0 • 14–42
SET STRUCTURE: ID • 14–40
SET STRUCTURE: POINTER • 14–41
SET STRUCTURE: RECORD • 14–41
SET STRUCTURE:TIMEOUT • 14–36
SET STRUCTURE options • 14–37
SET SYSTEM, LOGICAL: VALUE statement • 9–17
SET SYSTEM, SYMBOL: VALUE statement • 9–21
SET SYSTEM: COMMENT statement • 9–15
SET SYSTEM: PROCESS statement • 9–19
SETUP
Access rules • 16–16
array elements • 16–14
Brief Field Display • 16–19
Call • 16–2
Classification • 16–9
Create • 16–1
Create data file procedure • 16–24
Database engine • 16–3
Data dictionary • 16–5
Dataset • 16–4
Data Type • 16–12
Define • 16–1
Define fields procedure • 16–8
Description • 16–12
Ending position • 16–13
Exit • 16–1
Field length • 16–13
Field Name • 16–8
File organization • 16–25
First position • 16–13
Full Field Display • 16–20
Help • 16–3
Help text • 16–16
List Fields Display • 16–20
Modify structure • 16–1
Modify structure general information • 16–22
Modify structure procedure • 16–21
Modify structure security information • 16–23
occurrences • 16–14
Primary key • 16–26
Print mask • 16–15
Procedures • 16–7
Prompt text • 16–15
Record size • 16–26
Report heading • 16–15
Screen mask • 16–16
Semantics • 16–14

SETUP (cont'd)
Show • 16–1
Show procedure • 16–18
Structure Display • 16–21
Structure name • 16–3
Validation rules • 16–17
SETUP DEFINE screen • 16–8, 16–11
SETUP procedures • 16–1
SETUP procedures menu • 16–6
SETUP prompts • 16–2, 16–3
SETUP routine • 16–28
SETUP Routine • 16–1, F–1
SET WINDOW: COLUMN statement • 9–23
SET WINDOW: CURRENT statement • 9–24
SET WINDOW: DATA statement • 9–25
SET WINDOW: KEYMAP statement • 9–26
SET WINDOW: ROW statement • 9–29
SET WINDOW: TYPEAHEAD statement • 9–30
SET WINDOW AREA statement • 9–22
SET WINDOW KEYSTROKE: VALUE statement • 9–27
SET ZONEWIDTH statement • 9–31
SGN function • A–27
Show procedure in SETUP • 16–18
SIN function • A–27
SINH function • A–27
SIZE function • A–27
%SIZE menu directive • 5–26
SKIP function • A–27
SORT
ASCENDING • 14–13
DESCENDING • 14–13
SORT$ function • A–28
SORT in EXTRACT • 14–10
Sorting extracted records • 14–12
Sort keys • 14–10
SORT within EXTRACT STRUCTURE • 14–12
SPACE$ function • A–28
Specifications manual • xxi
Spelling correction with the TAB key • 1–11
%SPLIT menu directive • 5–26
SQR function • A–28
Statement presentation • 2–1
Statements
ADD STRUCTURE/END ADD • 14–5
ASK #chnl_num • 15–8
ASK/SET MARGIN • 9–8
ASK/SET MESSAGELINE • 9–9
ASK/SET SEED • 9–13
ASK/SET SYSTEM, LOGICAL: VALUE • 9–16, 9–17
Statements (cont'd)
ASK/SET SYSTEM, SYMBOL: VALUE • 9–20, 9–21
ASK/SET SYSTEM: COMMENT • 9–14, 9–15
ASK/SET SYSTEM: DIRECTORY • 9–15, 9–16
ASK/SET SYSTEM: PROCESS • 9–19
ASK/SET WINDOW: COLUMN • 9–23
ASK/SET WINDOW: CURRENT • 9–24
ASK/SET WINDOW: DATA • 9–25
ASK/SET WINDOW: KEYMAP • 9–26
ASK/SET WINDOW: ROW • 9–28, 9–29
ASK/SET WINDOW: TYPEAHEAD • 9–29, 9–30
ASK/SET WINDOW AREA • 9–22
ASK/SET ZONEWIDTH • 9–30, 9–31
ASK ERRORS • 9–4
ASK KEYSTROKES • 9–7
ASK PAGESIZE • 9–10
ASK RESPONSES • 9–12
ASK STRUCTURE options • 14–21
ASK SYSTEM, PASS: SUCCESS • 9–18
ASK SYSTEM: MODE • 9–17
ASK SYSTEM: PARAMETER • 9–18
ASK SYSTEM: RIGHTS • 9–20
ASK SYSTEM: USER • 9–21
BREAK • 4–14
CALL • 13–2
CANCEL ADD • 14–6
CANCEL EXTRACT • 14–17
CASE • 11–19
CASE ELSE • 11–20
CASE IS • 11–21
CAUSE EXCEPTION • 12–1
CHAIN • 11–21
CLEAR • 6–19
CLOSE #chnl_num • 15–4
CLOSE ALL • 14–5, 15–4
CLOSE STRUCTURE • 14–4
CONTINUE in HANDLER routine • 12–6
CSET • 8–9
CSET FILL • 8–10
DATA • 8–10
DEBUG ON/OFF • 4–8, 4–9
DECLARE • 8–1
DECLARE PREFIX • 8–3
DECLARE STRUCTURE • 8–2
DELAY • 6–18
DELETE STRUCTURE • 14–8
DIM • 10–1
DISPATCH • 11–30
DO in DO/LOOP • 11–7
END • 4–4
END DO • 11–10

Statements (cont'd)
EXCLUDE in EXTRACT STRUCTURE • 14–12
EXECUTE • 11–13
EXIT ADD • 14–7
EXIT DO • 11–10
EXIT EXTRACT • 14–18
EXIT FOR • 14–14
EXIT FOR in FOR/NEXT loop • 11–4
EXIT HANDLER in HANDLER routine • 12–8
EXIT ROUTINE • 11–31
EXTRACT STRUCTURE ... END EXTRACT • 14–9
FOR EACH ... NEXT structure • 14–14
FOR in FOR/NEXT loop • 11–1
FRAME ON/OFF • 6–16
GOSUB/RETURN • 11–27
GOTO • 11–24
HALT • 4–6
IF/THEN...END IF • 11–14
INCLUDE in EXTRACT STRUCTURE • 14–11
INPUT • 5–1
INPUT #chnl_num • 15–5
ITERATE • 14–14
ITERATE DO • 11–12
ITERATE FOR in FOR/NEXT loop • 11–6
KILL a file • 15–12
LET • 8–6
LIBRARY • 13–1
LINE INPUT #chnl_num • 15–6
LIST STATS • 4–13
LOCK/UNLOCK STRUCTURE • 14–8
LOOP in DO/LOOP • 11–7
LSET • 8–8
LSET FILL • 8–10
MESSAGE • 6–16
NEXT in FOR/NEXT loop • 11–1
ON..GOSUB..[ELSE] • 11–28
ON..GOTO..[ELSE] • 11–25
OPEN #chnl_num: NAME • 15–1
OPEN STRUCTURE • 14–3
OPTION BASE • 8–5, 10–4
OPTION COMMUNICATE ON/OFF • 6–22
OPTION REQUIRE DECLARE • 8–5
PASS [NOWAIT:/NORETURN:] • 11–22
PRINT • 6–1
PRINT #chnl_num • 15–4
PROGRAM • 4–4
RANDOMIZE • 4–7
READ • 8–10
REDIM • 10–3
REEXTRACT STRUCTURE ... END EXTRACT • 14–18

Statements (cont'd)
REM • 4–1
REPEAT • 14–14
REPEAT DO • 11–11
REPEAT FOR in FOR/NEXT loop • 11–5
REPEAT ROUTINE • 11–32
RESTORE • 8–12
RESUME in HANDLER routine • 12–7
RETRY in HANDLER routine • 12–5
ROUTINE/END ROUTINE • 11–31
RSET • 8–8
RSET FILL • 8–10
SELECT CASE/END SELECT • 11–18
SET #chnl_num • 15–10
SET AUTOEXIT • 9–1
SET BACK ON/OFF • 9–2
SET ERROR ON/OFF • 9–3, 9–4
SET EXIT ON/OFF • 9–5, 9–6
SET HELP ON/OFF • 9–6, 9–7
SET PORT ON/OFF • 9–10, 9–11
SET SCROLL • 9–12
SET STRUCTURE options • 14–37
SET WINDOW KEYSTROKE: VALUE • 9–27
SORT in EXTRACT STRUCTURE • 14–12
STATS ON/OFF • 4–12, 4–13
STEP • 4–15
STOP • 4–5
TRACE ON/OFF • 4–10, 4–12
WHEN EXCEPTION IN/USE/END WHEN • 12–2
WHEN EXCEPTION USE • 12–3
STATISTICS feature in DEBUG system • 4–9, 4–10
STATS ON/OFF in debug system • 4–12, 4–13
_STATUS system function • A–2
STEP option in FOR/NEXT loop • 11–3
STEP statement in debug system • 4–15
STEP statement with DEBUG • 4–9
STOP statement • 4–5
STR$ function • A–28
STR file change • 16–22, 16–24
String data • 3–6
String delimiters • 3–8
String expressions - concatenated • 3–15
String format characters in PRINT USING • 6–8
_STRING system function • 5–10, A–2
Structure • 14–1
structure contents • 16–3
Structure display • 16–21
Structure name • 16–3
Structure references • 3–11
Structures • 3–11
field names • 3–12
fields • 3–12

Structures (cont'd)
records • 3–12
Structure security level • 16–23
Structure types • 14–32
Submenu indicator • 5–21
Substrings • 3–11
System functions • A–1
System utilities access • 1–2
SYSTEXT$ • 12–1
SYSTEXT$ function • A–28

T

TAB function • A–29
TAB in PRINT statement • 6–3
TAB key - usage
completing commands • 1–11
correct spelling • 1–11
expanding abbreviations • 1–12
indenting • 1–10
TAB key with INPUT SCREEN • 5–19
TAN function • A–29
TANH function • A–29
Target limits for GOSUB • 11–29
Target limits for GOTO • 11–26
Terminal message trapping • 6–22
_TERMINATOR system function • 5–28, A–3
Tilde - use for highlighting • 4–3
TIME$ function • A–29
TIME(5) function • A–29
TIME directive used with PRINT USING • 6–15
TIME format with FORMAT$ • A–16
TIME function • A–29
TIMEOUT option in INPUT • 5–12
%TITLE menu directive • 5–26
TPU editor • 2–10
TRACE feature in DEBUG system • 4–9, 4–10
TRACE ON/OFF in debug system • 4–10, 4–12
Translate • A–36
Trapping messages • 6–22
TRIM$ function • A–30
Trim leading spaces • A–19
TRUE function • A–30
TRUNCATE function • A–30
TTI_PRINTER_TYPE symbol • 9–11
TTI_RUN:COBOL_DEF.INT • 16–27
TTI_RUN:ISHOW.COM • 9–15

U

UBOUND function • A–30
UCASE$ function • A–30
UCASE directive used with PRINT USING • 6–14
UNDERLINE - screen attribute • 6–21
UNFORMATTED option in OPEN #chnl_num • 15–3
Unique key field • 16–17
UNIQUE option in OPEN #chnl_num • 15–3
Unlock a structure record • 14–8
UNQUOTE$ function • A–30
UNSAVE command • 2–16
UNTIL option in DO/LOOP • 11–8, 11–9, 11–10
UP-arrow during input • 5–4
Update access level • 16–24
USERBASE • 16–1, 16–3, F–1
User conventions • xxii
USE with WHEN EXCEPTION IN • 12–3
USING option in PRINT statement • 6–7
Using the INTOUCH screen • 6–16

V

VAL function • A–31
Validation rules • 5–11, 16–17
ALLOW • A–31
CHARACTERS • A–32
CODE • A–33
DATE • A–33
DATE DMONCY • A–33
DATE DMONY • A–33
DECIMALS • A–32
DIGITS • A–32
DISALLOW • A–31
EXPRESSION • A–34
FILTER • A–35
CHANGE • A–35
LCASE$ • A–35
LTRIM$ • A–35
REMOVE • A–35
REPLACE • A–35
RESTORE • A–35
RTRIM$ • A–35
TRIM$ • A–35
UCASE$ • A–35
FULLTIME • A–33
INTEGER • A–32
INTEGER WORD • A–32

Validation rules (cont'd)
LCASE • A–32
LENGTH • A–32
LETTERS • A–32
MAXLENGTH • A–32
MENU • A–34
MINLENGTH • A–31
NOCHARACTERS • A–32
NUMBER • A–32
PATTERN • A–34
PRINTMASK • A–34
REQUIRED • A–34
UCASE • A–32
VRULES • A–34
YES/NO • A–34
VALID function • 16–17, A–31
VALID option in INPUT • 5–11
Variable length fields • 16–13
Variables - description • 3–9
%VBAR menu directive • 5–26
VMS file specifications • 17–1
VMS languages • 13–1
VMS library • 13–1
VMS utilities access • 1–2
VRULES - field definition item • 14–29

W

WHEN EXCEPTION IN • 12–2
WHEN EXCEPTION USE • 12–3
WHILE option in DO/LOOP • 11–8, 11–9, 11–10
WIDE option in PRINT statement • 6–5
%WIDTH menu directive • 5–27
WRAP$ function • A–36
Wrapped text example • 5–13
Write access level • 16–24
Writing to a file • 15–4

X

XLATE$ function • A–36

Z

ZIPCODE directive used with PRINT USING • 6–16
ZIPCODE format with FORMAT$ • A–16
Zonewidth in SET MARGIN • 9–8