VAX® INTOUCH®

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 REMOTE TAPE FACILITY

DYNAMIC LOAD BALANCER PLUS FRAGMENTATION CONTROLLER REMOTE DEVICE FACILITY

The following are trademarks of System Management Software, Inc., and may be used only to describe products of System Management Software, Inc.:

RMS Expert

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 VMS

Last Revision date: 14-Feb-1994

Copyright ©1992, 1994 Touch Technologies, Inc.

Contents

PREFACE xxi

CHAPTER 1 GETTING STARTED WITH INTOUCH 1–1

1.1 THE INTOUCH ENVIRONMENT 1–1
1.1.1 Command Recall History 1–2
1.1.2 Using the Tab Key for Indenting 1–3
1.1.3 Command Completion 1–3
1.1.4 Spelling Correction 1–4
1.1.5 Abbreviations 1–4

1.2 USING THE INTOUCH ENVIRONMENT–SOME EXAMPLES 1–5

1.3 TRYING OUT INTOUCH PROGRAMS 1–6

1.4 TO RUN AN INTOUCH PROGRAM 1–6

1.5 TO GET HELP ON INTOUCH 1–7
1.5.1 The EXIT Command 1–9

1.6 TO INTERRUPT AN INTOUCH PROGRAM 1–9

1.7 TO CHANGE AN INTOUCH PROGRAM 1–11

CHAPTER 2 SUMMARY OF INTOUCH COMMANDS 2–1

2.1 ABOUT INTOUCH COMMANDS 2–2

2.2 NEW 2–2

2.3 SAVE 2–4

Contents

2.4	RUN	2–5
2.5	REPLACE	2–7
2.6	ABORT	2–8
2.7	OLD	2–9
2.8	LIST	2–10
2.9	LIST ERROR	2–13
2.10	EDIT	2–14
2.11	RENUMBER	2–15
2.12	FRAME ON | OFF	2–16
2.13	REFRESH	2–17
2.14	IDE	2–18
2.15	COMPILE	2–19
2.16	GO	2–21
2.17	UNSAVE	2–22
2.18	%INCLUDE	2–23
2.19	%INCLUDE CONDITIONAL	2–24
2.20	%DEBUG	2–24

2.21 DELAY 2–24

CHAPTER 3 DIRECTIVES 3–1

3.1 %MESSAGE ERROR 3–1

3.2 %MESSAGE 3–1

3.3 %INCLUDE 3–2

CHAPTER 4 ASSIGNING VARIABLES 4–1

4.1 DECLARE 4–1
4.1.1 DECLARE STRUCTURE 4–2
4.1.2 DECLARE PREFIX 4–3

4.2 OPTION 4–5
4.2.1 OPTION REQUIRE DECLARE 4–5
4.2.2 OPTION BASE 4–5

4.3 [LET] 4–6

4.4 LSET, RSET AND CSET 4–8
4.4.1 LSET 4–8
4.4.2 RSET 4–9
4.4.3 CSET 4–10

4.5 LSET, RSET, CSET FILL 4–10

4.6 READ/DATA STATEMENTS 4–11
4.6.1 READ 4–11

4.7 RESTORE STATEMENT 4–13
4.7.1 RESTORE 4–13

Contents

CHAPTER 5 INPUTTING DATA 5–1

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

5.2 INPUT OPTIONS 5–4

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

5.4 INPUTTING STRINGS 5–5

5.5 INPUTTING MULTIPLE VARIABLES 5–6
5.5.1 Simple Input Statement 5–6
5.5.2 LINE INPUT Statement 5–7

5.6 "PROMPT TEXT" 5–7

5.7 PROMPT OPTION 5–8

5.8 AT OPTION 5–8

5.9 ATTRIBUTES OPTION 5–9

5.10 LENGTH OPTION 5–9

5.11 DEFAULT OPTION 5–10
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 5–13

5.16 SCREEN OPTION 5–14
5.16.1 Screen Format Characters 5–15
5.16.2 Screen Format Options 5–16
5.16.3 The User's Response to the SCREEN Option 5–20

5.17 MENU OPTION 5–21
5.17.1 Items 5–21
5.17.2 Descriptions 5–22
5.17.3 Data Value 5–22
5.17.4 Sub-menu Indicator 5–22
5.17.5 Menu DIRECTIVES: 5–22

5.18 KEY INPUT 5–29

CHAPTER 6 FILE HANDLING 6–1

6.1 OPEN #CHNL_NUM: NAME ... 6–1
6.1.1 OPEN Options 6–2

6.2 CLOSE #CHNL_NUM 6–4

6.3 KILL 6–4

6.4 INPUT #CHNL_NUM: VAR, VAR... 6–5

6.5 LINE INPUT #CHNL_NUM: STR_VAR, STR_VAR... 6–6

6.6 ASK #CHNL_NUM: 6–8
6.6.1 ASK Options 6–9

6.7 SET #CHNL_NUM: EXPR 6–11

Contents

CHAPTER 7 PRINTING AND DISPLAYING DATA 7–1

7.1 PRINT 7–1
7.1.1 Printing Multiple Expressions 7–2

7.2 PRINT #CHNL_NUM 7–14

7.3 USING THE INTOUCH SCREEN 7–15

7.4 MESSAGE 7–15

7.5 CLEAR 7–17

7.6 OPTION COMMUNICATE ON | OFF 7–20

CHAPTER 8 DEFINING ARRAYS 8–1

8.1 DIM AND REDIM 8–1
8.1.1 DIM 8–1
8.1.2 REDIM 8–3

8.2 OPTION BASE 8–5

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 9–3

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 9–5

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 SET PORT ON | OFF 9–9
9.10.1 SET PORT ON 9–9
9.10.2 SET PORT OFF 9–11

9.11 ASK RESPONSES 9–11

9.12 SET SCROLL 9–11

9.13 ASK | SET SEED 9–12

9.14 ASK | SET SYSTEM 9–13
9.14.1 ASK SYSTEM: COMMENT 9–13
9.14.2 SET SYSTEM: COMMENT 9–14
9.14.3 ASK SYSTEM: DIRECTORY 9–15
9.14.4 SET SYSTEM: DIRECTORY 9–15

9.14.5 ASK SYSTEM, LOGICAL: VALUE 9–16
9.14.6 SET SYSTEM, LOGICAL: VALUE 9–16
9.14.7 ASK SYSTEM: MODE 9–17
9.14.8 ASK SYSTEM: PARAMETER 9–17
9.14.9 ASK SYSTEM, PASS: SUCCESS 9–18
9.14.10 ASK SYSTEM: PROCESS 9–18
9.14.11 SET SYSTEM: PROCESS 9–19
9.14.12 ASK SYSTEM: RIGHTS 9–19
9.14.13 ASK SYSTEM, SYMBOL: VALUE 9–20
9.14.14 SET SYSTEM, SYMBOL: VALUE 9–20
9.14.15 ASK SYSTEM: USER 9–21

9.15 ASK | SET WINDOW 9–21
9.15.1 ASK WINDOW AREA 9–21
9.15.2 SET WINDOW AREA 9–22
9.15.3 ASK WINDOW: COLUMN 9–22
9.15.4 SET WINDOW: COLUMN 9–23
9.15.5 ASK WINDOW: CURRENT 9–23
9.15.6 SET WINDOW: CURRENT 9–24
9.15.7 ASK WINDOW: DATA 9–25
9.15.8 SET WINDOW: DATA 9–25
9.15.9 ASK WINDOW: KEYMAP 9–26
9.15.10 SET WINDOW: KEYMAP 9–27
9.15.11 SET WINDOW KEYSTROKE: VALUE 9–28
9.15.12 ASK WINDOW: ROW 9–29
9.15.13 SET WINDOW: ROW 9–30
9.15.14 ASK WINDOW: TYPEAHEAD 9–30
9.15.15 SET WINDOW: TYPEAHEAD 9–31

9.16 ASK | SET ZONEWIDTH 9–32
9.16.1 ASK ZONEWIDTH 9–32
9.16.2 SET ZONEWIDTH 9–32

CHAPTER 10 LOOPS, CONDITIONALS, CHAINING AND BRANCHING 10–1

10.1 FOR/NEXT LOOP 10–1
10.1.1 The STEP Option 10–3
10.1.2 EXIT FOR 10–4
10.1.3 REPEAT FOR 10–5
10.1.4 ITERATE FOR 10–6

10.2 DO 10–8
10.2.1 WHILE Option 10–9
10.2.2 UNTIL Option 10–9
10.2.3 EXIT DO 10–11
10.2.4 REPEAT DO 10–11
10.2.5 ITERATE DO 10–12

10.3 EXECUTE 10–13

10.4 CONDITIONALS 10–14

10.5 IF/THEN 10–15
10.5.1 END IF 10–15
10.5.2 ELSE Option 10–16
10.5.3 ELSEIF Option 10–18

10.6 SELECT CASE/END SELECT 10–19
10.6.1 CASE ELSE Option 10–21
10.6.2 CASE IS Option 10–22

10.7 CHAINING PROGRAMS 10–23
10.7.1 CHAIN 10–23

10.8 PASS COMMANDS TO THE OPERATING SYSTEM 10–24
10.8.1 PASS [NOWAIT: | NORETURN:] 10–24

10.9 BRANCHING 10–26

10.10 GOTO 10–26
10.10.1 ON..GOTO [ELSE] 10–27

10.11 GOSUB/RETURN 10–29
10.11.1 GOSUB 10–29
10.11.2 ON...GOSUB 10–30

10.12 DISPATCH 10–32

10.13 ROUTINE/END ROUTINE 10–33

10.13.1 EXIT ROUTINE 10–33
10.13.2 REPEAT ROUTINE 10–34
10.13.3 Branching and Code Blocks 10–34

CHAPTER 11 EXCEPTION HANDLING 11–1

11.1 CAUSE EXCEPTION 11–1

11.2 WHEN EXCEPTION IN 11–2
11.2.1 WHEN EXCEPTION USE 11–3
11.2.2 RETRY 11–5
11.2.3 CONTINUE 11–6
11.2.4 RESUME 11–7
11.2.5 EXIT HANDLER 11–8

CHAPTER 12 CALLING ROUTINES WRITTEN IN OTHER LANGUAGES 12–1

12.1 LIBRARY 12–1

12.2 CALL 12–3

CHAPTER 13 DATA STRUCTURE STATEMENTS 13–1

13.1 GENERAL INFORMATION 13–1

13.2 DATA STRUCTURE STATEMENTS 13–2

13.3 OPEN STRUCTURE 13–2
13.3.1 ACCESS Option 13–4
13.3.2 LOCK Option 13–4
13.3.3 DATAFILE Option 13–4

13.4 CLOSE STRUCTURE STRUC_NAME 13–4

13.5	EXTRACT STRUCTURE	13–5
13.5.1	EXTRACT STRUCTURE: KEY	13–7
13.5.2	EXTRACT STRUCTURE, FIELD: PARTIAL KEY	13–8
13.5.3	CANCEL EXTRACT	13–9
13.5.4	EXIT EXTRACT	13–10
13.5.5	REEXTRACT STRUCTURE	13–10
13.5.6	EXTRACT STRUCTURE: APPEND	13–11
13.5.7	EXTRACT STRUCTURE, SET, USING	13–12
13.5.8	EXTRACT STRUCTURE, SET, FIELD: KEY	13–13
13.6	INCLUDE STATEMENT	13–13
13.7	EXCLUDE STATEMENT	13–14
13.8	SORT STATEMENT	13–15
13.9	FOR EACH/NEXT	13–16
13.10	SET STRUCTURE	13–17
13.11	SET STRUCTURE: CURRENT	13–18
13.12	SET STRUCTURE, FIELD: KEY	13–19
13.13	SET STRUCTURE, FIELD: PARTIAL KEY	13–19
13.14	SET STRUCTURE: ID	13–20
13.15	SET STRUCTURE: POINTER	13–21
13.16	SET STRUCTURE: RECORD	13–21
13.17	SET STRUCTURE : EXTRACTED 0	13–22
13.18	SET STRUCTURE, SET, USING: EXPRESSION 'OWNER'	13–22

13.19	SET STRUCTURE, SET: EXPRESSION 'OWNER'	13–23
13.20	SET STRUCTURE, SET, USING, FIELD: KEY	13–23
13.21	SET STRUCTURE, SET, USING	13–24
13.22	ASK STRUCTURE	13–24
13.23	ASK STRUCTURE FIELD: ITEM	13–25
13.24	ASK STRUCTURE: CURRENT	13–34
13.25	ASK STRUCTURE: FIELDS	13–35
13.26	ASK STRUCTURE: KEYS	13–35
13.27	ASK STRUCTURE: CAPABILITY	13–35
13.28	ASK STRUCTURE: EXTRACTED	13–36
13.29	ASK STRUCTURE: ID	13–36
13.30	ASK STRUCTURE: POINTER	13–37
13.31	ASK STRUCTURE: RECORD	13–38
13.32	ASK STRUCTURE: RECORDSIZE	13–39
13.33	ASK STRUCTURE: ACCESS	13–39
13.34	ASK |SET STRUCTURE: TIMEOUT	13–40
13.35	ASK | SET STRUCTURE #STRING_EXPR . . .	13–40

13.36 ASK STRUCTURE: ENGINE 13–41

13.37 DELETE STRUCTURE 13–41

13.38 ADD STRUCTURE 13–42

13.39 CANCEL ADD 13–43

13.40 EXIT ADD 13–44

13.41 LOCK | UNLOCK STRUCTURE 13–44

CHAPTER 14 WRITING AND DEBUGGING INTOUCH PROGRAMS 14–1

14.1 DEBUG ON/OFF 14–1
14.1.1 DEBUG ON 14–1
14.1.2 DEBUG OFF 14–2

14.2 MONITORING PROGRAM EXECUTION 14–3

14.3 THE TRACE AND STATISTICS FEATURES 14–3
14.3.1 TRACE ON 14–4
14.3.2 TRACE OFF 14–5
14.3.3 STATS ON 14–6
14.3.4 STATS OFF 14–6
14.3.5 LIST STATS 14–7

14.4 BREAK AND STEP 14–8
14.4.1 BREAK 14–8
14.4.2 STEP 14–9

14.5 GO–CONTINUING PROGRAM EXECUTION 14–11
14.5.1 GO 14–11

14.6 COMMENTS 14–13

14.6.1 REM 14–13
14.6.2 ! COMMENT_TEXT 14–14
14.6.3 ~ Used to Border Part of a Program 14–16

14.7 PROGRAM, END, STOP AND HALT STATEMENTS 14–16
14.7.1 PROGRAM 14–17
14.7.2 END 14–17
14.7.3 STOP 14–18
14.7.4 HALT 14–19
14.7.5 RANDOMIZE 14–21

CHAPTER 15 SETUP ROUTINE 15–1

15.1 SETUP–OVERVIEW 15–1
15.1.1 Entering SETUP 15–2

15.2 SETUP PROMPTS 15–3

15.3 DEFINE PROCEDURE 15–5

15.4 SHOW PROCEDURE 15–11

15.5 DEFINITIONS 15–11

15.6 SHOW STRUCTURE 15–12

15.7 CREATE PROCEDURE 15–14

15.8 SECURITY 15–17

CHAPTER 16 ELEMENTS OF AN INTOUCH PROGRAM 16–1

16.1 GENERAL INFORMATION 16–1
16.1.1 Continuing Program Lines 16–4

16.2 LABELS 16–4

16.3 STORING INTOUCH PROGRAMS 16–5

16.4 DATA IN INTOUCH PROGRAMS 16–5
16.4.1 BOOLEAN 16–7
16.4.2 Expressions 16–7
16.4.3 Constants 16–7
16.4.4 Variables 16–9
16.4.5 Structure References 16–12
16.4.6 Multiple Occurrence Fields 16–14
16.4.7 Compound Expressions 16–15
16.4.8 Numeric Expressions 16–15
16.4.9 Conditional Expressions 16–16
16.4.10 Order of Evaluation 16–19

CHAPTER 17 INTOUCH RUNNING UNDER VMS 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

APPENDIX D ASCII CHARACTER SET D–1

Contents

APPENDIX E CONVENTIONS USED IN THIS MANUAL E–1

APPENDIX F MAKING LIBRARIES FOR USE WITH INTOUCH F–1

APPENDIX G INTOUCH DATABASE INTERFACES G–1

G.1 ADABAS G–1
- G.1.1 SETUP G–1
- G.1.2 Limitations G–1
- G.1.3 An example of Setting Up and Using the ADABAS Interface G–2
- G.1.4 Referencing ADABAS Multi-value and Periodic Group Fields G–2
- G.1.5 Sample INTOUCH Program Accessing ADABAS G–2
- G.1.6 NATURAL Interface G–3

G.2 DBMS G–3

G.3 RDB G–4
- G.3.1 SETUP G–4
- G.3.2 Overview G–4
- G.3.3 Limitations G–4
- G.3.4 Performance Topics G–5
- G.3.5 Sample Description and Use of a Database G–5
- G.3.6 Setting Up the Structure G–5
- G.3.7 Sample Rdb Table Definition G–5
- G.3.8 Sample INTOUCH Program Accessing Rdb G–6

G.4 DBASEIII+ G–6
- G.4.1 SETUP G–6
- G.4.2 Overview G–6
- G.4.3 Limitations G–6
- G.4.4 dBASEIII+ datatypes and INTOUCH attributes G–6

G.5 CDD G–7
- G.5.1 SETUP G–7
- G.5.2 Overview G–7

G.5.3 Attributes G–7
G.5.4 Edit Mask Conversions G–8
G.5.5 Datatype Conversions G–8
G.5.6 Limitations G–9
G.5.7 Sample Description and Use of a CDD/Plus Record G–9
G.5.8 Sample CDD/Plus Record Definition G–9
G.5.9 Setting Up the Structure G–10
G.5.10 Sample INTOUCH Program Using the CDD Definitions G–11

INDEX

EXAMPLES

1–1 The INTOUCH Frame 1–1
1–2 Using VMS System Utilities 1–2
1–3 Option Menu for Command Completion 1–3
1–4 Spelling Correction 1–4
1–5 PRINT "HELLO" 1–5
1–6 Sample Program 1–6
1–7 Typing in Information 1–7
1–8 HELP 1–8
1–9 CTRL/C feature 1–10
1–10 Example of Changed Program Code 1–11
1–11 Error Message 1–12
1–12 Corrected Error 1–13
2–1 The RUN Command 2–2
2–2 The NEW Command 2–3
2–3 The SAVE Command 2–5
2–4 The RUN Command 2–6
2–5 The RUN Command 2–6
2–6 The RUN Command 2–7
2–7 The REPLACE Command 2–8
2–8 The OLD Command 2–9
2–9 The LIST Command 2–11
2–10 The LNH Command 2–11
2–11 The LIST Command 2–12
2–12 FRAME OFF 2–12
2–13 The FRAME OFF Command 2–17
2–14 The IDE Command 2–18
2–15 The GO Command 2–22
2–16 The UNSAVE Command 2–23
14–1 TRACE ON 14–5
14–2 LIST STATS 14–8

Contents

14–3	BREAK	14–9
14–4	STEP	14–10
14–5	STEP - With Number	14–11
14–6	GO	14–12
14–7	STOP—Sample Program	14–18
14–8	HALT—Sample Program	14–20
15–1	To Use Setup	15–2
15–2	Setup Prompts	15–2
15–3	DEFINE Procedure	15–5
15–4	Field Contents Prompts	15–6
15–5	SHOW	15–11
15–6	SHOW Definitions	15–11
15–7	Field Information Displayed	15–12
15–8	SHOW STRUCTURE	15–12
15–9	Structure Displayed	15–13
15–10	EXIT	15–13
15–11	CREATE	15–14
15–12	File Organization Prompt	15–15
15–13	Field Contents Prompts	15–15
15–14	Record Size Prompt	15–16
15–15	Proceed Prompt	15–17
16–1	Executing an INTOUCH program	16–2
16–2	Structures	16–12
G–1	Setup	G–10

FIGURES

15–1	Character Positions	15–8

TABLES

5–1	PARAMETERS	5–13
9–1	Concepts—Supported	9–29
13–1	Structure Types	13–36
16–1	E Notation	16–8
A–1	DATE$ Function - Integer Values	A–7
A–2	DAYS Function - Integer Values	A–8
A–3	EDIT$ Function - Operation Values	A–10
A–4	FULLTIME$ Function - Integer Values	A–15
D–1	ASCII Character Set	D–1

Preface

INTOUCH is a next generation database language for VAX 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. The interface includes:

- simple structure statements for complex manipulations data from RMS files.
- 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.

Preface

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 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. It may 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.

1 Getting Started with INTOUCH

1.1 The INTOUCH Environment

You can invoke INTOUCH by typing the word "INTOUCH" at the VMS prompt ($). You will enter the INTOUCH environment. When you enter the INTOUCH environment, you see the INTOUCH frame:

Example 1–1 The INTOUCH Frame

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
-------------------------------------------------------------------




 INTOUCH

-------------------------------------------------------------------
     INTOUCH Copyright (c) 1984, 1994 Touch Technologies, Inc.
 EXIT = Exit                              \ = Back   HELP = Help
```

The frame consists of highlighted bars across the top and bottom of the screen. In the frame, the INTOUCH prompt is the word "INTOUCH". It appears highlighted.

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

You can use VMS system utilities from within the INTOUCH environment. At the INTOUCH prompt, type a dollar sign ($) and the name of the VAX utility you want to use. INTOUCH runs the utility. When the utility ends, INTOUCH returns to the INTOUCH environment.

Example 1–2 Using VMS System Utilities

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status :
------------------------------------------------------------------

 INTOUCH

 $ SHOW USERS

                   VAX/VMS Interactive Users
                    20-NOV-1988  15:13:15.39
              Total number of interactive users = 2

              Username    Process Name    PID      Terminal
              ALLEN       TTA7_ALLEN   00000488   TTA7:
              TOM         TXA4_TOM     00000447   TXA4:

 INTOUCH

------------------------------------------------------------------

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

- The INTOUCH prompt is the word "INTOUCH".
- To use a VMS system utility at the INTOUCH prompt, type $ and the name of the program or DCL command you want to use.
- To use a VMS system utility at the INTOUCH prompt, type $$ and the DCL command you want to use. This allows you to stay at the DCL command level until you press the RETURN key.

1.1.1 Command Recall History

The INTOUCH environment handles command recall. The last 50 previously entered statements can be recalled using the UP/DOWN ARROW keys, as well as the CTRL/B (up) and CTRL/V (down) key combinations and the Prev (up) and Next (down) keys.

An alternate way to perform command recall is by using the Find and PF2 keys. Type the partial command and press the Find or PF2 key.

1.1.2 Using the Tab Key for Indenting

INTOUCH handles the Tab key as follows:

- INTOUCH indents a line according to statement type. Typically, 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.

1.1.3 Command Completion

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

In the example below, INTOUCH completes SO to form the command, SORT:

```
INTOUCH
10 SO Tab

10 SORT
```

When entering an INTOUCH statement, pressing the Tab will give a menu of the available options. If only one option is available, that option will automatically be selected. In the example below, REN is an ambiguous command. When you press the Tab, you are given a menu of choices to complete the command:

Example 1–3 Option Menu for Command Completion

```
INTOUCH 4.1                    Program: NONAME
The Next Generation Language   Status :
--------------------------------------------------------------------------
                                                    +---Choices----+
                                                    |  RENAME      |
                                                    |  RENUM       |
                                                    |  RENUMBER    |
                                                    +--------------+

10 REN

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

Example 1–3 Cont'd. on next page

Example 1–3 (Cont.) Option Menu for Command Completion

1.1.4 Spelling Correction

The Tab key can be used to correct misspelled words in INTOUCH. Just position the cursor on the misspelled word and press Tab. 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 is pressed.

Example:

Example 1–4 Spelling Correction

```
INTOUCH 4.1                    Program: NONAME
The Next Generation Language   Status :




INTOUCH

10 set strctre Tab.
10 set structure
```

1.1.5 Abbreviations

INTOUCH offers abbreviations for a number of statements. To expand the abbreviation, press the Tab key.

```
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
```

1.2 Using the INTOUCH Environment–Some Examples

Once you enter the INTOUCH environment, all of INTOUCH's features are available to you. For example, type PRINT "HELLO" and press the RETURN key. INTOUCH prints HELLO.

Example 1–5 PRINT "HELLO"

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
-------------------------------------------------------------------



 INTOUCH

 PRINT "HELLO"
 HELLO

 INTOUCH


-------------------------------------------------------------------

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

PRINT can be executed from the INTOUCH environment, as above. It can also be a statement in an INTOUCH program. For example, type the following program at the INTOUCH prompt.

Note

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

Note

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

1.3 Trying out INTOUCH programs

In the program below, type each line exactly as shown. At the end of each line, press the RETURN key.

Example 1–6 Sample Program

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
------------------------------------------------------------


  NEW

  INTOUCH

  10 INPUT "Your name, please": NAME$
  20 PRINT "Hello, "; NAME$
     END



------------------------------------------------------------

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

1.4 To RUN an INTOUCH program

This is an INTOUCH program. When you execute (RUN) the program, INTOUCH asks for your name and then says hello to you.

Type RUN to execute the program. At each prompt, e.g., Your name, please?, type in the information:

Example 1–7 Typing in Information

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
------------------------------------------------------------------
 10 INPUT "Your name, please": NAME$
 20 PRINT "Hello, "; NAME$
    END
 RUN
 NONAME        1-FEB-1985 14:30
 Your name, please? John
 Hello, John

 INTOUCH

------------------------------------------------------------------

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

1.5 To get HELP on INTOUCH

Type HELP and press the RETURN key. The screen will clear and you will enter INTOUCH's help facility.

Example 1–8 HELP

```
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,
          or press CTRL/Z

  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?
```

You can now enter one of the topics listed. INTOUCH will provide information on the topic you choose. Let's exit the HELP facility and try INTOUCH's code-change and continue capability. Press the RETURN key to leave the HELP facility. Type RUN to execute the program again.

INTOUCH has an extensive library of HELP text on many topics. Typing HELP by itself causes INTOUCH to display a list of topics and the prompt "Topic?".

You enter a topic from the list. INTOUCH provides information about that topic. If there is more detailed information, INTOUCH displays a list of subtopics available and the prompt "Subtopic?".

You may also request specific information by entering the desired topic and subtopic at the first prompt:

```
HELP topic subtopic subtopic...
```

Press the RETURN key at the "Topic?" prompt to exit the HELP facility. Type a question mark at any HELP prompt to get a list of the available topics and sub-topics to be displayed.

1.5.1 The EXIT Command

At the bottom of the INTOUCH frame, a summary message appears. The commands described in the lower frame may be used at any time while you are in INTOUCH. These commands are described below:

EXIT	You can exit an INTOUCH routine or procedure by typing the word EXIT at a prompt.
CTRL/Z	You can exit an INTOUCH routine or procedure by pressing CTRL + Z. At any prompt in a program, you can exit in either way.

At the INTOUCH prompt, you can type the word EXIT to exit INTOUCH and return to the VMS system prompt (usually "$"). CTRL/Z does not exit the INTOUCH environment.

When you EXIT 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.

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

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

HELP	You can get help by typing HELP. Type HELP at the INTOUCH prompt or in an INTOUCH routine to get helpful information.

1.6 To Interrupt an INTOUCH Program

You can press CTRL + C to interrupt a program while it is running. To use the CTRL/C command, do the following:

- While pressing down the CTRL key, press the letter C.

In the following example, when INTOUCH asks for your name, use the CTRL/C feature:

Example 1–9 CTRL/C feature

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status : BREAK at 10
------------------------------------------------------------------------

 INTOUCH

 RUN
 NONAME         1-FEB-1985 15:10
 Your name, please? CTRL/C
 Cancel
  RETURN

 BREAK at 10

 INTOUCH

------------------------------------------------------------------------

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

INTOUCH interrupts program execution and tells you where it stopped: "BREAK at 10". At this point, we can change the program code. Let's change the program.

Type:

```
10 NAME$="Fred"
```

Now, type GOTO 10 and press the RETURN key. We will resume program execution at line 10. INTOUCH executes the new line, setting NAME$ equal to "Fred". Then INTOUCH says hello to Fred.

Example 1–10 Example of Changed Program Code

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
------------------------------------------------------------------------
 RUN
 NONAME       1-FEB-1985 15:15
 Your name, please?  CTRL/C
 Cancel
  RETURN

 BREAK at 10

 INTOUCH

 10 NAME$='Fred'
 GOTO 10
 Hello, Fred

 INTOUCH

------------------------------------------------------------------------

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

1.7 To Change an INTOUCH program

When you create a program, INTOUCH compiles each statement as it is entered. This means it returns any compile-time errors immediately. Now, let's see what happens if you misstype something.

First list the program and then change line 10 to "10 OOPS".

Example 1–11 Error Message

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
-------------------------------------------------------------

 INTOUCH

 LIST
 NONAME             1-FEB-1985 15:32
 10 NAME$='FRED'
 20 PRINT "Hello, "; NAME$
    END

 10 OOPS

 10 OOPS
 ..1
 Unrecognized statement


-------------------------------------------------------------

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

INTOUCH gives us an error, saying that OOPS is not a statement. At this point, we can correct the error by retyping the statement, we can delete the line, etc.

Let's retype the original line 10:

```
10 INPUT "Your name, please": NAME$
```

Press the RETURN key when you have finished, and run the program.

Example 1–12 Corrected Error

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
------------------------------------------------------------------------


 RUN
 NONAME      1-FEB-1985 15:35
 Your name, please? John
 Hello, John

 INTOUCH

------------------------------------------------------------------------

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

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

2 Summary of INTOUCH Commands

The following chapters explain the statements in the INTOUCH language. The purpose of the summaries is to present the information you need to use each statement.

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

FORMAT:

A description of the statement's format. For example:

```
INPUT [options:] var [, var...]
```

Keywords are words that must be typed as shown to use the statement or its options. Key words are in upper case, e.g., INPUT. The elements in lower case describe information that must be provided to use the statement. For example, "var" stands for variable. Therefore, a variable must be provided with the INPUT statement. You can provide more than one.

Elements in brackets are optional. Separate multiple options with commas.

```
INPUT PROMPT "Full name", TIMEOUT 30 : NAME$
```

INTOUCH is space sensitive, so spaces must be included where they are shown in the format. For a list of the conventions used in the statement formats, see Appendix.

EXAMPLE:

```
10  INPUT "Full name", TIMEOUT 30, ELAPSED X%: NAME$
    NAME$
    PRINT "Elapsed time:"; X%
20  END

RNH
Full name? Suzanne Scott
Suzanne Scott
Elapsed time: 2

INTOUCH
```

The example shows the statement used correctly. Wherever possible, the example is a full program rather than a program fragment.

We encourage you to type in the examples and try them with INTOUCH.

PURPOSE:

A one- or two-line explanation of the statement's purpose and its uses.

DESCRIPTION:

A more detailed description of the statement and how it works.

2.1 About INTOUCH Commands

INTOUCH commands cause INTOUCH to take some action. You can use these commands at the INTOUCH prompt. For instance, in the following example, the command "RUN" executes the program you enter.

Example 2–1 The RUN Command

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
------------------------------------------------------------------
 NEW

 INTOUCH

 10 INPUT "Your name, please": NAME$
 20 PRINT "Hello, "; NAME$
 30 END
 RUN
 NONAME       1-FEB-1985 17:40
 Your name, please? John
 Hello, John

 INTOUCH
------------------------------------------------------------------

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

2.2 NEW

FORMAT:

```
NEW ["workspace_name"]
```

EXAMPLE:

```
NEW "HELLO"
```

PURPOSE:

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

- may be up to, but not longer than, 19 characters
- may consist of letters, numbers, underscores, and hyphens.

DESCRIPTION:

Whenever you begin writing a new program, type NEW followed by the name of the new program.

If you don't 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 to the current workspace. The workspace name appears at the "Program:" heading in the upper right corner of the screen.

Example 2–2 The NEW Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-----------------------------------------------------------------
RUN
NONAME       1-FEB-1985 17:45
Your name, please? John
RNH
Hello, John
INTOUCH

NEW "HELLO"

INTOUCH

-----------------------------------------------------------------

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

When you use the NEW command, current memory is cleared. Therefore, any programs in memory are lost. INTOUCH writes all active output buffers, closes all files and releases all storage.

2.3 SAVE

FORMAT:

```
SAVE ["file_spec"]
```

EXAMPLE:

```
SAVE "HELLO"
```

PURPOSE:

After a program has been written, the SAVE command saves it for later use.

DESCRIPTION:

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. 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 and errors are discussed in Appendix C.)

After you have saved a file once, use the REPLACE command to save new copies.

Example 2–3 The SAVE Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-----------------------------------------------------------------
NEW "HELLO"

INTOUCH

10 INPUT "Your name, please": NAME$
20 PRINT "Hello, "; NAME$
   END
SAVE "HELLO"

INTOUCH

-----------------------------------------------------------------

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

2.4 RUN

FORMAT:

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

EXAMPLE:

```
RUN "HELLO"
     or

RNH 'HELLO'
```

PURPOSE:

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

DESCRIPTION:

The RUN command executes programs. RUN with no file specification runs the current program in memory.

Example 2–4 The RUN Command

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
------------------------------------------------------------------

10 INPUT "Your name, please": NAME$
20 PRINT "Hello, "; NAME$
   END
RUN
NONAME        1-FEB-1985 16:40
Your name, please? John
Hello, John

INTOUCH

------------------------------------------------------------------

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

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 included, INTOUCH searches for the file, loads it, and then runs it. (See Chapter on the rules for file specifications.) If no extension is given, INTOUCH will use the default extension ".INT".

Example 2–5 The RUN Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
------------------------------------------------------------------

RUN "HELLO"
HELLO        1-FEB-1985 16:40
Your name, please? John
Hello, John

INTOUCH

------------------------------------------------------------------

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

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

Example 2–6 The RUN Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-----------------------------------------------------------------

RNH "HELLO"
Your name, please? John
Hello, John

INTOUCH

-----------------------------------------------------------------

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

Program execution begins at the lowest line number. Lines are executed sequentially by line number. When program execution stops, control returns to the INTOUCH environment.

2.5 REPLACE

FORMAT:

```
REPLACE ["file_spec"]
```

EXAMPLE:

```
REPLACE "HELLO"
```

PURPOSE:

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

DESCRIPTION:

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 extension ".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.

Example 2–7 The REPLACE Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-------------------------------------------------------------------
NEW 'HELLO'

INTOUCH

5  DIM NAME$(5)
10 FOR I = 1 TO 5
     INPUT 'Your name, please': NAME$(I)
     PRINT 'Hello, '; NAME$(I)
   NEXT I
20 END
REPLACE

INTOUCH

-------------------------------------------------------------------

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

2.6 ABORT

FORMAT:

```
ABORT
```

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 to pass to the operating system:

```
ABORT int_expr
```

2.7 OLD

FORMAT:

```
OLD ["file_spec"]
```

EXAMPLE:

```
OLD "HELLO"
```

PURPOSE:

The OLD command loads a program you have saved.

DESCRIPTION:

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 included with the OLD command, INTOUCH tries to load the file carrying the current workspace name and an extension of ".INT".

Example 2–8 The OLD Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-------------------------------------------------------------------

OLD 'HELLO'

INTOUCH

LIST
HELLO      1-FEB-1985 18:00
5  DIM NAME$(5)
10 FOR I = 1 TO 5
     INPUT 'Your name, please': NAME$(I)
     PRINT 'Hello, '; NAME$(I)
   NEXT I
20 END

-------------------------------------------------------------------

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

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. 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. The file in current memory does not change.

2.8 LIST

FORMAT:

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

EXAMPLES:

```
LIST 510-570, FRED
       or
LNH 510-570, FRED
```

PURPOSE:

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

DESCRIPTION:

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. (If no program is in current memory, nothing will be listed.)

When the LIST command is executed, INTOUCH displays a header with the program name, the current date and time. INTOUCH then lists all the lines from the program in current memory. It lists the program sorted sequentially by line number.

Example 2–9 The LIST Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status : LIST
-------------------------------------------------------------------
OLD "HELLO"

INTOUCH

LIST
HELLO        1-FEB-1985 16:40
5  DIM NAME$(5)
10 FOR I = 1 TO 5
     INPUT 'Your name, please': NAME$(I)
     PRINT 'Hello, '; NAME$(I)
   NEXT I
20 END
```

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

Example 2–10 The LNH Command

```
LNH
10 INPUT "Your name, please": NAME$
20 PRINT "Hello, "; NAME$
   END
```

A program must be in current memory to use the LIST or LNH command. If no program is in current memory, nothing will be listed. LIST (or LNH) by itself causes all the program lines in current memory to be displayed. Specific sections of a program can be listed by referencing the program's line numbers and labels.

LIST [label | line_num | routine]

causes the line with the number, label, or routine specified to be listed.

LIST [label | line_num | routine] [, label | line_num | routine...]

lists the line with the first number, routine, or label, the line with the second number, routine, or label, and so on. INTOUCH lists the lines in the order specified.

Example 2–11 The LIST Command

```
INTOUCH 4.1                              Program: REPEAT
The Next Generation Language             Status :
-----------------------------------------------------------------
INTOUCH

LIST VERIFY,5
10 VERIFY: FOR I = 1 TO 5
     INPUT "Your name, please": NAME$(I)
     PRINT "Hello, "; NAME$(I)
   NEXT I
5  DIM NAME$(5)

INTOUCH

-----------------------------------------------------------------

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

You can list ranges of lines.

LIST line_first - line_last

lists the line with the first number, and all lines up to and including the second number. The first line must sequentially precede the second line.

Example 2–12 FRAME OFF

```
INTOUCH 4.1                              Program: REPEAT
The Next Generation Language             Status :
-----------------------------------------------------------------

INTOUCH

LIST 5-20
5 DIM NAME$(5)
10 FOR I = 1 TO 5
     INPUT "Your name, please": NAME$(I)
     PRINT "Hello, "; NAME$(I)
20 NEXT I

INTOUCH

-----------------------------------------------------------------

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

Example 2–12 Cont'd. on next page

Example 2–12 (Cont.) FRAME OFF

You can list from a given line to the end of the program.

LIST line_num lists all lines from the specified line number to the end of the program.

Any number of lines may be listed. You can combine the above methods. 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.

2.9 LIST ERROR

FORMAT:

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

EXAMPLE:

```
LIST ERROR
    or
LNH ERROR
```

PURPOSE:

Use LIST ERROR to see a list of the compile-time errors INTOUCH has detected in your program.

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 lists the errors without the program header, date and time.

All available options with the LIST statement are also available.

2.10 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 this editor. When the editor is exited, INTOUCH returns to the INTOUCH prompt.

INTOUCH uses your default system editor. On many systems, this is EDT. To use another editor, you can define a command synonym for the word EDIT, at DCL level. For example, to use VAX TPU, define EDIT like this:

```
EDIT   :==   EDIT/TPU
```

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.

The INTOUCH environment can be integrated into the TPU editor. To activate this feature, define the logical, **INTOUCH_EDITOR** to TRUE or YES.

```
$ define INTOUCH_EDITOR TRUE
```

EDIT [label | line_num | routine]

causes the line with the number or label specified to be placed in the editor.

```
EDIT DO_TOTALS
```

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

places in the editor the line with the first number, routine, or label. Next, the line with the second number, routine, or label, the line with the third number, routine, or label, and so on. INTOUCH will place the lines in the editor in the order specified in the EDIT statement.

```
EDIT 10,DO_TOTALS
```

EDIT line_first - line_last

edits all the lines from the first line number, up to and including the second line number. The first line number must precede the second line number.

```
EDIT 500-1050
```

EDIT line_num-

edits all lines from the specified line number to the end of the program.

Any number of lines may be edited. The above methods can be combined. If a line is specified which does not exist, nothing is edited. INTOUCH places program lines in the editor in the order requested. Therefore, lines can be edited out of sequential order.

2.11 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.

2.12 FRAME ON | OFF

FORMAT:

```
FRAME [ON | OFF]
```

EXAMPLE:

```
FRAME ON
FRAME OFF
```

PURPOSE:

Turns the INTOUCH window frame on or 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 back on.

Example 2–13 The FRAME OFF Command

```
INTOUCH
FRAME OFF
INTOUCH
```

2.13 REFRESH

FORMAT:

```
REFRESH
```

EXAMPLE:

```
REFRESH
```

PURPOSE:

Use REFRESH to repaint your screen in case it gets garbled.

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.

2.14 IDE

FORMAT:

```
IDE
```

EXAMPLE:

```
IDE
```

PURPOSE:

IDE prints the INTOUCH version number. Use the version number when contacting Touch Technologies, Inc. with suggestions or problems.

DESCRIPTION:

IDE causes INTOUCH to display its copyright and version number.

Example 2–14 The IDE Command

```
 INTOUCH 4.1                              Program: REPEAT
 The Next Generation Language             Status :
-------------------------------------------------------------

 INTOUCH

 IDE

                       Version 4.1
   INTOUCH Copyright (C) 1991,1992, 93 by Touch Technologies, Inc.

        USA:  1 (800) 525-2527    CA:  1 (800) 325-2527

 INTOUCH

-------------------------------------------------------------

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

2.15 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". 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 VAX prompt. For example:

```
$ INTOUCH HELLO
```

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

When the program ends, control returns to the system prompt.

COMPILED IMAGES

INTOUCH procedures can also be compiled into fast-starting memory images. These images are activated within two seconds, regardless of the size of the INTOUCH procedure. The drawback of using compiled images is that they are very large. The minimum size is about 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 MYPROGRAM
```

Compiled image files have a file extension of .INT_IMG for VAX systems and .INT_IMG_AXP for AXP (ALPHA) systems.

Example:

```
Compiling:
    $INTOUCH/COMPILE CUSTOMER

will create:
    CUSTOMER.INT_IMG            <--- on VAX systems
or
    CUSTOMER.INT_IMG_AXP        <--- on AXP systems
```

Running Compiled Images

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

```
$ INTOUCH/IMAGE filename
```

Example:

```
$ INTOUCH/IMAGE myprogram
```

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"
```

2.16 GO

FORMAT:

```
GO
```

EXAMPLE:

```
GO
```

PURPOSE:

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

DESCRIPTION:

GO resumes program execution after it has been interrupted. Once execution has stopped, you can enter immediate mode and debugger 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.

Example 2–15 The GO Command

```
INTOUCH 4.1                                Program: NONAME   DEBUG
The Next Generation Language               Status :
-------------------------------------------------------------------
5  DEBUG ON
10 FOR I = 1 TO 6
20   IF  I = 4  THEN BREAK
30 NEXT I
40 END
RNH
Break at 20

INTOUCH

PRINT SQR(I) ! Type an immediate mode command
 2

INTOUCH

25  PRINT I;  ! Add a new line of code
GO
 4 5 6
INTOUCH
-------------------------------------------------------------------

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

2.17 UNSAVE

FORMAT:

```
UNSAVE ["file_spec"]
```

EXAMPLE:

```
UNSAVE "HELLO"
```

PURPOSE:

To delete programs you no longer want.

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.

Example 2–16 The UNSAVE Command

```
INTOUCH 4.1                              Program: HELLO
The Next Generation Language             Status :
-----------------------------------------------------------------
5  DIM NAME$(5)
10 FOR I = 1 TO 5
     INPUT 'Your name, please': NAME$(I)
     PRINT 'Hello, '; NAME$(I)
   NEXT I
20 END

INTOUCH

UNSAVE "HELLO"

INTOUCH

-----------------------------------------------------------------

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

This command line qualifier causes source code to be replaced automatically when INTOUCH exits.

```
$INTOUCH/DEVELOP   MyProgram.INT
```

2.18 %INCLUDE

FORMAT:

```
%INCLUDE "File_spec"
```

EXAMPLE:

```
10  %INCLUDE "TTI_RUN:EXAMPLE"
```

PURPOSE:

Allows you to put common subroutines into a separate file to be shared among applications.

DESCRIPTION:

%INCLUDE includes source code from another file into the current INTOUCH program. The included file cannot contain line numbers. The default extension for the file is .INC.

2.19 %INCLUDE CONDITIONAL

FORMAT:

```
%include CONDITIONAL: "include_file"
```

DESCRIPTION:

If the "include_file" does not exist, no error is generated. This allows programs to conditionally include modules.

2.20 %DEBUG

FORMAT:

```
%DEBUG INTOUCH_statement
```

EXAMPLE:

```
10 %DEBUG PRINT "DEBUG Items"
20 END
```

DESCRIPTION:

This directive gives an error only if an image compile (/COMPILE) is being done. Use this directive to make sure that DEBUG code doesn't show up in production systems.

2.21 DELAY

FORMAT:

```
DELAY [num_expr]
```

EXAMPLE:

```
10 PRINT 'Waiting a bit'
20 DELAY 3.5
30 PRINT 'Done'
40 END

RNH
Waiting a bit
Done

INTOUCH
```

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. If the numeric expression is omitted, INTOUCH prints a message and waits until the RETURN key is pressed.

The numeric expression does not have to be a whole number; you can use fractions. For example:

```
DELAY 3.5
```

If you use DELAY without a number, INTOUCH prints 'Press RETURN to continue' as a message at the bottom of the screen.

As does INPUT, DELAY returns the following if num_expr is omitted:

EXIT or CTRL/Z

You can type the word EXIT or press CTRL/Z to exit from a procedure or a series of questions. If you enter EXIT or CTRL/Z, INTOUCH sets the internal variable _EXIT to true (1).

\ or Up-arrow

You can press the backslash key "\" or the Up-arrow key to back up from one question to the previous one. If you enter one of these commands, INTOUCH sets the internal variable _BACK to true (1).

HELP

You can get help for an input by typing the word HELP or by pressing the Help key or the PF2 key. If you enter "HELP", INTOUCH sets the internal variable _HELP to true (1).

All input statements set the internal variables _BACK, _EXIT, _HELP.

3 Directives

3.1 %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 displays an error message (quoted_text) on the message line.

3.2 %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.

3.3 %INCLUDE

FORMAT:

```
%INCLUDE "File_spec"
```

EXAMPLE:

```
10  %INCLUDE "TTI_RUN:EXAMPLE"
```

PURPOSE:

Allows you to put common subroutines into a separate file to be shared among applications.

DESCRIPTION:

%INCLUDE includes source code from another file into the current INTOUCH program. The included file cannot contain line numbers. The default extension for the file is .INC.

4 Assigning Variables

Variables specify storage locations. Numeric and string variables are assigned values with the LET statement. String variables may 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.

4.1 DECLARE

FORMAT:

```
DECLARE [STRING | INTEGER | REAL | DYNAMIC] var, var...
```

EXAMPLE:

```
5  DECLARE STRING NAME, SEX
   DECLARE INTEGER AGE
   DECLARE REAL AMOUNT
10 INPUT 'Enter your name': NAME
   INPUT 'Enter your age': AGE
   INPUT 'Enter your sex': SEX
   INPUT 'Enter an amount': AMOUNT
20 PRINT NAME; ' is a'; AGE; 'year old '; SEX; '.'
   PRINT NAME; ' entered the amount'; AMOUNT; '.'
30 END
```

EXAMPLE:

```
10 DECLARE DYNAMIC MY_VAR
   MY_VAR = 4 + 5
   PRINT MY_VAR
   MY_VAR = 'Bob'
   PRINT MY_VAR
20 END

RNH
 9
Bob
```

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 don't 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 may be used in each DECLARE statement. Any number of variables may 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.

4.1.1 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.

4.1.2 DECLARE PREFIX

FORMAT:

```
DECLARE PREFIX str_expr
```

EXAMPLES

```
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:

This statement causes all non-prefixed variable references, structure name references, and field name references to inherit a prefix of 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 xxxx
END ROUTINE
DECLARE PREFIX NONE
```

In addition, at run time, prefixing is turned off before and after a string is EXECUTEd. This allows you to locally prefix executed code:

EXAMPLE:

```
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.

EXAMPLE:

```
 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
```

4.2 OPTION

4.2.1 OPTION REQUIRE DECLARE

FORMAT:

```
OPTION REQUIRE DECLARE
```

EXAMPLE:

```
OPTION REQUIRE DECLARE
20 DECLARE STRING NAME, COMMENT
30 INPUT 'Your name, please': NAME
40 LINE INPUT 'A comment': COMMENT
50 PRINT NAME;' says ';COMMENT
60 END

RNH
Your name, please? Suzanne
A comment? "It's a beautiful day!"
Suzanne says "It's a beautiful day!"
```

PURPOSE:

To ensure that all variables in your program are declared with DECLARE statements.

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.

4.2.2 OPTION BASE

FORMAT:

```
OPTION BASE  [0 | 1]
```

EXAMPLE:

```
10 OPTION BASE 1
   DIM NAMES$(10)
   FOR X% = 1 to 10
     INPUT 'Name': NAMES$(X%)
   NEXT X%
20 END

RUN

Name? Jim
Name?
Name?
Name?
Name?
Name?
Name?
Name?
Name?
Name?

INTOUCH
```

PURPOSE:

Set the default low bound for arrays.

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).

4.3 [LET]

FORMAT:

```
[LET] var = expr
```

EXAMPLE:

```
10 INPUT 'Last name': LAST$
   INPUT 'First name': FIRST$
   LET NAME$ = FIRST$ & ' ' & LAST$
   PRINT NAME$
20 END
RUN
Last name? Taylor
First name? Rick
Rick Taylor

INTOUCH
```

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 "Data in INTOUCH Programs"). The variable's and the expression's data types must match. For instance, if var is a string variable, expr must be a string expression.

The keyword LET is optional:

```
10 INPUT 'Last name': LAST$
   INPUT 'First name': FIRST$
   NAME$ = FIRST$ & ' ' & LAST$
   PRINT NAME$
20 END
```

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

INTOUCH
```

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.

4.4 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 may only be used to assign string values.

LSET, RSET and CSET justify the new value within the length of the previous value. For instance, in the following program, HEADING$ has a length of twenty characters:

```
10 HEADING$ = REPEAT$('.',20)  ! Twenty dots
20 PRINT '('; HEADING$; ')'
30 END
RNH
(....................)

INTOUCH
```

HEADING$ consists of twenty dots. You can use the RSET statement 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)

INTOUCH
```

4.4.1 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             )

INTOUCH
```

PURPOSE:

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. Then 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 may only be used with strings.

4.4.2 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)

INTOUCH
```

PURPOSE:

To right-justify string data, use the RSET statement.

DESCRIPTION:

When INTOUCH executes the RSET statement, it evaluates the string expression on the right side of the equal sign. Then 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 may only be used with strings.

4.4.3 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       )

INTOUCH
```

PURPOSE:

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 may only be used with strings.

4.5 LSET, RSET, CSET FILL

FORMAT:

```
LSET|RSET|CSET FILL str_expr: str_var = expr
```

EXAMPLE:

```
10  A$ = "....."
20  LSET FILL "1": A$ = "$"
30  PRINT A$
40  END

RNH
$1111
```

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.

4.6 READ/DATA Statements

4.6.1 READ

FORMAT:

```
DATA [num_const | str_const] [,[num_const | str_const]...]
               .
               .
               .
READ [num_var | str_var] [,[num_var | str_var]...]
```

EXAMPLE:

```
10 DIM MONTHS$(12)
20 DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY
30 DATA AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
40 FOR I = 1 TO 12
     READ MONTHS$(I)
     PRINT MONTHS$(I)
   NEXT I
50 END
```

PURPOSE:

Use READ and DATA 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 may 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 may be string, numeric or integer variables. They may 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 comma (,), the data item should be enclosed with single or double quotes. For example:

```
5   DIM SALARY$(2)
10  DATA "$1,000", "$2,000"
20  READ SALARY$(1), SALARY$(2)
30  PRINT SALARY$(1), SALARY$(2)
40  END
RNH
$1,000        $2,000
```

The variable types and data types must match or an exception will result. For instance, 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:

```
 5  DIM MONTHS$(4)
 10 DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE
    DATA JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER
 20 READ MONTHS$(1), MONTHS$(2)
 30 READ MONTHS$(3), MONTHS$(4)
 40 PRINT MONTHS$(1), MONTHS$(2), MONTHS$(3), MONTHS$(4)
 50 END

 RNH
 JANUARY     FEBRUARY      MARCH      APRIL

 INTOUCH
```

When the first READ statement is executed, INTOUCH reads the months JANUARY and FEBRUARY. When the READ statement at line 30 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–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 instance, you cannot not have your DATA statements in the main program unit and your matching READ statements in a subprogram.)

See also the RESTORE statement.

4.7 RESTORE Statement

4.7.1 RESTORE

FORMAT:

```
RESTORE
```

EXAMPLE:

```
10 DIM MONTHS$(12), MORE_MONTHS$(12)
20 DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY
30 DATA AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
40 FOR I = 1 TO 12
     READ MONTHS$(I)
     PRINT MONTHS$(I)
   NEXT I
45 RESTORE
50 FOR I = 1 TO 12
     READ MORE_MONTHS$(I)
     PRINT MORE_MONTHS$(I)
   NEXT I
60 END

RNH

JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER
JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER

INTOUCH
```

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.

For instance, in the example program, the twelve months will be read and assigned to the array MONTHS$. When the RESTORE is executed, the DATA statements will be restored. When the READ statement at line 50 is executed, the months will be read into the new array MORE_MONTHS$.

5 Inputting Data

5.1 INPUT

The INPUT statement reads data typed by the user at the terminal and assigns it to variables. Var is a variable 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 the chapter, "File Handling."

The user types data at the terminal in response to the INPUT statement. This response is assigned to the variable(s) listed. The user's 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 ("").

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. . .]
```

PURPOSE:

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

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:

- simple input from the terminal

  Example:

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
------------------------------------------------------------------
 10 INPUT "Your name, please": NAME$
 20 PRINT "Hello, "; NAME$
    END
 RUN
 NONAME        1-FEB-1985 14:30
 Your name, please? John
 Hello, John

 INTOUCH

------------------------------------------------------------------

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

- formatted data entry screens (Please refer to Screen Option.)

  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.: ___-__-____
```

- free format multiline text input (Please refer to Area Option.)

Example:

```
 10  CLEAR
 15  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$
 90  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.

INTOUCH
```

- pop-up menus (Please refer to MENU Option, below.)

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$
15   LINE INPUT MENU menu$: ans$
20   END

     RNH

     +---STRUCTURE---+
     |  TTI_CLIENT  +---ADDRESS--+
     |  ADDRESS     |------------|
     |  MISC...     |  STREET    |
     +--------------|  CITY      |
                    |  STATE     |
                    |  COUNTRY   |
                   +------------+
```

5.2 INPUT Options

The INPUT statement has the following options. (Please refer to individual options, described in detail later in 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 multiline 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.
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, there are several commands and keystrokes that the user can enter instead of an answer for an input.

EXIT or CTRL/Z

You can type the word EXIT or press CTRL/Z to exit from a procedure or a series of questions. If you enter EXIT or CTRL/Z, INTOUCH sets the internal variable _EXIT to true (1).

If you do an input and you type \ or press the Up-arrow key, the internal variable _BACK will be set to true (1). You can then examine the internal variable _BACK within your program and take appropriate action.

HELP

You can get help for an input by typing the word HELP or by pressing the Help key. INTOUCH then sets the internal variable _HELP to true (1).

All input statements set the internal variables _BACK, _EXIT, _HELP.

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 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 has 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 any commas.

```
10 INPUT name$
   PRINT name$
20 END

RNH
? JOHN
JOHN

INTOUCH
```

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

```
RNH
? '  John'
  John

INTOUCH
```

5.5 Inputting Multiple Variables

5.5.1 Simple Input Statement

One input statement inputs several variables. Separate the variables with commas.

```
10 INPUT name$, age%, sex$
20 PRINT name$, age%, sex$
30 END

RNH
? Ted
? Ted Turner, 68, Male
Ted Turner           68                Male

INTOUCH
```

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 more data than there are variables, INTOUCH generates an exception. Users can enter the data for a variable list in one of two ways:

If an exception occurs, INTOUCH restarts from the beginning.

One: they 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
? John,
? 20,
? Male
John                20                Male

INTOUCH
```

Two: users can enter the data on one line by separating the data with commas.

```
RNH
? John, 20, Male
John                20                Male

INTOUCH
```

Whichever method is used, INTOUCH will continue inputting 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 commas.

```
10 LINE INPUT comment_1$, comment_2$
   PRINT comment_1$
   PRINT comment_2$
20 END
```

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

```
RNH
? This is the first line.
? Here is the second line.
This is the first line.
Here is the second line.

INTOUCH
```

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 "PROMPT TEXT"

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

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

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

```
RNH
Your name? Fred
Fred

INTOUCH
```

5.7 PROMPT Option

The PROMPT option displays the prompt text without the question mark and space. The format of INPUT with the PROMPT Option is:

FORMAT:

```
PROMPT str_expr
```

where str_expr is a string expression which contains the prompt text. Str_expr can be any string expression. Separate the prompt text from the variable list with a colon (:).

EXAMPLE:

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

With the PROMPT Option, INTOUCH will print the prompt text exactly as specified, without a question mark and space after the text.

```
RNH
Your name: FRED
FRED

INTOUCH
```

5.8 AT Option

The AT Option positions the cursor on the desired row and column.

FORMAT:

```
AT row, col
```

where row is the row to print at. Col is the column to print at. Row and col may be any integer numeric constants.

If the message line is being altered and a message requiring a delay is present, the print will delay.

EXAMPLE:

```
10 INPUT "NAME", AT 3, 10 :name_1$
   INPUT "PHONE", AT 4, 10 :name_2$
40 END

RNH
NAME? Touch Technologies
PHONE? 619-455-7404
```

5.9 ATTRIBUTES Option

The **ATTRIBUTES** option allows input with attributes. Available attributes are as follows: BOLD, BLINK, REVERSE, UNDERLINE. Multiple attributes used in one INPUT statement are separated by commas.

FORMAT:

```
ATTRIBUTES attr_list
```

Example:

```
10  name_attr$ = 'BOLD, UNDERLINE'
    LINE INPUT 'Name', ATTRIBUTES name_attr$: ans$
20  END
```

5.10 LENGTH Option

The LENGTH Option limits the number of characters that a user can type. It causes INTOUCH to follow the prompt with underscore characters.

FORMAT:

```
LENGTH num_expr
```

where num_expr is the number of characters the user can type.

EXAMPLE:

```
10  INPUT 'NAME', LENGTH 20:name$
20  INPUT 'PHONE', LENGTH 15:phone$
30  PRINT name$, phone$
40  END

RNH
NAME? Touch Technologies__
PHONE? 619-455-7404___
Touch Technologies  619-455-7404
```

5.11 DEFAULT Option

DEFAULT lets you provide defaults for INPUT statements. INTOUCH automatically formats the default appropriately. Users can press RETURN to use the default you have provided.

FORMAT:

```
DEFAULT str_expr
```

where str_expr is a string expression to use as the default. When INTOUCH executes an INPUT statement with a default, it prints the default after the prompt.

EXAMPLES:

```
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
```

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;...
```

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

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.)

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  mnu$  = line1$ + line2$ + line3$ + line4$ + line5$ + line6$
    dflt$ = ""
50  DO
      INPUT MENU mnu$, DEFAULT dflt$: ans$
      IF  _EXIT  THEN EXIT DO
      MESSAGE 'Menu path was', _STRING
      dflt$ = _STRING
    LOOP
100 END
```

5.12 ERASE Option

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

5.13 VALID Option

The VALID Option validates user responses according to validation rules.

FORMAT:

```
VALID str_expr
```

where str_expr is the list of validation rules.

Please refer to the VALID( ) function in the appendix of *INTOUCH, A Guide to the Language.*

EXAMPLE:

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

RNH
NAME? Jack_________________
AGE? 39___
Jack                39

INTOUCH
```

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") is generated. The format of INPUT with the TIMEOUT Option is:

FORMAT:

```
TIMEOUT num_expr:
```

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

```
10 INPUT 'Name', TIMEOUT 30: 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. For example:

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

TIMEOUT 4.5 gives the user approximately 4.5 seconds to enter a name.

5.15 AREA Option

FORMAT:

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

Table 5–1 PARAMETERS

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
 15  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$
 90  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.

INTOUCH
```

DESCRIPTION:

The AREA Option does input from an area on the screen. During a line area input, 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/T	Moves cursor to the top of the area

GOLD/B	Moves cursor to the bottom 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

FORMAT:

```
SCREEN str_expr
```

where str_expr specifies the screen format.

The SCREEN Option is used to create formatted data entry screens. The screen Option lets the user enter data into a fill-in area. For instance, look at the following program:

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

INTOUCH
```

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 may be used within the format to control how the fill-in area works. These fall into two categories: format Options and 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 instance, the # might be used to get the user's 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. For example:

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

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

This program inputs a license number using the format '@@@ @@@@'. You can enter any seven characters.

.

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.For example:

```
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. For example:

```
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 ~ 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 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 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

Letters typed are not echoed (printed) on the screen.

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

RNH
Password? _____
Elene
```

DIGITS

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

```
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 one has been completely 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
```

In the above program, when the user has completely filled in the phone field, INTOUCH automatically jumps to the social security field without the user hitting return. But you must still hit return after entering the SSN.

REQ

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

AT

The AT Option displays the field on the screen at the position specified. The format of the AT Option is:

```
AT row, column
```

"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.

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.
```

To return to INTOUCH, press the <RETURN> key twice.

VALID

The VALID option allows full validation of each field on the screen. (See the VALID function for details.) 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-colon.

Example:

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

RNH

ID = 619-34-54
```

ELAPSED

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

FORMAT:

```
ELAPSED num_var
```

where 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).

Example:

```
10 INPUT 'Name', ELAPSED X: name$
20 PRINT name$; ', your elapsed time is'; X
40 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, 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.

Example:

```
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 keypad 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.

Example program:

```
10 A$ = "<AT 6,10   :Full name @@@@@@@@@@@@@@@@@@@@@@@@>" + &
        "<AT 8,10   :Address1@@@@@@@@@@@@@@@@@@@@@@@@@@@>" + &
        "<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 Jim Hardy_____________________

Address1  9990 Mesa Rim Rd._____________

City      San Diego_____    State CA
```

5.17 MENU Option

DESCRIPTION:

Use the MENU Option to create pop-up menus.

FORMAT:

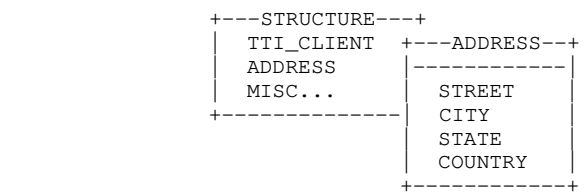
```
[LINE] INPUT MENU str_expr: str_var
```

where str_expr is a string expression that describes the menu. It consists of multiple, comma-separated elements.

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$
 15  LINE INPUT MENU menu$: ans$
 20  END

     RNH
```



5.17.1 Items

Consist of either a description or a description and = sign followed by either a data value or a sub-menu 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.

5.17.2 Descriptions

Descriptions are displayed on the menu for the user to choose.

5.17.3 Data Value

Value returned to the application when this item is chosen.

5.17.4 Sub-menu Indicator

Consists of a "{" followed by menu elements followed by "}".

5.17.5 Menu DIRECTIVES:

Menu DIRECTIVES:)

%TITLE "....."

Assign a title to this menu.

%AT row, column

Menu or sub-menu box starts at given row, column. CENTER can be used in place of row and/or column to center a menu or sub-menu.

Example:

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

%REPLACE

Can be located in any sub-menu. The calling menu is not kept on the screen. Pressing '\' will return to the calling menu.

%MULTI

Allows multiple items to be selected.

%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         |
+---------------+
```

%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 '%MULTI' is used and 'SEL' key is pressed, "Str_var" contains the names of the items. Each item is separated by a linefeed character.

%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
```

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

%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       |
+--------------------+
```

%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
```

```
+------------------+
|  OPEN            |
|  SHOW            |
|  PRINT           |
|                  |
|  GUIDE_OPTIONS   |
|  NOSYSTEM        |
|  MENU ON|OFF     |
+------------------+
```

%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
```

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

%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
```

```
+----------+-------------------------+
|  ENGINES |  DBMS       |   RDB     |
+----------|  RMS        |   DBASE   |
           |  ADABASE    |           |
           +-------------------------+
```

%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     |
|  CHAPTER +-------PAGE---------+
|  PAGE    |  1201   |   1305   |
+----------|  1202   |   1306   |
           |  1203   |   1307   |
           |  1204   |          |
           +---------------------+
```

%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   |   ...  |
+---------------+
```

%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   |
           +------------+
```

%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   |
            +------------+
```

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

%AUTODISPLAY

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

FORMAT:

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

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
```

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. For instance, the EXAMPLE program runs as follows:

```
RNH
See how quick you are!
Press a key, quick!
You took .15 seconds to press j.
```

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 characters in the format "CTRL/xxx". For example, if the user presses CTRL/G, _TERMINATOR returns "CTRL/G".

EXAMPLE:

```
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 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 the Command Language and DCL User's Guide of the OpenVMS documentation set.

The following pages describe the INTOUCH statements used to manipulate files.

6.1 OPEN #chnl_num: NAME ...

FORMAT:

```
OPEN #chnl_num: NAME "file_spec"
     [, ACCESS INPUT| OUTPUT | OUTIN ] [, UNFORMATTED]
     [,UNIQUE] [, OPTIMIZE OFF]
```

EXAMPLE:

```
10 OPEN #1: NAME 'LEDGER.LIS', ACCESS OUTPUT
20 PRINT #1: 'This is the first line of text.'
30 PRINT #1: 'This is the second line of text.'
40 CLOSE #1
RNH
1  This is the first line of text
2  Here 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.

ACCESS INPUT	Open the file for data input.
ACCESS OUTPUT	Create a new file to write to.
ACCESS OUTIN	Open a file for both input and output. Create a new file if it does not already exist.

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.

NAME gives the file specification of the file being opened. The file specification can be any string expression.

6.1.1 OPEN Options

The OPEN statement has several options. Separate multiple options with commas.

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.

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, then 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 with the specifications given.

ACCESS OUTIN

- The file is opened for input and output. Data can be read from, 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 with the specifications given.

UNFORMATTED

- When you write to a file opened UNFORMATTED, the writes are done without any carriage control. This allows various character sequences to be sent to the channel without having CR/LF sequences sent as well.

Example:

```
10  OPEN #1: NAME 'TT:', ACCESS OUTPUT, UNFORMATTED
20  FOR I = 1 TO 10
      PRINT #1: I;
    NEXT I
90  END

RNH
1  2  3  4  5  6  7  8  9  10
INTOUCH
```

OPTIMIZE OFF

- A file is opened without any INTOUCH I/O optimizations.

Example:

```
10  OPEN #2: NAME 'report.txt', OPTIMIZE OFF
20  END
```

UNIQUE

- The file is created with a unique name. These are temporary holding files for listings, etc.

Example:

```
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 str_expr.

Example:

```
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
```

6.2 CLOSE #chnl_num

FORMAT:

```
CLOSE [#chnl_num | ALL]
```

EXAMPLE:

```
10 OPEN #1: NAME 'LEDGER.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 one 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.

6.3 KILL

FORMAT:

```
KILL str_expr
```

EXAMPLE:

```
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 the string expression. You must include the full filename with extension. If you do not include a version number, KILL will delete the most recent version only.

6.4 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

RUN
       1  2  3

INTOUCH
```

DESCRIPTION:

The INPUT 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.

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. For instance:

```
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.

6.5 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 'LEDGER.LIS', ACCESS INPUT
20 LINE INPUT #1: LINE_1$, LINE_2$
   PRINT LINE_1$, LINE_2$
30 CLOSE #1
40 END
```

DESCRIPTION:

LINE INPUT 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 may 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.

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
```

Example:

```
10  ven_ch = _channel
    OPEN #ven_ch: NAME 'tti_run:vendor.int'
    DO
      LINE INPUT #ven_ch, EOF endfile?: datafile$
      IF endfile? THEN EXIT DO
      PRINT 'line was: '; datafile$
    LOOP
    CLOSE #1
20  END
```

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. For instance, say we have a file containing three lines of text (see the PRINT # statement):

```
10 OPEN #1: NAME 'LEDGER.LIS', ACCESS INPUT
20 LINE INPUT #1: line_1$, line_2$, line_3$
   PRINT '1  '; line_1$
   PRINT '2  '; line_2$
   PRINT '3  '; line_3$
30 CLOSE #1
40 END
```

```
RNH
1  This is the first line of text.
2  Here is the second line of text.
3  The third line has...tabs.
```

If an attempt is made to input more data than the file contains, INTOUCH generates an exception.

6.6 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
```

PURPOSE:

Use ASK to find what various characteristics of a device are 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.

6.6.1 ASK Options

A ask option must be included with 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.

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
```

MARGIN num_var

ASK MARGIN finds the right margin of the device specified and assigns its value to the numeric variable.

Example:

```
10 OPEN #3: NAME "STORAGE.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.

Example:

```
10   OPEN #1: NAME "TEMP.TMP", ACCESS OUTPUT
     FOR Z = 1 TO 50
        PRINT #1: 'Line '; Z
     NEXT Z
     CLOSE #1
20   OPEN #1: NAME "TEMP.TMP"
30   FOR I = 1 TO 10
35     LINE INPUT #1: A$
40   NEXT I
50   PRINT 'Tenth item was: '; A$
60   FOR I = 1 TO 10
65     LINE INPUT #1: A$
70   NEXT I
80   PRINT '20th item was: '; A$
90   ASK #1: CURRENT C$
100  FOR I = 1 TO 10
120    LINE INPUT #1: A$
130  NEXT I
140  PRINT '30th item was: '; A$
200  SET #1: CURRENT C$
210  LINE INPUT #1: A$
220  PRINT 'Back to 20th item again: '; A$
230  CLOSE #1
999  END

RNH

Tenth item was: Line  10
20th item was: Line  20
30th item was: Line  30
Back to 20th item again: Line  20

INTOUCH
```

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 a value into the str_var.

Example:

```
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:[ALLEN]MINUTES.LIS
```

6.7 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 SET #3: ZONEWIDTH 5
   PRINT #3: 34, 56, 78
30 CLOSE #3
40 END
```

DESCRIPTION:

SET 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.

SET #chnl_num: Options

A set 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 at which to set the device's print zones. Please refer to 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. Margin must be greater than zonewidth.

Example:

```
10 OPEN #3: NAME "STORAGE.DAT", ACCESS OUTPUT
20 SET #3: MARGIN 45
   PRINT #3: REPEAT$('1234567',10)
30 CLOSE #3
40 END
```

CURRENT str_expr

SET #chnl_num: CURRENT sets the current record to that specified by the str_expr. The str_expr contains the information for the record you want to make current.

Example:

```
10   OPEN #1: NAME "TEMP.TMP", ACCESS OUTPUT
20   FOR Z = 1 TO 50
        PRINT #1: 'Line '; Z
     NEXT Z
     CLOSE #1
     OPEN #1: NAME "TEMP.TMP"
25   FOR I = 1 TO 10
30     LINE INPUT #1: A$
35   NEXT I
40   PRINT 'Tenth item was: '; A$
50   FOR I = 1 TO 10
55     LINE INPUT #1: A$
60   NEXT I
70   PRINT '20th item was: '; A$
100  ASK #1: CURRENT C$
120  FOR I = 1 TO 10
130    LINE INPUT #1: A$
140  NEXT I
150  PRINT '30th item  was: '; A$
200  SET #1: CURRENT C$
210  LINE INPUT #1: A$
220  PRINT 'Back to 20th item again: '; A$
230  CLOSE #1
999  END
```

```
RNH
Tenth item was: LINE  10
20th item was: LINE  20
30th item was: LINE  30
Back to 20th item again: LINE  20

INTOUCH
```

7 Printing and Displaying Data

7.1 PRINT

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

INTOUCH
```

PURPOSE:

Prints text on the screen.

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

INTOUCH
```

7.1.1 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 may 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 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 the 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

INTOUCH
```

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
```

INTOUCH writes data sequentially. If an item is too long to write in one record, INTOUCH continues it in the next record.

```
10 OPEN #1: NAME 'LEDGER.LIS', ACCESS OUTPUT
   PRINT #1: 'This is the first line of text.'
   PRINT #1: 'Here is the second line of text.'
   PRINT #1: 'The third line has...            tabs.'
   CLOSE #1
20 OPEN #1: NAME 'LEDGER.LIS', ACCESS INPUT
   LINE INPUT #1: LINE_1$, LINE_2$
   PRINT '1  '; LINE_1$
   PRINT '2  '; LINE_2$
   CLOSE #1
30 END

RNH
1  This is the first line of text.
2  Here is the second line of text.
```

The TAB Function

The TAB function moves the print position to a given column.

The format of the TAB function is:

```
TAB(column)
```

"Column" is the column to print. TAB positions the cursor at the column specified. TAB always moves forward. If the cursor is at or past the column specified, INTOUCH jumps to the next line. The TAB function is a print item. Therefore, it must be separated from other print items with a comma or a semicolon.

```
10 INPUT name_1$, name_2$
20 PRINT TAB(5);name_1$; TAB(20);'MARY'; TAB(35);name_2$
30 END

RNH
? FRED, JOHN
    FRED           MARY           JOHN
```

AT Option

The AT option moves the cursor to a given place on the screen. The format of the AT option is:

```
AT row, col
```

Row is the row to print at. Col is the column to print at. Row and col may be any integer numeric constants. If the message line is being altered and a message requiring a delay is present, the print will delay.

```
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
```

ERASE Option

The ERASE option erases from 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
```

WIDE Option

This option allows to print double-width characters on the terminal screen.

```
10  PRINT WIDE: 'INTOUCH'
20  END
```

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

INTOUCH
```

- Negative numbers are preceded by a minus sign.

```
10 LET X = -7
   LET Y = -10
   PRINT X;Y
20 END
RNH
-7 -10

INTOUCH
```

- 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

INTOUCH
```

- 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 "Elements of an INTOUCH program.")

```
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.

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
```

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	BLINK causes the expressions in the print list to blink in low and high intensity.
UNDERLINE	UNDERLINE causes the print list to be underlined.
BOLD	BOLD causes the expressions in the print list to appear in bold (high intensity).
REVERSE	REVERSE causes the print list to appear in reverse video.

These options may be combined. For example, the following program would print HELLO boldfaced and in reverse video. It would then blink and underline the word GOODBYE:

```
10 PRINT BOLD, REVERSE: 'Hello'
20 PRINT UNDERLINE, BLINK: 'Goodbye'
30 END
```

The USING Option

Use the USING option to format text. The print_ mask indicates the format for the data in the print list. The format of the USING option is:

```
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. Example:

```
              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

INTOUCH
```

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.

INTOUCH
```

Fields

Fields are made up of format characters. The format characters tell INTOUCH how to print the expressions in the print list.

String Format Characters

#

The # is used to indicate a character position—a place in the format where a character may 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

INTOUCH
```

If the string expression is longer than the field, INTOUCH generates an exception.

Numeric Format Characters

#

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:

```
5  PRINT '1st 2nd 3rd'
10 PRINT USING "### ### ###": 193, 19, 1
20 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.

```
5  PRINT '1st 2nd 3rd'
10 PRINT USING "### ### ###": 193, 19, -1
20 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 zeroes:

```
5  PRINT '--1-- --2--'
10 PRINT USING "##.## ##.##": 1.3, 1.25
20 END

RNH
--1-- --2--
 1.30  1.25
```

<

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'
15 PRINT '123456789'
20 END
RNH
TEST HI
123456789
```

In this 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.

>

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'
15 PRINT '123456789'
20 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.

@

The @ indicates one character position with no translation.

```
5  PRINT USING '####': 0001
10 PRINT USING '@@@@': 0001
20 END
RNH
   1
0001
```

.

You can include a decimal point in a number by putting a period or decimal point in the format. Example:

```
10 PRINT USING "###.##": 19.3
20 END
RNH
 19.30
```

,

You can include commas in your numbers by putting commas in the format:

```
5  A$ = "##,###.##"
10 PRINT USING A$: 28290.06
20 PRINT USING A$: 8290.06
30 PRINT USING A$: 290.06
40 END

RNH
28,290.06
 8,290.06
   290.06
```

Commas may not be used in exponential format.

%

Pad on the left with zeros.

```
10  PRINT '-1- -2- -3-'
    PRINT USING "%%% %%% %%%": 193, 19, 1
20  END

RNH
-1- -2- -3-
193 019 001
```

*****

Pad on the left with asterisks. You can use this symbol for setting up check amounts.

```
10  PRINT USING '***,***.**': 19.42

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
```

+

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
```

-

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
```

$

Prints a floating dollar sign. The dollar sign appears before the number. $ causes INTOUCH to print '$-' before negative numbers and '$' 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
```

$+

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
```

-$

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
```

+$

+$ 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.

$-

Print 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 appear 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.

Directives

- *{UCASE}?*

  The **UCASE** directive converts str_expr to uppercase characters.

  ```
  10 PRINT USING '{UCASE}?' : 'march'
  20 END

  RNH
  MARCH

  INTOUCH
  ```

- *{LCASE}?*

  The **LCASE** directive converts str_expr to lowercase characters.

  ```
  10 PRINT USING '{LCASE}?' : 'MARCH'
  20 END

  RNH
  march
  ```

- *{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
```

7.2 PRINT #chnl_num

FORMAT:

```
PRINT #chnl_num [, USING print_mask]: [TAB(col){, | ;}] [expr {, | ;}
                                      [TAB(col){, | ;}] expr...]
```

EXAMPLE:

```
10 OPEN #1: NAME 'LEDGER.LIS', ACCESS OUTPUT
20 PRINT #1, USING '{UCASE}?': 'first line'
30 PRINT #1: tab (5),'second line'
40 PRINT #1: 'The third line has...         tabs.'
50 CLOSE #1
60 END
```

PURPOSE:

PRINT stores data in a file so you can use it later.

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. 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.

Note

For more information, see chapter entitled "File Handling."

7.3 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 this section can be used within your program to manipulate the screen.

The MESSAGE statement prints a message at line 23 on the screen. When the INTOUCH frame is on, line 23 is within the lower frame. The ERROR option to the MESSAGE statement causes the terminal's bell to ring and clears type-ahead when the statement is executed.

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.

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.

The OPTION COMMUNICATE statement turns the display of messages on or off.

7.4 MESSAGE

FORMAT:

```
MESSAGE [DELAY:] expr [; | ,] [ expr]
```

EXAMPLE:

```
10 INPUT 'Your name, please': name$
   ! Appendix A describes _EXIT and other system functions
   IF _EXIT THEN MESSAGE "Exiting"
20 END
RNH
              Exiting
```

The above program inputs a name. If the command EXIT is entered, INTOUCH prints the message "Exiting" and exits the program.

PURPOSE:

INTOUCH uses the message line (defaults to line 23) to display messages and errors. Use the MESSAGE statement to display your own messages and errors on this line.

DESCRIPTION:

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 print items with a semicolon causes the items immediately follow one another. When the items are printed, no spaces appear between the expressions.

```
MESSAGE "number is",123;456;789
```

The preceding example would cause INTOUCH to print on line 23 the following message;

```
number is 123456789
```

Commas

Separating items with a comma causes the item **following** the comma to be printed after one space.

The ERROR Option

INTOUCH makes sure that at least three seconds elapse before the message is cleaned. When MESSAGE ERROR is executed, INTOUCH rings the device's bell, purges type ahead and displays the specified message. (Type-ahead is the feature that accepts characters typed ahead of the computer's request for input.)

FORMAT:

```
MESSAGE [DELAY:] expr [; | ,] [ expr]
```

EXAMPLE:

```
10 INPUT 'input amount, please': amount$
   IF _EXIT THEN MESSAGE ERROR: "Error", "amount =";0
20 END

RNH
input amount, please? EXIT

INTOUCH

           Error amount =0
```

For instance, in the preceding example when the user entered "EXIT", INTOUCH would ring the terminal's bell, purge type ahead and print the message "Error amount =0" at line 23 on the screen.

The 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]
```

7.5 CLEAR

FORMAT:

```
CLEAR [AREA [BOX [, attr_list:] row1,col1,row2,col2]
```

EXAMPLE:

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

PURPOSE:

Use CLEAR to clear the screen or to clear specific areas of the screen. You can use this statement to clear the screen before executing code or printing information on the screen.

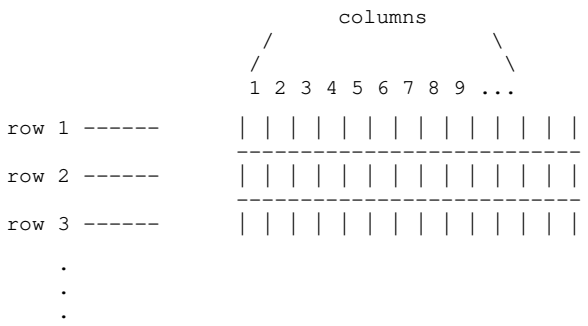
DESCRIPTION:

CLEAR by itself clears all text from the screen. CLEAR does not remove the INTOUCH frame; however, it does remove any message test that is displayed there. 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 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.

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 side. 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:] row1,col1,row2,col2]
```

Example:

```
10  CLEAR AREA REVERSE: 5, 10, 11, 60
20  PRINT AT 7, 20: 'Reverse text'
40  END
```

The **CLEAR AREA** statement allows the following attributes to be used in 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.

FORMAT:

```
CLEAR [AREA [BOX [, attr_list]:] row1,col1,row2,col2]
```

Example:

```
10  CLEAR AREA BOX, BOLD: 5, 10, 11, 60
20  PRINT AT 7, 20: 'INSIDE THE BOX'
30  PRINT AT 12, 1: 'DONE'
40  END
RNH
  +------------------------------------------------+
  |                                                |
  |          INSIDE THE BOX                        |
  |                                                |
  |                                                |
  |                                                |
  +------------------------------------------------+

DONE
```

DESCRIPTION:

The **CLEAR AREA BOX** statement allows the following attributes to be used in creating a box: BOLD, BLINK, REVERSE, UNDERLINE. Multiple attributes in one statement are separated by commas.

7.6 OPTION COMMUNICATE ON | OFF

FORMAT:

```
OPTION COMMUNICATE ON | OFF
```

EXAMPLE:

```
10  OPTION COMMUNICATE OFF
20  PRINT 'HELLO'
25  OPTION COMMUNICATE ON
30  END
```

PURPOSE:

Determines whether to display messages broadcast to the terminal.

DESCRIPTION:

The default is OPTION COMMUNICATE ON. If there is a message, INTOUCH displays it in the message area (bottom of INTOUCH frame). If you enter OPTION COMMUNICATE OFF, INTOUCH will not display the message.

8 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. Arrays are defined in the section on "Data in INTOUCH Programs".

8.1 DIM and REDIM

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.)

General Information

- The highest bound allowed is 2147483648.
- The lowest bound allowed is -2147483647.
- Each array can have up to 32 dimensions.
- See Chapter on "Elements of an INTOUCH Program" for more information on the storage allocation of arrays.
- If an array element outside of the specified bounds is referenced, an exception is generated.

8.1.1 DIM

FORMAT:

```
DIM [INTEGER | REAL | STRING]
          array_name ([int_expr TO] int_expr [, ...])
```

EXAMPLE:

```
5  DIM NAME$(4)
10 FOR I = 1 TO 4
     INPUT NAME$(I)
   NEXT I
15 PRINT
20 FOR I = 1 TO 4
     PRINT NAME$(I)
   NEXT I
30 END

RNH
? John
? Mary
? Rick
? Ed
```

```
John
Mary
Rick
Ed

INTOUCH
```

PURPOSE:

Use DIM to dimension arrays. Arrays are used to store tables of variable information. You must dimension arrays before you use them.

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:

```
5 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:

```
5 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)
```

8.1.2 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$(4)
    INPUT 'How many names': NUM
    REDIM NAME$(NUM)
    NUM = NUM + 1
20 WHEN EXCEPTION USE OUT_OF_BOUNDS
      FOR I = 1 TO NUM
        INPUT NAME$(I)
      NEXT I
   END WHEN
30 PRINT
40 WHEN EXCEPTION USE OUT_OF_BOUNDS
     FOR I = 1 TO NUM
       PRINT NAME$(I)
     NEXT I
   END WHEN
50 PRINT 'Finished'
60 HANDLER OUT_OF_BOUNDS
     MESSAGE ERROR : 'Subscript out of bounds - ' +
     & STR$(I)
     NUM = I - 1
     CONTINUE
   END HANDLER
70 END

RNH

How many names? 3
? JOHN
? MARY
? RICK
? TED          (Error message: "Subscript out of bounds -4")

JOHN
MARY
RICK
FINISHED

INTOUCH
```

PURPOSE:

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. Any values stored in an array are lost when the array is redimensioned. The REDIM statement has the same rules, options and limits as the DIM statement. (See the preceding DIM statement for instructions.)

You can dynamically expand arrays as needed. If you REDIM an array to a larger size, the old values are kept. If your applications depend 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.

8.2 OPTION BASE

FORMAT:

```
OPTION BASE [0 | 1]
```

EXAMPLE:

```
10 OPTION BASE 0
   DIM NAME$(5)
20 FOR I = 0 TO 5
     INPUT NAME$(I)
     PRINT 'Hello, '; NAME$(I)
   NEXT I
30 END
```

```
RNH
```

```
? June
Hello, June
? Joe
Hello, Joe
? Oscar
Hello, Oscar
? Ted
Hello, Ted
? John
Hello, John
? Larry
Hello, Larry
```

```
INTOUCH
```

PURPOSE:

Use OPTION BASE to set the default low bound for arrays to suit your needs.

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.

9 Set and Ask Statements

SET and ASK 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, please refer to the chapter, "File Handling" in this manual.

For information on SET STRUCTURE and ASK STRUCTURE statements, please refer to the chapter, "Data Structure Statements" in this manual.

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? Allen
Who? Bob
Who?

INTOUCH

(User fails to respond within 1 minute.)

RNH
Who? EXIT
Finished

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 then 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? ALLEN_____________________________
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
      SET ERROR OFF
      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 set 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 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
      SET ERROR OFF
      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
```

DESCRIPTION:

ASK ERRORS asks for the number of user errors. The MESSAGE ERROR: statement increments this.

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
```

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.

9.8.2 SET MARGIN

FORMAT:

```
SET MARGIN num_expr
```

EXAMPLE:

```
10 PRINT REPEAT$('.' ,200)
   PRINT
15 ASK MARGIN old_marg
20 INPUT 'What do you want the margin set to': NEW_MARG
30 SET MARGIN new_marg
40 PRINT REPEAT$('.' ,200)
45 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. Margin must be greater than zonewidth.

9.9 ASK | SET MESSAGELINE

9.9.1 ASK MESSAGELINE

FORMAT:

```
ASK MESSAGELINE num_var
```

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:

PURPOSE:

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 SET PORT ON | OFF

9.10.1 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'
90  END
```

PURPOSE:

Allows 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.

9.10.2 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'
90  END
```

PURPOSE:

Disable printing to an attached printer port on a terminal.

DESCRIPTION:

SET PORT OFF turns off the attached port.

9.11 ASK RESPONSES

FORMAT:

```
ASK RESPONSES num_var
```

DESCRIPION:

ASK RESPONSES asks for the number of completed input responses.

9.12 SET SCROLL

FORMAT:

```
SET SCROLL num_expr1, num_expr2
```

EXAMPLES:

```
10  FRAME OFF
    PRINT AT 21, 1: 'This text will not scroll.'
    SET SCROLL 5, 20
    PRINT AT 20, 1:;
    PRINT 'This'
    PRINT 'text'
    PRINT 'will'
    PRINT 'scroll.'
20  SET SCROLL  1,24
30  END
```

```
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.

9.13 ASK | SET SEED

FORMAT:

```
ASK SEED num_var
ASK 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.

9.14 ASK | SET SYSTEM

9.14.1 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:

ASK SYSTEM: COMMENT statement asks the INTOUCH operating system comment for the process.

9.14.2 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:

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.

9.14.3 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.

9.14.4 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:[JIM.INTOUCH]
Directory set to       USER:[JIM]
Directory set back to  USER:[JIM.INTOUCH]
```

DESCRIPTION:

SET SYSTEM: DIRECTORY sets the default device and directory.

9.14.5 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:[SUZANNE]
```

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.

9.14.6 SET SYSTEM, LOGICAL: VALUE

FORMAT:

```
SET SYSTEM, LOGICAL str_expr1: VALUE str_expr2
```

EXAMPLE:

```
10  SET SYSTEM, LOGICAL 'Myname': VALUE 'FRED'
    ASK SYSTEM, LOGICAL 'Myname': VALUE Z$
    PRINT 'Logical set to '; Z$
20  END

RNH
Logical set to FRED
```

DESCRIPTION:

This statement sets the operating system logical name in str_expr1 to the value in str_expr2.

9.14.7 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:

```
INTERACTIVE
BATCH
NETWORK
OTHER
```

9.14.8 ASK SYSTEM: PARAMETER

FORMAT:

```
ASK SYSTEM: PARAMETER str_var
```

EXAMPLE:

```
10  ASK SYSTEM: PARAMETER PDATA$
20  PRINT 'Parameter was: '; PDATA$
30  END
SAVE 'TEST'
EXIT

$ INTOUCH/SOURCE TEST Some data
Parameter was: SOME DATA
$
```

PURPOSE:

ASK SYSTEM: PARAMETER str_var returns any parameter from the command line given after the program name.

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.

9.14.9 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 1 if true or 0 if false in num_var indicating whether or not the last PASS command executed successfully.

9.14.10 ASK SYSTEM: PROCESS

FORMAT:

```
ASK SYSTEM: PROCESS str_var
```

DESCRIPTION:

ASK SYSTEM: PROCESS str_var asks the operating system for the current process name.

9.14.11 SET SYSTEM: PROCESS

FORMAT:

```
SET SYSTEM: PROCESS str_expr
```

DESCRIPTION:

SET SYSTEM: PROCESS str_expr changes the operating system process name to str_expr.

9.14.12 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: TEST
```

DESCRIPTION:

ASK SYSTEM: RIGHTS asks the operating system to return a list of the rights explicitly granted to the calling process.

9.14.13 ASK SYSTEM, SYMBOL: VALUE

FORMAT:

```
ASK SYSTEM, SYMBOL str_expr: VALUE str_var
```

EXAMPLE:

```
10  ASK SYSTEM, SYMBOL 'INTOUCH': VALUE I$
    PRINT 'Value of symbol INTOUCH is: '; I$
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.

9.14.14 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.

9.14.15 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: SUZANNE
```

PURPOSE:

This statement returns the operating system name or ID for the user. Under VMS this is the Username.

9.15 ASK | SET WINDOW

9.15.1 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
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 screen within the area defined by the given upper-left/lower-right coordinates into a string variable. 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.

9.15.2 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.

9.15.3 ASK WINDOW: COLUMN

FORMAT:

```
ASK WINDOW: COLUMN num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
15  ASK WINDOW: COLUMN cur_col
20  PRINT 'Cursor is at column'; cur_col
30  END

RNH
Cursor is at column 10
```

DESCRIPTION:

This statement returns the current column of the cursor's position.

9.15.4 SET WINDOW: COLUMN

FORMAT:

```
SET WINDOW: COLUMN num_expr
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
14  SET WINDOW: COLUMN 4
20  PRINT 'Hi!'
30  END

RNH
   Hi!
```

DESCRIPTION:

This statement positions the cursor at the num_expr column within the current row.

9.15.5 ASK WINDOW: CURRENT

FORMAT:

```
ASK WINDOW: CURRENT str_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,20, UNDERLINE: 'Sample screen'
20  DO
      LINE INPUT 'Name', AT 5,1, LENGTH 30: myname$
      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
90  END DO
95  END
```

DESCRIPTION:

ASK WINDOW and SET WINDOW 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.

9.15.6 SET WINDOW: CURRENT

FORMAT:

```
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: myname$
      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
90  END DO
95  END
```

DESCRIPTION:

ASK WINDOW and SET WINDOW 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.

9.15.7 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.

9.15.8 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.

9.15.9 ASK WINDOW: KEYMAP

FORMAT:

```
ASK WINDOW: KEYMAP str_var
```

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.

9.15.10 SET WINDOW: KEYMAP

FORMAT:

```
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.

9.15.11 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 consist of one or two components. The keystroke value, and the keystroke concept. For instance, 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 concepts are supported:

Table 9–1 Concepts—Supported

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

9.15.12 ASK WINDOW: ROW

FORMAT:

```
ASK WINDOW: ROW num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5,10:;
15  ASK WINDOW: ROW CUR_ROW
20  PRINT 'Cursor is at row'; CUR_ROW
30  END

RNH
Cursor is at row 5
```

DESCRIPTION:

This statement returns the current row of the cursor's position.

9.15.13 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.

9.15.14 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.)

9.15.15 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.

9.16 ASK | SET ZONEWIDTH

9.16.1 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.

9.16.2 SET ZONEWIDTH

FORMAT:

```
SET ZONEWIDTH num_expr
```

EXAMPLE:

```
10 PRINT 1,2,3
20 SET ZONEWIDTH 10
   PRINT 1,2,3
25 SET ZONEWIDTH 20
30 END
RNH
1                   2                   3
1         2         3

INTOUCH
```

DESCRIPTION:

SET ZONEWIDTH sets the print zone width of the device specified to the number designated. Num_expr indicates the width at which to set the device's print zones.

10 Loops, Conditionals, Chaining and Branching

A loop is a place where code is 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.

10.1 FOR/NEXT Loop

FORMAT:

```
FOR index_var = num_expr1 TO num_expr2 [STEP num_expr3]
        ---
        ---  block of code
        ---
NEXT index_var
```

EXAMPLE:

```
5  DIM name$(4)
10 FOR J = 1 TO 4
     INPUT name$(J)
     PRINT name$(J)
   NEXT J
20 PRINT 'Finished'
   PRINT 'Final value:'; J
30 END
```

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
        |
        |
    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 may 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 this example case, INTOUCH counts from 1 to 4, so the loop executes four times.

```
RNH
? Fred
Fred
? John
John
? Mary
Mary
? Kate
Kate
FINISHED
Final value: 5

INTOUCH
```

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$(1), Fred. Then INTOUCH prints NAME$(J).

```
            RNH
J = 1       ? Fred
            Fred
```

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       ? Fred
            Fred
J = 2       ? John
            John
J = 3       ? Mary
            Mary
J = 4       ? Kate
            Kate
            FINISHED
            Final value: 5

            INTOUCH
```

10.1.1 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.

```
5  DIM name$(4)
10 FOR J = 1 TO 4 STEP 3
     INPUT name$(J)
     PRINT name$(J)
   NEXT J
20 PRINT 'Finished'
   PRINT 'Final value:'; J
30 END
```

```
RNH
? Fred
Fred
? John
John
FINISHED
Final value: 7

INTOUCH
```

Nesting Loops

FOR loops can be nested. A nested loop is a loop which begins and ends inside of another loop. **Loops may not overlap**. The inner loop must begin and end completely within the outer loop.

```
start of        5  DIM name$(4)
outer loop ---  10 FOR J = 1 TO 4
                     INPUT name$(J)

                     FOR K = 1 TO J
       inner loop    ---   PRINT name$(K);'  ';
                     NEXT K

  end of             PRINT

  outer loop ---     NEXT J

               20 PRINT 'Finished'
               30 END

               RNH
               ? FRED
               FRED
               ? JOHN
               FRED  JOHN
               ? MARY
               FRED  JOHN  MARY
               ? KATE
               FRED  JOHN  MARY  KATE
               Finished
```

10.1.2 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
```

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. The EXAMPLE program would run like this:

```
RNH
Your name, please? Fred
Hello, Fred
Your name, please? EXIT
Finished

INTOUCH
```

10.1.3 REPEAT FOR

FORMAT:

```
REPEAT FOR
```

EXAMPLE:

```
10 FOR I = 1 TO 10
     PRINT I
     INPUT 'Your name, please': name$
     IF name$ = '' THEN REPEAT FOR
     PRINT 'Hello, '; name$
   NEXT I
20 END
```

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. For instance, the EXAMPLE program runs like this:

EXAMPLE:

```
RNH
 1
Your name, please? Fred
Hello, Fred
 2
Your name, please?
 2
Your name, please?
```

If REPEAT FOR is used within a nested loop, INTOUCH repeats the innermost loop.

EXAMPLE:

```
                    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
```

10.1.4 ITERATE FOR

FORMAT:

```
ITERATE FOR
```

EXAMPLE:

```
 3  FOR I = 1 TO 10
      PRINT I
      INPUT 'Your name, please' : name$
      IF name$ = 'SKIP' THEN ITERATE FOR
    PRINT 'Hello, '; name$
    NEXT I
20 END
```

PURPOSE:

To repeat a FOR loop.

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. For instance, the EXAMPLE program would run like this:

```
RNH
 1
Your name, please? Fred
Hello, Fred
 2
Your name, please? SKIP
 3
Your name, please?
```

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
INTOUCH will           PRINT J
repeat this            INPUT 'Your name, please': name$
inner loop --------    IF name$ = 'SKIP' THEN ITERATE FOR
                       PRINT 'Hello, '; name$
                     NEXT J

                     PRINT 'We now have'; I; 'sets of name(s).'
                   NEXT I
                20 END
```

10.2 DO

FORMAT:

```
DO [WHILE expr | UNTIL expr]
       ---
       ---  block of code
       ---
LOOP [WHILE expr | UNTIL expr]
```

EXAMPLE:

```
5   A = 3
10  DO UNTIL A = 0
      INPUT 'Your name, please': name$
      PRINT 'Hello, '; name$
      A = A - 1
    LOOP
20 END
```

PURPOSE:

Use DO loops 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 the CTRL/C command, see chapter 1.

DO loops can be nested. **Loops may not 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.

10.2.1 WHILE Option

Format: WHILE cond_expr

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.

```
5  A = 3
10 DO
     INPUT 'Your name, please': name$
     PRINT 'Hello, '; name$
     A = A - 1
   LOOP WHILE A>0
20 PRINT 'Finished'
30 END
```

```
RNH
Your name, please? FRED
Hello, FRED
Your name, please? JOHN
Hello, JOHN
Your name, please? KATE
Hello, KATE
Finished
```

10.2.2 UNTIL Option

Format: UNTIL cond_expr

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.

```
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
```

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 instance, when the following program runs, INTOUCH executes the loop until A equals 5 or the user types 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
```

10.2.3 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
```

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. The EXAMPLE program would run like this:

```
RNH
Your name, please? Fred
Hello, Fred
Your name, please? EXIT
FINISHED

INTOUCH
```

10.2.4 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
```

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. For instance, the EXAMPLE program would run like this:

```
RNH
Your name, please? Fred
Hello, Fred
Your name, please?
Your name, please?
```

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
```

10.2.5 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
```

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. For instance, the EXAMPLE program would run like this:

```
RNH
Your name, please? Fred
Hello, Fred
Your name, please? SKIP
Your name, please? EXIT

INTOUCH
```

If ITERATE DO is used in a nested loop, INTOUCH iterates 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
```

10.3 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.

10.4 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 is well-suited to 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.).

10.5 IF/THEN

10.5.1 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
```

10.5.2 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 IF Block Construct

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.

Here is a sample run of the example program:

```
RNH
Enter your age: 19
Enter your sex: M

INTOUCH
```

In the example program, when INTOUCH executes the first IF statement, it evaluates the conditional expression, SEX$[1:1] = 'M'. Since the user is 'Male' the condition is **TRUE**. Therefore, 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.

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 40
   PRINT
30 IF AGE < 40 THEN
     PRINT 'Please go to line A.'
   ELSE
     PRINT 'Please go to line B.'
   END IF
40 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.

10.5.3 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. For example:

```
10 INPUT PROMPT 'Enter your age: ': AGE
   INPUT PROMPT 'Enter your sex: ': SEX$
20 IF UCASE$(SEX$[1:1]) = 'M' THEN GOTO 40 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 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**. Therefore, INTOUCH executes the code following the ELSEIF and prints 'Please go to line B.', then exits the conditional.

10.6 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
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.
```

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.

```
RNH
Your income per year? 0
No income?
Your income per year? CTRL/C>
```

- The main expression and all CASE expressions must be the same data type. (For instance, if the main expression is a string expression, all the CASE expressions must be strings also.) If the data types don't 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.

CASE expr, expr, expr

Each CASE statement can include several expressions separated with 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.

EXAMPLE

```
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

INTOUCH
```

The following example illustrates how to check for a range of values:

EXAMPLE:

```
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
```

10.6.1 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.

EXAMPLE:

```
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
                                    Error message:
                                    Procedure must be add,
                                    del, or exit
Procedure (add, del, exit)? EXIT

INTOUCH
```

10.6.2 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.

Example:

```
10 DO
     INPUT 'Your income per year': income
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? CTRL/C>
```

10.7 Chaining Programs

10.7.1 CHAIN

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
 30 INPUT 'Add to CLIENT structure (Y/N)': reply$
    IF reply$ = 'Y' THEN CHAIN "ADD"
 40 END

 RNH
 Your name (last, first)? Woods, Jack
 Add to CLIENT structure (Y/N)? n

 INTOUCH
```

PURPOSE:

Use CHAIN to exit the current program and run another one.

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.

10.8 Pass Commands to the Operating System

10.8.1 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? memory

The Next Generation Language  Status :

Fixed-Size Pool Areas   Total    Free    In Use     Size
(packets):

Dynamic Memory Usage    Total    Free    In Use     Largest
(bytes):

Paging File Usage Free  Reservable       Total
(pages):

Of the physical pages in use, 19220 pages are permanently allocated
to VMS.

INTOUCH
```

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, on OpenVMS, 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 : 'WAIT 00:01:00'
20  PASS NOWAIT : 'WAIT 00:01:00'
30  PASS NOWAIT : 'SHOW PROCESS/SUB'
40  END

RNH
26-MAR-1990 15:26:18.96                    User: SUZANNE

Processes in this tree:

SUZANNE,1BC
  SUZANNE_1
  SUZANNE_2
  SUZANNE_3
  SUZANNE_4 (*)
```

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: "dir/col=1"
40  END

RNH
B E F O R E

Directory WORK:[DOC.INTOUCH]

RELEASE.NOTES;5
WOMBAT.EXE;1

Total of 2 files.
$
```

10.9 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.

10.10 GOTO

FORMAT:

```
GOTO target
```

EXAMPLE:

```
10 INPUT 'Your Name':name$
   IF _EXIT THEN GOTO 20
   PRINT 'Hello, '; name$
   GOTO 10
20 END
```

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 line number or a label. 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.

```
10 INPUT name$
   IF _EXIT THEN GOTO 20
   PRINT 'Hello, '; name$
   GOTO 10
20 PRINT 'Finished'
30 END

RNH
? Fred
Hello, Fred
? Jake
Hello, Jake
? EXIT
Finished
```

10.10.1 ON..GOTO [ELSE]

FORMAT:

```
ON int_expr GOTO target1 [, target2...] [ELSE statement]
```

where int_expr is the integer expression to check.

where target1, target2 ... is a list of places to jump to. Each target can be a line number or a label.

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
```

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:

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. For instance, the EXAMPLE program would run like this:

```
RNH
Procedure (1=add, 2=del, 3=exit)? 2
Deleting...
Procedure (1=add, 2=del, 3=exit)? 3
```

If the expression doesn't 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 instance, 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 line numbers or labels.

INTOUCH supports 128 targets for ON...GOTO and ON..GOSUB. More targets than 128 gives an "expression too complex" exception.

ELSE Option

The ELSE option executes a statement if the integer expression doesn't yield 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 INPUT 'Procedure (1=add, 2=del, 3=exit)': pro
20 ON pro GOTO ADD, DEL, DONE ELSE PRINT 'Enter 1, 2 or 3'
   GOTO 10
   ADD:
     PRINT 'Adding...'
     GOTO 10
   DEL:
     PRINT 'Deleting...'
     GOTO 10
30 DONE:END
```

```
RNH
Procedure (1=add, 2=del, 3=exit)? -2
Enter 1, 2 or 3
Procedure (1=add, 2=del, 3=exit)? 3
```

10.11 GOSUB/RETURN

10.11.1 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
```

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 line number, a label, or a ROUTINE statement. 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.

The EXAMPLE program would run like this:

```
RNH
Enter your name? John
Hello, JOHN
Finished
```

You can nest 64 GOSUBs up to 64 levels deep.

INTOUCH supports 128 targets for ON...GOTO and ON..GOSUB. More targets than 128 gives an "expression too complex" exception.

10.11.2 ON...GOSUB

FORMAT:

```
ON int_expr GOSUB target1 [, target2...] [ELSE statement]
```

EXAMPLE:

```
10 INPUT 'Procedure (1=add, 2=del, 3=exit)':pro
20 ON pro GOSUB ADD, DEL, DONE ELSE PRINT 'Enter 1, 2 or 3'
   GOTO 10
   ADD:
     PRINT 'Adding...'
     RETURN
   DEL:
     PRINT 'Deleting...'
     RETURN
30 DONE:END
```

```
RNH
Procedure (1=add, 2=del, 3=exit)? add
Procedure (1=add, 2=del, 3=exit)? 1
ADDING...
Procedure (1=add, 2=del, 3=exit)? 2
Deleting...
Procedure (1=add, 2=del, 3=exit)? 3
```

```
INTOUCH
```

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 line numbers, ROUTINE statements, or labels. 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. The EXAMPLE program would run like this:

```
RNH
Procedure (1=add, 2=del, 3=exit)? 1
Adding...
Procedure (1=add, 2=del, 3=exit)? 2
Deleting...
Procedure (1=add, 2=del, 3=exit)? 3

INTOUCH
```

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 instance, 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.

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.

EXAMPLE:

```
10 INPUT 'Procedure (1=add, 2=del, 3=exit)':pro
20 ON pro GOSUB ADD, DEL, DONE ELSE PRINT 'Enter 1, 2 or 3'
   GOTO 10
   ADD:
     PRINT 'Adding...'
     RETURN
   DEL:
     PRINT 'Deleting...'
     RETURN
30 DONE:END
RNH
Procedure (1=add, 2=del, 3=exit)? add
Procedure (1=add, 2=del, 3=exit)? 1
ADDING...
Procedure (1=add, 2=del, 3=exit)? 2
Deleting...
Procedure (1=add, 2=del, 3=exit)? 3

INTOUCH
```

10.12 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
```

PURPOSE:

Executes a routine that the program determines at run-time.

DESCRIPTION:

DISPATCH looks at the contents of a 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.

The program would run like this:

```
RNH
Enter routine name? ADD
Adding information
```

10.13 ROUTINE/END ROUTINE

FORMAT:

```
ROUTINE routine_name
   ---
   --- block of code
   --- [REPEAT ROUTINE]
   --- [EXIT ROUTINE]
END ROUTINE
```

EXAMPLE:

```
10      CLEAR
20      get_username
100     END

12000   ROUTINE get_username
          INPUT PROMPT 'Username: ': uname$
          IF  _BACK or _EXIT  THEN EXIT ROUTINE
          IF  uname$ = '' THEN REPEAT ROUTINE
        END ROUTINE
```

PURPOSE:

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.

10.13.1 EXIT ROUTINE

FORMAT:

```
EXIT ROUTINE
```

DESCRIPTION:

The **EXIT ROUTINE** statement enables you to exit from the executed routine.

10.13.2 REPEAT ROUTINE

FORMAT:

```
REPEAT ROUTINE
```

DESCRIPTION:

The **REPEAT ROUTINE** statement enables you to repeat the entire routine execution. When INTOUCH executes the *REPEAT ROUTINE* statement, the routine is reexecuted.

10.13.3 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) = 'NONE' THEN GOTO HELLO
branches out of block --   IF _EXIT THEN EXIT FOR
                         HELLO:
                           PRINT 'Hello!'
                    30 NEXT I
                    40 PRINT 'Finished'
                    50 END
```

11 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 Appendix A.

11.1 CAUSE EXCEPTION

FORMAT:

```
CAUSE EXCEPTION exception_number
```

EXAMPLE:

```
10 INPUT 'Pick a number between 1 and 10': PICK
20 IF PICK<1 OR PICK>10 THEN CAUSE EXCEPTION 1001
30 END

RNH
Pick a number between 1 and 10? 11   Error message:
                                       "Illegal number at 20"
INTOUCH
```

PURPOSE:

Use CAUSE 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 Appendix for a list of Exception Messages.

11.2 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
40 END
```

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.

```
RNH
Your name, please? JOHN
How old are you? 3X
Not a valid age
How old are you? 34

John is 34
```

11.2.1 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
40 HANDLER FIX_AVERAGE
     AVERAGE = 0
     CONTINUE
45 END HANDLER
50 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. Handl_name is the name of the handler. 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 below, 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 may occur before or after the protected block.

```
10 HANDLER FIX_AVERAGE
     AVERAGE = 0
     CONTINUE
   END HANDLER

   INPUT 'Enter total sales amount': TSALES
   INPUT 'Enter number of sales': NSALES
   WHEN EXCEPTION USE FIX_AVERAGE
     AVERAGE = TSALES/NSALES
   END WHEN
40 PRINT 'The average is'; AVERAGE
50 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
  first protected block ----- INPUT 'How old are you': AGE
                           END WHEN

                        20 INPUT 'Your name, please': NAME$

                           WHEN EXCEPTION USE NUMBERS

second protected block ----- INPUT 'Your birthdate': BIRTH
                           END WHEN
```

```
                    PRINT NAME$; ' was born on'; BIRTH
                50 HANDLER NUMBERS
handler routine ---   PRINT 'Enter numbers only, please.'
                      RETRY
                    END HANDLER
                60 END
```

11.2.2 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
```

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.

```
RNH
Your name, please? JOHN
How old are you? 3X
Not a valid age
How old are you?
```

11.2.3 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
   PRINT 'The average is:'; AVERAGE
   HANDLER FIX_AVERAGE
     AVERAGE = 0
     CONTINUE
   END HANDLER
50 END
```

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.

```
RNH
Enter total sales amount? 18.00
Enter number of sales? 0
The average is: 0
```

11.2.4 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
   PRINT 'The average is:'; AVERAGE
   HANDLER FIX_AVERAGE
     AVERAGE = 0
     PRINT 'Invalid numbers.  Try again.'
     RESUME 10
   END HANDLER
50 END
```

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.

```
RNH
Enter total sales amount? 18.00
Enter number of sales? 0
Invalid numbers.  Try again.
Enter total sales amount? 18.00
Enter number of sales? 3
The average is: 6
```

11.2.5 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 1
     EXIT HANDLER
   END HANDLER
30 END
```

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 may only be used within an exception handler.

```
RNH
Enter your age? thirty-four
Oops...
Non-numeric input when number expected at 10.1
```

12 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.

12.1 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...

INTOUCH
```

PURPOSE:

Use LIBRARY statements to specify the libraries you will use in your program.

DESCRIPTION:

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 may 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 make libraries for INTOUCH, you will need to follow the steps described below:

1 Write the source code using any VAX editor. The source code may be in BASIC, FORTRAN or any other VMS 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 may contain any number of subprograms.

2 Compile the source code with the appropriate compiler. For instance, compile this BASIC routine with the BASIC compiler:

```
$ BASIC ROUTINE.BAS
```

3 Write an options file, declaring the subprogram as a universal symbol. Use a text editor, and create a one-line file called MULT.OPT with the following line:

```
UNIVERSAL = MULT
```

4 Link the routine into a shareable image. Specify the options file in the link command. Use the command described below:

```
$ LINK/SHARE routine_name,options_file/OPTIONS
```

For the above BASIC routine, the LINK command would be:

```
$ LINK/SHARE ROUTINE,MULT/OPTIONS
```

You have now created the shareable image MULT.EXE, containing the routine MULT. You may now use this routine from within INTOUCH.

To use the routine, you need to name the shareable image with a LIBRARY statement. When you use the LIBRARY statement, be sure to supply the COMPLETE FILE SPECIFICATION, including device and directory, for the library:

```
$ LIBRARY "DUA0:[MYFILE]ROUTINE.EXE"
```

Finally, use the CALL statement to execute the routine:

```
10  CALL MULT(8%, 9%, ANSWER%)
    PRINT 'The answer is';ANSWER%
20  END
```

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.

12.2 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...
```

PURPOSE:

Use CALL to call and execute library routines. You can use these routines to perform procedures rather than writing the code yourself.

DESCRIPTION:

Use CALL to call and execute a routine from a library. 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...)
```

Optional Arguments

Some routines have optional arguments–arguments which **may** 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%]
40 END
```

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 may be changed by the routine they are
                passed to.


BY VALUE        By value.  Arguments passed by value may not 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 may 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 Appendix for more information.)

13 Data Structure Statements

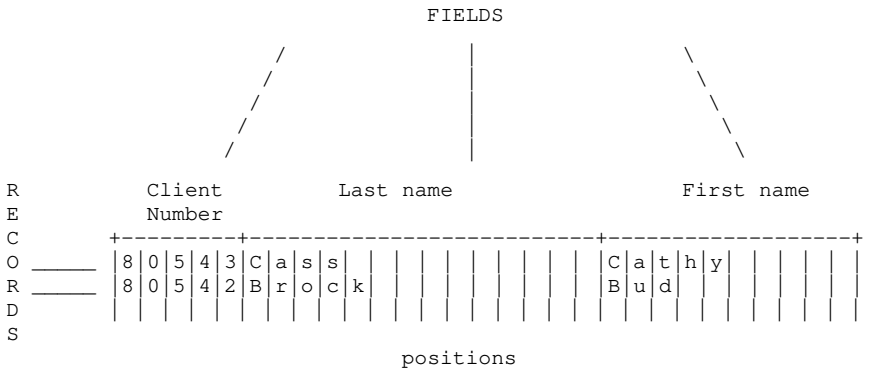
This chapter explains the INTOUCH data structure statements. One of the major features of INTOUCH is its ability to perform database operations as part of the language. The data structure statements allow you to manipulate data structures in your own programs.

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

13.1 General Information

INTOUCH's data management system stores data in **structures**. A structure looks something like this:



Each structure is made up of **records** and **fields**.

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, indicate the structure and the field you want; put parentheses around the field name:

```
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
60 END

RNH
Errant             Earl
```

13.2 Data Structure Statements

13.3 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 my 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 my California clients by last name
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676

INTOUCH
```

PURPOSE:

Use OPEN STRUCTURE to open a data structure. You must open the structure to reference fields, change field data or add new records to the structure.

DESCRIPTION:

OPEN STRUCTURE opens a data structure. The structure must be open in order to reference data in the structure. Struc_name is a name you assign to the structure, to refer to the structure within the program. The structure name must be unique. 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 on the "SETUP Routine" for a description of 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. If no ACCESS option is included, 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 doesn't exist, an exception is generated.

13.3.1 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.

13.3.2 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 can not be modified by others. This may speed up INTOUCH's structure statements (EXTRACTs especially). The particular effect depends on the file management system being used.

13.3.3 DATAFILE Option

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.

13.4 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 My California Clients'
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END

RNH
List of My California Clients
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676

INTOUCH
```

PURPOSE:

Use CLOSE STRUCTURE to close a structure from further access.

DESCRIPTION:

CLOSE closes a structure from further access. Once the structure is closed, you cannot reference fields or add them, and you cannot change field data. Struc_name is the name associated with the structure, the one assigned with the OPEN statement.

You can use the statement CLOSE ALL to close all structures and other files.

13.5 EXTRACT STRUCTURE

FORMAT:

```
EXTRACT STRUCTURE struc_name
        ---  block of code
[INCLUDE | EXCLUDE] cond_expr
[SORT [ASCENDING | DESCENDING] BY expression]
       ---
END EXTRACT
```

When INTOUCH does the 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 following program extracts information from each record in the structure. It displays each client's name and phone number.

```
10 OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
20 PRINT 'List of My Clients'
   PRINT
30 EXTRACT STRUCTURE CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   END EXTRACT
40 CLOSE STRUCTURE CL
50 END

RNH
List of My 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 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.

13.5.1 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. INTOUCH evaluates the expression. It checks the structure's index for records with matching keys, and extracts only these records.

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.

For example, we have a structure with the following 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

In the program below, the KEY option is used to extract the client with the ID number 80561.

```
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
50  END

RNH
Client:            Dale Derringer        80561
```

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.

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. For 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
50 END

RNH
Enter the lowest ID to check? 80540
Enter the highest ID to check? 80570
80542    Brock
80543    Cass
80561    Derringer
```

13.5.2 EXTRACT STRUCTURE, FIELD: PARTIAL KEY

FORMAT:

```
EXTRACT STRUCTURE struc_name, FIELD field_expr: PARTIAL KEY str_expr
```

DESCRIPTION:

The PARTIAL KEY option lookups in the EXTRACT STRUCTURE to look for part of a key value. For example, here is a structure with the following client information:

EXAMPLE:

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 program below creates an extract list containing only those clients with a last name starting with "ROS".

```
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
40  CLOSE STRUCTURE CL
50  END
```

13.5.3 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
15  CLOSE STRUCTURE CL
20  END
```

```
RNH
Client: Smith
Press return to continue? EXIT
Records extracted: 0
```

PURPOSE:

CANCEL EXTRACT cancels the current extract of a record and transfers control to the next statement after the END EXTRACT statement.

DESCRIPTION:

The statement can be used only within an EXTRACT block—that is, between an EXTRACT STRUCTURE and an END EXTRACT pair of statements.

13.5.4 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: Jones
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.

13.5.5 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
40 PRINT 'List of Clients in Area Code 619'
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
   CLOSE STRUCTURE CL
60 END
```

PURPOSE:

Use REEXTRACT 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.

DESCRIPTION:

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.

13.5.6 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
40      FOR EACH detail
          PRINT detail(prodnbr); TAB(7); detail(lineid); TAB(17); detail(qty)
        NEXT detail
50      END

RNH
```

```
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.

13.5.7 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, the **EXTRACT STRUCTURE struc_name1, SET 'set_name' USING struc_name2** this statement fetches all structures, (struc_name1) owned by another structure (struct_name2) within a given set.

13.5.8 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:

The **EXTRACT STRUCTURE struc_name1, SET 'set_name', FIELD field_expr: KEY str_expr** statement allows you to extract all structures specified by struc_name1 owned by the current record within the given set ('set_name') with field a field_expr equal to the given key.

13.6 INCLUDE Statement

The INCLUDE statement includes records depending on the value of a conditional expression.

FORMAT:

```
INCLUDE cond_expr
```

Cond_expr is a conditional expression. INTOUCH evaluates the expression. 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, we have 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

The program below creates an extract list containing only those clients located in California.

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
20  EXTRACT STRUCTURE CL
      INCLUDE CL(STATE) = 'CA'
    END EXTRACT
    PRINT 'List of My California Clients'
    PRINT
    FOR EACH CL
      PRINT CL(FIRST);' ';CL(LAST),CL(STATE)
    NEXT CL
    CLOSE STRUCTURE CL
50  END

RNH
List of My California Clients

Earl Errant         CA
Cathy Cass          CA
Dale Derringer      CA
```

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.

13.7 EXCLUDE Statement

The EXCLUDE statement excludes records from the extract list, depending on the value of a conditional expression.

FORMAT:

```
EXCLUDE cond_expr
```

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 below creates an extract list of all the clients in the client structure– except those with an area code of 619.

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
20  EXTRACT STRUCTURE CL
      EXCLUDE CL(PHONE)[1:3] = '619'
    END EXTRACT
30  PRINT 'List of My Clients'
    PRINT
    FOR EACH CL
      PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
    NEXT CL
    CLOSE STRUCTURE CL
50  END

RNH
List of My 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
```

Cathy Cass is excluded from the extract list because her phone number has an area code of 619.

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.

13.8 SORT Statement

FORMAT:

```
SORT [ASCENDING | DESCENDING] BY expr
```

The SORT statement sorts the records in an extract list in either descending or ascending 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. For example, a structure may have the following 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

The following program creates a list of these clients, sorted by last name.

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
    EXTRACT STRUCTURE CL
      SORT ASCENDING BY CL(LAST)
    END EXTRACT
    PRINT 'List of My Clients'
    PRINT
    FOR EACH CL
      PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
    NEXT CL
    CLOSE STRUCTURE CL
50  END

RNH
List of My 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
```

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
30  PRINT 'List of My Clients'
    PRINT
    FOR EACH CL
      PRINT CL(LAST);', ';CL(FIRST),CL(STATE)
    NEXT CL
   CLOSE STRUCTURE CL
50 END

RNH
List of My 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.

13.9 FOR EACH/NEXT

FORMAT:

```
FOR EACH struc_name
        ---
        ---  block of code
        ---
NEXT struc_name
```

EXAMPLE:

```
10 OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
   EXTRACT STRUCTURE CL
     INCLUDE CL(STATE) = 'CA'
     EXCLUDE CL(PHONE)[1:3] = '619'
     SORT ASCENDING BY CL(LAST)
   END EXTRACT
   PRINT 'List of My California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END
```

PURPOSE:

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.

DESCRIPTION:

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.

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|
```

The EXAMPLE program would run as follows:

```
RNH
List of My California Clients

Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

13.10 SET STRUCTURE

FORMAT:

```
SET STRUCTURE struc_name: struc_option [num_var | str_var]
```

PURPOSE:

Use SET to change various device and structure characteristics from within your programs.

DESCRIPTION:

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 below.

13.11 SET STRUCTURE: CURRENT

FORMAT:

```
SET STRUCTURE struc_name: CURRENT str_expr
```

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.

EXAMPLE:

```
10  DIM A$(100)
    LET I = 0
    OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
    EXTRACT STRUCTURE CL
    END EXTRACT
    FOR EACH CL
      PRINT CL(LAST); ', '; CL(FIRST)
      INPUT 'Would you like to see this record (Y/N)': YN$
      IF  YN$ = 'Y'  THEN
        LET I = I + 1
        ASK STRUCTURE CL: CURRENT A$(I)
      END IF
    NEXT CL
    PRINT
    FOR J = 1 TO I
      SET STRUCTURE CL: CURRENT A$(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
```

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.

```
SET STRUCTURE struc_name: CURRENT ""
```

13.12 SET STRUCTURE, FIELD: KEY

FORMAT:

```
SET STRUCTURE struc_name, FIELD field_expr: KEY str_expr
```

The FIELD option lets you get a record by means of a key field in a structure. You can use CALL SETUP's SHOW DEFINITIONS (please refer to Chapter on "Setup Routine") 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 the string expression.

For example, if you want to look in the example CLIENT structure for an ID:

EXAMPLE:

```
10  OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
20  LINE INPUT 'Enter an ID': ID$
    SET STRUCTURE CL, FIELD ID: KEY ID$
    IF  _EXTRACTED = 0  THEN
       MESSAGE ERROR: 'Not found'
    ELSE
       PRINT CL(LAST)
       END IF
40  END
```

_EXTRACTED contains the number of records extracted. If the operation fails, _EXTRACTED = 0 and error message is displayed.

13.13 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? S
32001 Smith
```

DESCRIPTION:

This statement retrieves the first record matching the partial key in str_expr.

13.14 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.

13.15 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, to provide 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.

13.16 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)

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.

13.17 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
30  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.

13.18 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:

The **SET STRUCTURE struc_name1, SET 'set_name', USING struc_name2: EXPRESSION 'owner'** 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'.

13.19 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:

The **SET STRUCTURE struc_name1, SET 'set_name': EXPRESSION 'owner'** statement is used for DBMS handling.

13.20 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:

The **SET STRUCTURE struc_name1, SET 'set_name', USING 'struc_name2', FIELD field_expr: KEY str_expr** 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.

13.21 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:

The **SET STRUCTURE struc_name1, SET 'set_name', USING struc_name2** statement used for DBMS handling fetches the first struc_name1 within a set. The record referred to by struc_name2 is the owner.

13.22 ASK STRUCTURE

FORMAT:

```
ASK STRUCTURE struc_name: struc_option [num_var | str_var]
```

PURPOSE:

Use ASK to ask about various device and structure characteristics from within programs.

DESCRIPTION:

You can use the ASK statement to find out about characteristics of INTOUCH structures.

Struc_name is the name of a structure whose characteristics are being asked. Struc_option can be any of the structure options available.

13.23 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 age
    Prompt      : Age
    Position    : 190
    Field length: 1
    Rpt. heading: Age
    Print mask  : <##
    Screen mask : ###
    Help        :
```

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 which kind of information to obtain about this field. The information is stored in the variable specified.

The following *ITEMS* are available with ASK STRUCTURE FIELD:

- *ACCESS*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_name: ACCESS str_var
```

EXAMPLE:

```
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
```

DESCRIPTION:

This ACCESS retrieves the access rules for the specified field of the specified structure.

- *APPLICATION*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: APPLICATION str_var
```

EXAMPLE:

```
10 OPEN STRUCTURE cl : NAME 'tti_run:client'
   ASK STRUCTURE cl, FIELD id : APPLICATION str$
   PRINT str$
20 END

RNH
REPORTING_ID
```

DESCRIPTION:

APPLICATION returns a name of an application for this structure and field in a string variable.

- *ATTRIBUTES*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: ATTRIBUTES str_var
```

DESCRIPTION:

ATTRIBUTES returns the INTOUCH field attributes NUM (number), UC (upper-case), etc., in a string variable.

This statement returns the date format and numeric scale (DATE:YMD or NUM:3, rather than just DATE and NUM).

The attributes are:

Attributes	Description
NUM	Numeric
NUM:scale	Numeric with scale factor
UC	Upper case
LS	Sign on far left for negative number
DATE	Date in the format indicated
EB	EBCDIC
EB	Information to be stored as EBCDIC coded characters.
RJ	Causes data to be right-justified.
RS	Causes negative numbers to have a minus sign as the far right character.
UC	Enters all letters in upper-case.
ZF	Zero fill on left
ZF	Data entered is zero filled.
ZS	Zero suppress
ZS	Causes all leading zeros to be suppressed and converted to spaces.

- *CHANGEABLE*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: CHANGEABLE num_var
```

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

RNH
 0
 1
```

DESCRIPTION:

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.

- *CLASSIFICATION*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: CLASSIFICATION str_var
```

EXAMPLE:

```
10 OPEN STRUCTURE cl : NAME 'tti_run:client'
   ASK STRUCTURE cl, FIELD last : CLASSIFICATION str$
   PRINT str$
20 END

RNH
NAME
```

DESCRIPTION:

CLASSIFICATION returns the classification for this field in a string variable.

- *DATATYPE*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: DATATYPE str_var
```

DESCRIPTION:

DATATYPE returns the data type, such as CH (character), IN (integer), etc., in a string variable.

- *DESCRIPTION*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: DESCRIPTION str_var
```

DESCRIPTION:

DESCRIPTION returns the description for this field in a string variable.

- *HEADING*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: HEADING str_var
```

DESCRIPTION:

HEADING returns the column heading for this field in a string variable.

- *HELP*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: HELP str_var
```

DESCRIPTION:

HELP returns the help text for this field in a string variable.

- *KEYED*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: KEYED num_var
```

DESCRIPTION:

KEYED returns a value of true or false in a numeric variable. If it is a key field, then the value is true; otherwise, the value false.

- *LENGTH*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: LENGTH num_var
```

DESCRIPTION:

LENGTH returns the length of this field in a numeric variable.

- *NAME*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: NAME str_var
```

DESCRIPTION:

NAME returns the name of the field in a string variable.

- *NULL*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_name: NULL int_var
```

DESCRIPTION:

If a field is null, this statement returns TRUE. If the field is not null, the statement returns FALSE.

- *NUMBER*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: NUMBER num_var
```

DESCRIPTION:

The NUMBER returns the field number of this field in the numeric variable specified. Fields are numbered sequentially. If the field does not exist, INTOUCH returns a value of 0.

- *OPTIMIZED*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: OPTIMIZED num_var
```

EXAMPLE:

```
10 OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
   ASK STRUCTURE CL, FIELD id: OPTIMIZED z
   PRINT z
   CLOSE STRUCTURE CL
20 END

RNH
 1
```

DESCRIPTION:

OPTIMIZED returns a value of TRUE or FALSE. If the key field in field_expr is optimized, the value is TRUE; otherwise, the value is FALSE.

- *POSITION*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: POSITION num_var
```

DESCRIPTION:

POSITION returns the starting position for this field in a numeric variable.

- *PRINTMASK*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: PRINTMASK str_var
```

DESCRIPTION:

PRINTMASK returns the print mask for this field in a string variable.

- *PROMPT*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: PROMPT str_var
```

DESCRIPTION:

PROMPT returns the prompt for this field in a string variable.

- *SCREENMASK*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: SCREENMASK str_var
```

DESCRIPTION:

SCREENMASK returns the screen mask for this field in a string variable. This option is not currently used.

FIELD expressions

The field_expr 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 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
```

```
10  OPEN STRUCTURE CL: NAME 'tti_run:client'
20  DO
      GOSUB ASK_FIELDS
      IF  _EXIT  THEN  EXIT DO
      GOSUB SHOW_DATA
    LOOP
    STOP

30  ASK_FIELDS:
    DO
      LINE INPUT 'Enter fields to display': FIELDS$
      IF  _EXIT  THEN  EXIT DO
      ERROR = FALSE
      FOR F = 1 TO ELEMENTS(FIELDS$)
        FIELD$ = ELEMENT$(FIELDS$, F)
        WHEN EXCEPTION IN
          ASK STRUCTURE CL, FIELD #FIELD$: NUMBER Z
        USE
          MESSAGE ERROR: 'Illegal field:' + FIELD$
          ERROR = TRUE
        END WHEN
      NEXT F
      IF ERROR THEN REPEAT DO
    END DO
    RETURN

50  SHOW_DATA:
    EXTRACT STRUCTURE CL
      FOR F = 1 TO ELEMENTS(FIELDS$)
        FIELD$ = ELEMENT$(FIELDS$, F)
        PRINT CL(#FIELD$)
      NEXT F
    PRINT
    END EXTRACT
    RETURN
70  END
```

- *VRULES*

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: VRULES str_var
```

EXAMPLE:

```
10 OPEN STRUCTURE cl : NAME 'tti_run:client'
   ASK STRUCTURE cl, FIELD bday : VRULES str$
   PRINT str$
20 END

RNH
date ymd; minlength 8
```

DESCRIPTION:

VRULES returns the validation rules for this field in a string variable.

13.24 ASK STRUCTURE: CURRENT

FORMAT:

```
ASK STRUCTURE struc_name: CURRENT str_var
```

EXAMPLE:

```
10  DIM A$(100)
    OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
20  EXTRACT STRUCTURE CL
    END EXTRACT
    FOR EACH CL
      PRINT CL(LAST); ','; CL(FIRST)
      INPUT 'Would you like to see this record (Y/N)': YN$
      IF  YN$ = 'Y'  THEN
        LET I = I + 1
        ASK STRUCTURE CL: CURRENT A$(I)
      END IF
    NEXT CL
35  PRINT
40  FOR J = 1 TO I
      SET STRUCTURE CL: CURRENT A$(J)
      PRINT CL(LAST); ','; CL(FIRST), CL(STATE), CL(PHONE)
50  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:

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 option to get this record.

INTOUCH returns a zero length string if there is no CURRENT record.

13.25 ASK STRUCTURE: FIELDS

FORMAT:

```
ASK STRUCTURE struc_name: FIELDS num_var
```

DESCRIPTION:

You can find out the number of fields in a structure with the FIELDS option. The number is assigned to the numeric variable specified.

13.26 ASK STRUCTURE: KEYS

FORMAT:

```
ASK STRUCTURE struc_name: KEYS num_var
```

DESCRIPTION:

The KEYS option returns number of keys that are accessible by INTOUCH. It returns the value of 0 if no keys are available.

13.27 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 13–1 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.

13.28 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
 11
```

DESCRIPTION:

ASK STRUCTURE: EXTRACTED asks the operating system for the last extracted count for the structure given.

13.29 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:

This statement asks the operating system for the ID of a structure and returns it into the string variable given.

13.30 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 pointer.

13.31 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.

13.32 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
```

PURPOSE:

This statement returns the record size of the structure data file.

13.33 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:

This statement retrieves the access rules for the specified structure. Security level, data file read, write, update, and delete rules are returned.

13.34 ASK | SET STRUCTURE: TIMEOUT

FORMAT:

```
ASK STRUCTURE struc_name: TIMEOUT int_var
SET STRUCTURE struc_name: TIMEOUT int_expr
```

DESCRIPTION:

ASK STRUCTURE statement asks for the current TIMEOUT setting.

The SET STRUCTURE statement is used when a record is locked for over the int_expr. INTOUCH then returns a TIMEOUT error.

13.35 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'
    GOSUB DO_WORK
    STOP

20  ROUTINE DO_WORK
    ASK STRUCTURE #STR$, FIELD #FLD$: DESCRIPTION DSC$
    PRINT 'Description is: '; DSC$
    END ROUTINE

90  END

RNH
Description is: Client ID number

INTOUCH
```

PURPOSE:

You can use a string expression for the structure name in an ASK STRUCTURE or SET STRUCTURE statement.

DESCRIPTION:

In the ASK STRUCTURE and SET STRUCTURE statements, you can use a string expression for the structure. This allows you to write generalized code to work for several structures.

13.36 ASK STRUCTURE: ENGINE

FORMAT:

```
ASK STRUCTURE struc_name : ENGINE str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl : name 'tti_run:vendor'
30  ASK STRUCTURE  cl : ENGINE ename$
40  PRINT 'DATABASE ENGINE is:'; ename$
50  END

RNH

DATABASE ENGINE is: RMS
```

DESCRIPTION:

The **ASK STRUCTURE struc_name : ENGINE str_var** statement returns the name of the database engine being used for the specified structure in str_var.

13.37 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
   CLOSE STRUCTURE CL
50 END
```

PURPOSE:

Use DELETE STRUCTURE to delete a record from a structure.

DESCRIPTION:

DELETE STRUCTURE deletes the current record from the structure specified. Struc_name is a structure name associated with the structure. For instance, suppose there is a structure with the following 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

The EXAMPLE program would run as follows:

```
RNH

Deleting Earl Errant...record deleted
Deleting Cathy Cass...record deleted
Deleting Dale Derringer...record deleted
```

13.38 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$
   ADD STRUCTURE CL
     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
   CLOSE STRUCTURE CL
50 END
```

PURPOSE:

Use ADD STRUCTURE to add records to a structure.

DESCRIPTION:

ADD STRUCTURE adds a record to a structure. The ADD STRUCTURE statement begins the add block. Struc_name is a name associated with the structure. 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. Use this block of code with LET statements 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 statements, it evaluates the expression and then assigns the value of this expression to the field specified. END ADD writes out the new record.

13.39 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)
      INPUT 'First name': CL(FIRST)
      PRINT 'Adding client'
    END ADD
20  END

RNH
Client ID? EXIT
```

DESCRIPTION:

CANCEL ADD cancels the current adding of a record and transfers control to the next statement after the END ADD statement.

This statement can be used only within an ADD block—that is, between an ADD STRUCTURE and an END ADD pair of statements.

13.40 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.

13.41 LOCK | UNLOCK STRUCTURE

FORMAT:

```
[LOCK | UNLOCK] STRUCTURE struc_name
```

EXAMPLE:

```
10 OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT", ACCESS OUTIN
   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
   CLOSE STRUCTURE CL
50 END
```

PURPOSE:

Use LOCK/UNLOCK STRUCTURE to lock or unlock the data 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.

DESCRIPTION:

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 Writing and Debugging INTOUCH Programs

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 Appendix A for further description of 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.

14.1 DEBUG ON/OFF

14.1.1 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:

DEBUG ON enables INTOUCH's DEBUG system. 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 may 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.

14.1.2 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 disables INTOUCH's DEBUG system. DEBUG OFF may 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.

14.2 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" may be abbreviated to "STATS" (STATS ON, STATS OFF, LIST STATS). This abbreviation will be used frequently in this Guide.

14.3 The TRACE and STATISTICS Features

Both the TRACE and STATISTICS features operate on program lines.

```
10 PRINT A$
```

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
```

The first line is noted as 10, the program line following this is 10.1. The next 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
```

14.3.1 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.

To use TRACE, you must have the frame on.

DESCRIPTION:

TRACE ON enables INTOUCH's trace feature. The trace feature lists each program line as it is executed. TRACE ON may 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$(5)
   MAIN: FOR I = 1 TO 5
           INPUT 'Your name, please': NAME$(I)
           IF _EXIT THEN EXIT FOR
           PRINT 'Hello, '; NAME$(I); '!'
         NEXT I
20 END
```

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.

Example 14–1 TRACE ON

```
INTOUCH 4.1                              Program: HELLO  DEBUG
The Next Generation Language             Status : RUN
-------------------------------------------------------------------
TRACE ON

INTOUCH

RNH
Your name, please? John
Hello, John!
Your name, please?

-------------------------------------------------------------------
[10] [MAIN]  [MAIN.1]  [MAIN.2]  [MAIN.3]  [MAIN.4]  [MAIN.1]

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

14.3.2 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.

14.3.3 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 may 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.

14.3.4 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 may 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.

14.3.5 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 the statistics recorded by the statistics feature. It lists each program line along with the number of times the line was executed and the execution time of each line.

Example 14–2 LIST STATS

```
INTOUCH 4.1                          Program: NONAME STATS DEBUG
The Next Generation Language         Status :
------------------------------------------------------------------
LIST STATS
NONAME 6-DEC-1985 18:51
1   0.00  10 I = 1
100   0.25     DO WHILE I<100
 99   1.07       ROOT = SQR(I)
 99   5.77       PRINT I, ROOT
 99   0.28       I = I+1
 99      0.04             LOOP
  1      0.02          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 "LIST" statement are also available with LIST STATS. (See Chapter Two for details.)

14.4 BREAK and STEP

14.4.1 BREAK

FORMAT:

```
BREAK
```

EXAMPLE:

```
1050 BREAK
```

Use BREAK to stop program execution when DEBUG is ON. For instance, you might use BREAK to stop program execution 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 14–3 BREAK

```
INTOUCH 4.1                              Program: NONAME  DEBUG
The Next Generation Language             Status : BREAK at 10.2
-------------------------------------------------------------------
 10 PRINT "1",
    PRINT "2",
    BREAK
    PRINT "3",
    PRINT "4"
 20 END
 DEBUG ON

 INTOUCH

 RNH
 1                 2                BREAK at 10.2

 INTOUCH
-------------------------------------------------------------------

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

14.4.2 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 14–4 STEP

```
INTOUCH 4.1                              Program: NONAME  DEBUG
The Next Generation Language             Status :
-------------------------------------------------------------------
 5  DEBUG ON
 10 PRINT "1",
    PRINT "2",
    BREAK
    PRINT "3",
    PRINT "4"
 20 END
 RNH
 1                2                 BREAK at 10.2

 INTOUCH

 STEP
 3
 INTOUCH
-------------------------------------------------------------------
                         PRINT "4"

 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.

Example 14–5 STEP - With Number

```
INTOUCH 4.1                              Program: NONAME  DEBUG
The Next Generation Language             Status :
-------------------------------------------------------------

 RNH
 1                 2                 BREAK at 10.2

 INTOUCH

 STEP 5
 3                 4

 INTOUCH

-------------------------------------------------------------

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

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

14.5 GO–Continuing Program Execution

14.5.1 GO

FORMAT:

```
GO
```

EXAMPLE:

```
GO
```

PURPOSE:

Continue program execution after it has been interrupted.

DESCRIPTION:

GO resumes program execution after it has been interrupted. 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 14–6 GO

```
INTOUCH 4.1                              Program: NONAME   DEBUG
The Next Generation Language             Status :
-------------------------------------------------------------------
 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) ! type an immediate mode command
 2

 INTOUCH

 25  PRINT I;  ! add a new line of code
 GO
  4 5 6
 INTOUCH
-------------------------------------------------------------------

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

14.6 Comments

You may 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$(10)
20 REM Main body of program
   FOR I = 1 TO 10                          ! Begin the loop
     INPUT 'Your name, please': NAME$(I)    ! Get a name
     IF _EXIT THEN EXIT FOR                 !End if they want
     PRINT 'Hello, '; NAME$(I)              !Say hello
   NEXT I                                   !End the loop
30 END                                      !END the program

RNH
Your name, please? Marian
Hello, Marian
Your name, please? EXIT

INTOUCH
```

14.6.1 REM

FORMAT:

```
REM comment_text
```

EXAMPLE:

```
5  REM Get a name
   INPUT 'Your name, please': NAME$
10 PRINT 'Hello, '; NAME$
20 END

RNH
Your name, please? Lucy
Hello, Lucy

INTOUCH
```

PURPOSE:

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

DESCRIPTION:

When a program is listed, the REM line is displayed as it was typed. For example:

```
LIST
NONAME      2-FEB-1985 8:50
5  REM Get a name and say hello
   INPUT 'Your name, please': NAME$
10 PRINT 'Hello, '; NAME$
20 END

RNH
Your name, please? Glenna
Hello, Glenna

INTOUCH
```

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. It continues execution at the next executable line. When INTOUCH executes the above program, it ignores the REM statement and begins executing at the INPUT statement. REM can be used anywhere in a multiple line statement. For example:

```
5  INPUT 'Your name, please': NAME$
   REM Say hello
   PRINT 'Hello, '; NAME$
10 END

RNH
Your name, please? Tom
Hello, Tom

INTOUCH
```

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

14.6.2 ! COMMENT_TEXT

FORMAT:

```
! comment_text
```

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$  ! Get a name
   PRINT 'Hello, '; NAME$            ! Say hello
20 END

RNH
Your name, please? Mike
Hello, Mike

INTOUCH
```

PURPOSE:

Use the ! to put comments in a program.

DESCRIPTION:

You may want to use comments to clarify parts of your program (! This program prints W2 forms), to give dates, provide sources (! See Dr. McCann) etc.

When the program is listed, the "!" line is displayed as it was typed. For example:

```
LIST
NONAME      2-FEB-1985 9:00
10 INPUT 'Your name, please': NAME$  ! Get a name
   PRINT 'Hello, '; NAME$            ! Say hello
20 END
```

When INTOUCH executes the above program, it executes the INPUT statement, ignores the "!" and the comment text following it. INTOUCH continues execution at the INPUT 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:

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

14.6.3 ~ Used to Border Part of a Program

FORMAT:

```
~
```

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$
   PRINT 'Hello, '; NAME$
   ~~~~~~~~~~~~~~~~
     INPUT 'Again (Y/N)': ANSW$
     IF ANSW$ = 'Y' THEN GOTO 10
   ~~~~~~~~~~~~~~~~
20 END

RNH
Your name, please? Liz
Hello, Liz
AGAIN (Y/N)? Y
Your name, please? Liz
Hello, Liz
AGAIN (Y/N)? N

INTOUCH
```

PURPOSE:

Use the tilde to highlight and 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. However, INTOUCH ignores them when it executes the program. (Tildes within quotes ('~') are treated as string constants.) In the above program, INTOUCH would execute all the statements in the program—the INPUT and IF/THEN statements bordered by the tildes.

14.7 Program, End, Stop and Halt Statements

PROGRAM is used to name programs. END is used to END program execution. 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.

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

HALT interrupts program execution but does not close files or write active output buffers.

14.7.1 PROGRAM

FORMAT:

```
PROGRAM prog_name
```

EXAMPLE:

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

PURPOSE:

Use PROGRAM to name your program.

DESCRIPTION:

PROGRAM is used to name programs. Prog_name is the program name. It must meet the specifications for variable names. Therefore, it:

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

14.7.2 END

FORMAT:

```
END
```

EXAMPLE:

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

RNH
Your name, please? 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.

14.7.3 STOP

FORMAT:

```
STOP
```

EXAMPLE:

```
10 INPUT 'Your name, please': 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

Your name, please? Ted
How old are you? 58
Ted is 58

INTOUCH
```

PURPOSE:

Use STOP to terminate program execution where you don't want to mark the physical end of your program.

DESCRIPTION:

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

Example 14–7 STOP—Sample Program

Example 14–7 Cont'd. on next page

Example 14–7 (Cont.) STOP—Sample Program

```
INTOUCH 4.1                              Program: NONAME
The Next Generation Language             Status :
-------------------------------------------------------------------

 RNH
 Your name, please? JOHN
 How old are you? .5
 Not a valid age
 Halt at 10.4

 INTOUCH

-------------------------------------------------------------------

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

14.7.4 HALT

FORMAT:

```
HALT
```

EXAMPLE:

```
10 INPUT 'Your name, please': 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
Your name, please? Chris
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:

Example 14–8 HALT—Sample Program

```
INTOUCH 4.1                                   Program: NONAME
The Next Generation Language                  Status :
------------------------------------------------------------------------
 RNH
 Your name, please? JOHN
 How old are you? .5
 NOT A VALID AGE
 Halt at 20.4

 INTOUCH

 LET AGE = 34

 INTOUCH

 GO
 JOHN is 34

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

14.7.5 RANDOMIZE

FORMAT:

```
RANDOMIZE
```

EXAMPLE:

```
5  RANDOMIZE
10 X = RND
   PRINT X
20 END

RNH
 .244013674718

INTOUCH

RNH
 .524856061388

INTOUCH
```

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 Appendix on functions for more information on the RND function.)

15 SETUP Routine

15.1 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:

```
DBMS
RMS
USERBASE
POISE
FASTFILE
S1032
Rdb
DBASE
ADABAS
```

Procedures:

The following procedures are available in SETUP:

- DEFINE

  Define fields in a structure. Use DEFINE when you add new fields, modify existing ones, or delete fields.

- SHOW

  Show field definitions. Use SHOW to see a display of the fields defined for this structure.

- CREATE

  Create a DATA File.

- EXIT

  Exit INTOUCH's 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 defined on the following pages.

15.1.1 Entering SETUP

To enter SETUP, type CALL SETUP at the INTOUCH prompt:

Example 15–1: To Use Setup

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status :

 INTOUCH

 CALL SETUP



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

SETUP will ask you a number of questions including:

- structure name
- data file name
- record system being used
- definition file

Example 15–2 Setup Prompts

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status :
          S e t   u p   D a t a   S t r u c t u r e s

 Structure name? CUSTOMER
 Data file? CUSTOMER.DAT
 Record system? RMS
 Definition file? CUSTOMER.DEF


 Procedure? define

             Procedures:  define, show, create, exit

 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 may define fields in the structure or display field definitions.

15.2 SETUP Prompts

The following pages describe the SETUP prompts and your options for answering them.

- **Structure name?** CUSTOMER

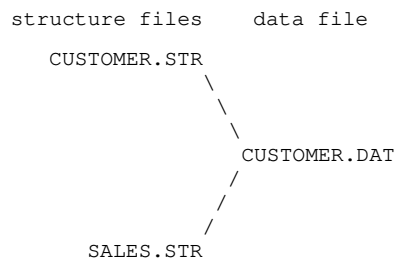
Enter a structure name. If you are creating a new structure, this will be the name of the structure file. The default extension for the file is .STR.

Your structure name can be any VMS file specification. (See chapter on descriptions of the VMS file specifications.)

- **Data file?** CUSTOMER.DAT

Enter the name of the data file 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 may 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 data file name can be any VMS file specification. See chapter on descriptions of the VMS file specifications.

- **Record system?** RMS

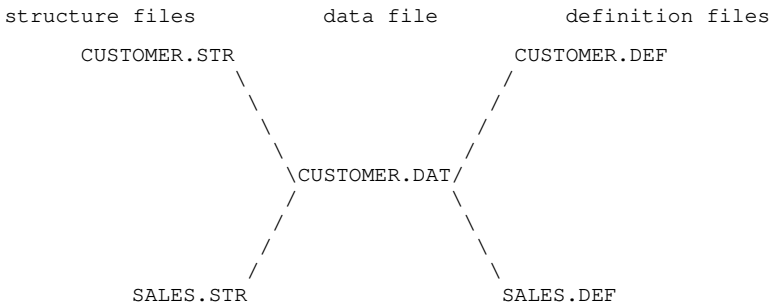
Enter the record management system you wish to use. INTOUCH supports several record management systems. They are displayed at the bottom of the screen when you get the record system prompt. (Please refer to the manual INTOUCH Interfaces to VAX Database Engines for an explanation of DBMS, ADABAS, Rdb, dBASE and CDD.)

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 record management system specified in the structure file. FASTFILE is an INTOUCH record management system designed for speed.

- **Definition file?** CUSTOMER

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 chapter on descriptions on the VMS file specifications.)

- Definition file? NONE

The keyword NONE is valid when the datafile is RDB or dBASE. When you enter NONE, INTOUCH uses the definitions embedded in the database engines with the fields.

If you are using CDD, you can enter CDD> and a name of a CDD/Plus Directory and recordname:

- Definition file?

```
CDD>DEVDISK:[DEVINT.CDDTEST]CUSTOMER
    ^                       ^^      ^
    |                       ||      |
    +-----------------------++------+
       Directory name        record
                             name
```

CDD Plus access is read only at this time.

- **Procedure?** DEFINE

15.3 DEFINE Procedure

You can use the DEFINE procedure to define the fields in a structure.

Choose the DEFINE procedure by typing DEFINE at the Procedure prompt. You will be asked first for the name of the field you wish to add or modify. You will then be prompted for information about the field.

Example 15–3 DEFINE Procedure

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                      Definitions for CUSTOMER

 Structure name? CUSTOMER
 Data file? CUSTOMER.DAT
 Record system? RMS
 Definition file? CUSTOMER.DEF

 Procedure? DEFINE

             Procedures:  define, show, create, exit

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

Next, you will be prompted to define the contents of the fields:

Example 15–4 Field Contents Prompts

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                     Definitions for CUSTOMER

 Field name? ID
 Description? Customer ID Number
 Data type? CH
 First position? 1
 Field length? 5
 Attributes? none
 Prompt text? Customer ID Number
 Report heading? Customer ID Number
 Print mask?
 Screen mask?
 Help text? Enter a 5-digit ID number
 Access rules? read:x


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

- **Field name?**

Enter the name of the field you wish to add or modify. The field name:

- must begin with a letter.
- may only consist of numbers, letters and underscores "_".
- may be up to 32 characters long.

If you are adding a new field, the field name will be used to identify this field. (See the SHOW procedure for a list of field names.)

You can delete a field by typing a minus sign "-", and then the field name. For instance:

```
Field name? -ID
```

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. Change the defaults by retyping the response.

To return to the previous prompt, type a backslash "\" and then press the RETURN key. SETUP will return you to the previous prompt where you may change your response.

- **Description?** Customer ID number

Enter a description of the field. Your description:

- can consist of any characters on the keyboard.
- can be up to 30 characters long.

If you are adding this field, the field name is the default description.

- **Data type?** CH

Enter the data type for this field. The data type is the internal representation of the data. It may be:

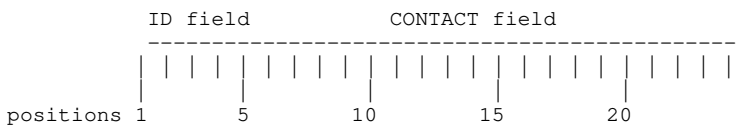
```
AC      ASCII counted

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)
DS      IBIS date     (length=6)

EB      EBCDIC

FL      Floating point

GF      G floating point

IN      Integer signed

IU      Integer unsigned

PF      Packed floating point

PZ      Packed zipcode

QS      Quadword signed

RO      Right overpunch

RS      Right sign separate

UN      Undefined

ZE      Zoned EBCDIC

ZN      Zoned punch; used by DIBOL
```

The DATE attribute is currently only used by applications and has no effect within INTOUCH.

- **First position?** 1

Enter the position at which this field will start. Character positions are numbered sequentially. For example:

Figure 15–1 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.

- **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 enter the ending-position for the field length. You enter "P ending-position".

Example:

```
Length?  P46
```

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.

- **Attributes?** none

Enter the attributes for this field. The field attributes available are:

```
DATE:YMD        This field contains a date.  The field must
     MDY        be six or eight digits long.

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.

EB              Information to be stored as EBCDIC coded
                characters.

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.

NUM             Allows numbers only.

NUM:scale       SCALE specifies the scale factor -- how many
                decimal places to use.  Scale factors may be
                positive or negative (from -99 to 99).

RJ              Causes data to be right-justified.

UC              Enters all letters in upper-case.

ZF              Data entered is zero filled.

ZS              Causes all leading zeros to be suppressed and
                converted to spaces.
```

If you do not want any of these field attributes, enter NONE at the prompt. If you are adding a new field, NONE is the default. If you are modifying an existing field, the default will be the attribute you have previously specified.

- **Prompt text?** Customer ID Number

Enter the prompt text for this field. The prompt text will be used when data is entered for this field. The prompt text:

- can use 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.

- **Report heading?** Main Phone Number

Enter the heading you want to appear on reports. This heading will appear at the top of the column containing this field's data. If you are adding a new record, the default report heading is the description.

- **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 field information, or when you use one of the INTOUCH routines. Print masks are created with the # and @ characters just as formats are created in PRINT USING. (See "Printing Data to a Terminal" for more information.)

- **Screen mask?**

The screen mask is used to enter data into this field. This prompt should be left blank for the time being.

- **Help text?**

Enter the help text for this field. The help text can be displayed during data entry if the user types HELP at this field prompt. Your help text:

- can be up to 70 characters long.
- can consist of any characters on the keyboard.

- **Access rules?** normal

Enter the access rules for this field. The access rules determine who will have access to this field. NORMAL lets users add, read and change field information.

When you have answered all the field prompts, the screen will clear and you may enter definitions for the next field:

```
Field name? PHONE
Description? Main phone number
Data type? CH
First position? 10
Field length? 137
Attributes? num, rj
Prompt text? Main phone number
Report heading? Main phone number
Print mask? (###)~ ###~-####
Screen mask?
Help text? Enter a phone number
Access rules? normal
```

When you have finished defining fields, type EXIT at the field prompt, or while holding down the CTRL key, press Z.

- Field name? EXIT

You will return to the Procedure prompt.

15.4 SHOW Procedure

The SHOW procedure displays information about the structure.

Choose the SHOW procedure by typing SHOW at the Procedure prompt:

You will be asked if you want the definitions for the defined fields. You may currently SHOW field definitions.

Example 15–5 SHOW

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :




 Procedure? SHOW
             Procedures:  define, show, create, exit
 EXIT = Exit                                 \ = Back   HELP = Help
```

15.5 Definitions

SHOW DEFINITIONS displays the following information for all defined fields: field name, description, beginning position, field length, data type, numeric attribute, scale factor, right-justification, zero fill and upper-casing. Field information is displayed in order by starting position.

To display definitions, at the Show prompt, type in definitions:

Example 15–6 SHOW Definitions

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :




 Show? definitions
                  Show: definitions, structure
 EXIT = Exit                                 \ = Back   HELP = Help
```

Example 15–6 Cont'd. on next page

Example 15–6 (Cont.) SHOW Definitions

Next, you see the Definitions displayed:

Example 15–7 Field Information Displayed

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                    Definitions for CUSTOMER

Field name   Description        Pos - len  DT   N   SC   R  Z  U
----------   -----------------  ---------  --   -   --   -  -  -
 ID          Customer ID number   1 -   5  CH   N        N  N  N
 CONTACT     Primary contact      6 -  35  CH   N        N  N  N
 COMPANY     Company name        41 -  30  CH   N        N  N  N
 ADDR        Address line        71 -  25  CH   N        N  N  N
 CITY        City                96 -  30  CH   N        N  N  N
 STATE       State              126 -   2  CH   N        N  N  N
 ZIP         Zip code           128 -   9  CH   Y        N  N  N
 PHONE       Main phone number  137 -  10  CH   Y        N  N  N

                     Press RETURN to continue

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

15.6 Show Structure

The SHOW STRUCTURE menu item displays the structure name. At the Procedure prompt, enter SHOW STR:

Example 15–8 SHOW STRUCTURE

Example 15–8 Cont'd. on next page

Example 15–8 (Cont.) SHOW STRUCTURE

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                    S H O W    S T R U C T U R E


 Procedure? SHO STR

             Procedures:  define, show, create, exit

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

Next, you see the structure displayed:

Example 15–9 Structure Displayed

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :

                     S H O W   S T R U C T U R E

  Structure name : AR_DATA:CUSTOMER.STR
  Data file      : AR_DATA:CUSTOMER.DAT
  Record system  : RMS
  Definition file: AR_DATA:CUSTOMER.DEF
  Access levels  : SECURITY:N, READ:N, WRITE:N, UPDATE:N, DELETE:N

             Procedures:  define, show, create, exit

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

After you have looked at the field definitions, type EXIT at the Procedure prompt. INTOUCH will return to the INTOUCH prompt:

Example 15–10 EXIT

Example 15–10 Cont'd. on next page

Example 15–10 (Cont.) EXIT

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :




 Procedure? EXIT
                  Show: definitions, structure
 EXIT = Exit                                 \ = Back   HELP = Help
```

15.7 CREATE Procedure

Choose the CREATE procedure by typing CREATE at the Procedure prompt.

You will be asked for the record size of your structure records. The legal record sizes depend on the file management system you use. For RMS, sizes range from 1 to approximately 32000. After you have entered the record size, your data file will be created using the appropriate record management system.

Note

Some record management systems may ask additional questions.

Example 15–11 CREATE

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :




 Procedure? CREATE
             Procedures:  define, show, create, exit
 EXIT = Exit                                 \ = Back   HELP = Help
```

Example 15–11 Cont'd. on next page

Example 15–11 (Cont.) CREATE

Next, you will be prompted for the file organization:

- **File Organization?** INDEXED

Example 15–12 File Organization Prompt

```
INTOUCH 4.1                  Program: NONAME
The Next Generation Language  Status :

                        C r e a t e   R M S   f i l e

Data file        :  CUSTOMER.DAT




File organization? INDEXED___

                           Choices: indexed, sequential
EXIT = Exit                                              \ = Back  HELP = Help
```

Next, you will be prompted to define the contents of the fields:

Example 15–13 Field Contents Prompts

Example 15–13 Cont'd. on next page

Example 15–13 (Cont.) Field Contents Prompts

```
INTOUCH 4.1                    Program: NONAME
The Next Generation Language  Status :

                        C r e a t e   R M S   f i l e

Data file        :  CUSTOMER.DAT
File organization:  INDEXED
Record size      :  182
Primary key      :  ID
Key  2           :  COMPANY





Duplicates? yes

                                Choices: Yes, No
EXIT = Exit                                                \ = Back  HELP = Help
```

- **Record Size?** 182

Example 15–14 Record Size Prompt

```
          INTOUCH 4.1                                Program: NONAME
The Next Generation Language              Status :





 Record size? 200

            CUSTOMER.dat  created with record size 200

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

- **Primary Key, field name?** ID

- **Key 2, field name?** COMPANY

Next, you are asked if you want to proceed with file creation. To proceed, enter Y:

Example 15–15 Proceed Prompt

```
INTOUCH 4.1                    Program: NONAME
The Next Generation Language   Status :

                        C r e a t e   R M S   f i l e

Data file         :  CUSTOMER.DAT
File organization:   INDEXED
Record size       :  182
Primary key       :  CONTACT
Key  2            :  COMPANY, duplicates allowed




Proceed with file creation (y/n)? no__

                                   Choices: Yes, No
EXIT = Exit                                              \ = Back  HELP = Help
```

15.8 Security

The SECURITY menu item allows structure security to be assigned.

The security levels assigned within CALL SETUP can be compared to security levels assigned to an individual user. The levels are used to control access.

```
                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____
```

16 Elements of an INTOUCH Program

16.1 General Information

This chapter describes the basic elements which make up INTOUCH programs.

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 "Your name, please": NAME$
20 PRINT "Hello, "; NAME$
   END
```

The INPUT statement tells INTOUCH to ask for a name. The PRINT statement tells INTOUCH to print "Hello, " followed by the name that was input. END tells INTOUCH it has reached the end of the program.

When you execute this program with INTOUCH, it will look like this:

Example 16–1 Executing an INTOUCH program

```
INTOUCH 4.1

                              Program: NONAME
The Next Generation Language            Status :
------------------------------------------------------------------

INTOUCH

10 INPUT "Your name, please": NAME$
20 PRINT "Hello, "; NAME$
END
RUN
NONAME     1-FEB-1985 16:50
Your name, please? John
Hello, John

INTOUCH

------------------------------------------------------------------

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

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.

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 'Your name, please': NAME$
                      20 PRINT 'Hello, '; NAME$
     END statement --    END
```

Each program unit is made up of program lines. A program line consists of one or more statements.

```
20 PRINT 'Hello, '; NAME$
   END
```

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

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

RNH
Your name, please? Suz
Hello, Suz

INTOUCH
```

Notice that line 20 contains two statements. If you include more than one statement on 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  FOR I = 1 TO 10
      PRINT I
    NEXT I
20  PRINT 'Done'

30  END

RNH
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 Done

 INTOUCH
```

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

Example:

```
INTOUCH
FOR Z1 = 4. to 7. STEP .5 \ PRINT Z1 \ NEXT Z1
 4
 4.5
 5
 5.5
 6
 6.5
 7

INTOUCH
```

INTOUCH accepts line numbers in the range of 1 to 99999. Line numbers must be integers.

16.1.1 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. For example:

```
10 INPUT A$
   IF A$<>"" THEN PRINT A$; &
   " is OK."
20 END
```

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

```
10 INPUT A$
   IF A$<>"" THEN PRINT A$; &    ! required ampersand
   &  " is OK."                  ! optional ampersand
20 END
```

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.

16.2 Labels

A label is a way to name a place in a program. Labels must begin with a letter, can be up to 31 characters long, and may contain letters, digits, and underscores.

To set up a label, put the name on a program line and separate it from the following statement with a colon. For example:

```
10 INPUT 'Your name, please': NAME$
20 MAIN: IF _EXIT THEN GOTO 40
30       PRINT 'Hello, '; NAME$
40 END

RNH

Your name, please? EXIT

INTOUCH
```

You can also use a label before multiple statements attached to one line.

```
10 INPUT 'Your name, please': NAME$
   MAIN: IF _EXIT THEN GOTO 30
         PRINT 'Hello, '; NAME$
30 END
```

16.3 Storing INTOUCH Programs

INTOUCH programs have a default of ".INT". You do not have to give an extension with your file name in the INTOUCH commands (SAVE, UNSAVE, OLD, etc.). We recommend that you use this extension for all your INTOUCH programs.

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

16.4 Data in INTOUCH Programs

Programs manipulate data. The data can be numeric or textual, but data must be presented according to the rules of the INTOUCH language. INTOUCH accepts three basic types of data: integer numbers, real numbers and string data.

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

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 may end with a trailing percent sign (%).

- 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.

Real Numbers

A real number may 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**.

- 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 may be anywhere in the range of -1.7E38 to +1.7E38.
- Real numbers have a precision of .29E-38 to 1.7E38.

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"
```

- 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 may contain up to 65535 characters.
- String variables are dynamic in length.

16.4.1 BOOLEAN

Boolean variables (they are either TRUE or FALSE) 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
```

16.4.2 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.

16.4.3 Constants

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

Numeric constants

Integer numeric constants are written with digits, with no decimal fraction. An integer constant may 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.

E Notation

E notation is a shorthand way of representing very large or very small numbers. Real numeric constants may 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 16–1 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.

For instance, if we wanted to represent 1000 in E notation, we would use:

```
+1E3
```

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

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

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
RUN
NONAME       1-FEB-1985 17:10
Account number (##-####)
    Account number     (##-####)

INTOUCH
```

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 double quotes (") or single quotes (’). However, the quotes must match.

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!"
```

16.4.4 Variables

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

- Simple variables. AVERAGE NAME$
- Array elements. AVERAGE(3) 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. NAME$[7:12]
- Structure references. SALES(AVERAGE) 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.
- may be followed by a proper data type designation (% for integers and $ for strings).
- may be declared explicitly.

Some examples of valid variables are:

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

Variables can be integer numeric, real numeric or string. Integer variables may 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.

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 may 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)
```

- Arrays must be defined with the DIM statement and may be redefined with the REDIM statement.
- Arrays may have up to 32 dimensions.
- An array may 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 the section on OPTION BASE.
- You can indicate the low bound explicitly with a TO clause: DIM COSTS(5 TO 10)

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
RUN
NONAME       1-FEB-1985 17:20
IJKLMN

INTOUCH
```

The substring ALPHABET$[9:14] tells INTOUCH to begin at the ninth character.

Next, INTOUCH counts 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:

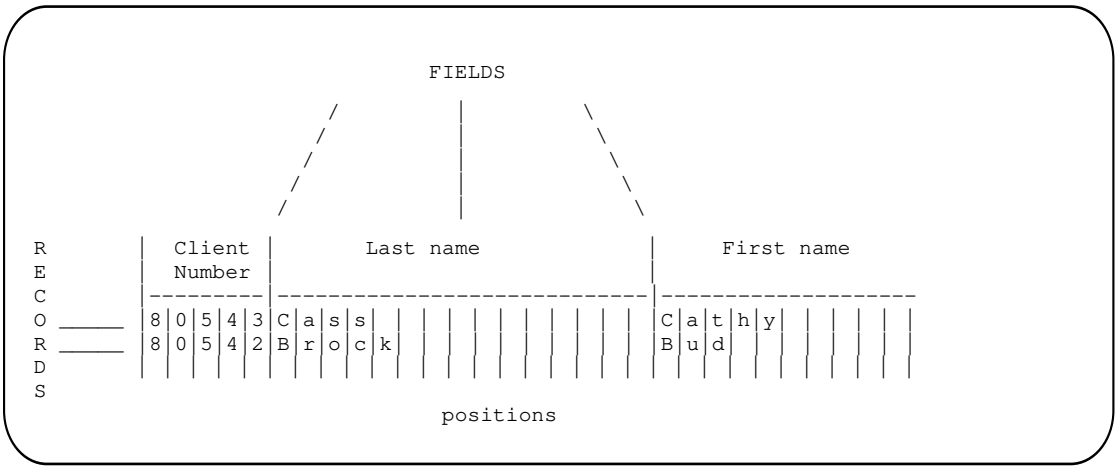
```
10 LET A$ = "Your tests are in."
20 PRINT A$
30 LET A$[6:10] = "results"
40 PRINT A$
50 END
RUN
NONAME     1-FEB-1985 17:21
Your tests are in.
Your results are in.

INTOUCH
```

16.4.5 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.

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

Example 16–2: Structures



Each structure is made up of **records** and **fields**. The records are the horizontal lines of information. 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.

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 may be either a string or numeric expression.

You may 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
```

Following is an example of a program which uses field expressions:

```
10  OPEN STRUCTURE CL:  NAME 'TTI_run:client'
20  DO
      GOSUB ASK_FIELDS
      IF _EXIT THEN EXIT DO
      GOSUB SHOW_DATA
    LOOP
    STOP

30  ASK_FIELDS:
    DO
      LINE INPUT 'Enter fields to display': FIELD_LIST$
      IF  _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:
    EXTRACT STRUCTURE CL
      FOR F = 1 TO ELEMENTS(FIELD_LIST$)
        FIELD$ = ELEMENT$(FIELDS_LIST$, F)
        PRINT CL(#FIELD$),
      NEXT F
      PRINT
60  END EXTRACT
    RETURN
70  END
```

```
RNH
Enter fields to display? LAST,FIRST,PHONE
Smith               Tom                 (619) 455-7404
Jones               Tom                 (   ) 000-0000
Kent                                    (   ) 000-0000
Johnson             Paul                (   ) 890-7898
Rodrigues           Homero              (   ) 000-0000
Donaldson           George              (   ) 000-0000
Errant              Earl                (619) 632-8121
Brock               Bud                 (218) 555-4322
Cass                Cathy               (714) 555-1212
Porter                                  (   ) 000-0000
Derringer           Dale                (818) 555-1212
Farmer              Fred                (305) 552-7872
```

16.4.6 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.

Example:

```
10  OPEN STRUCTURE CUST: NAME "TTI_RUN:CUSTOMER", ACCESS INPUT
20  EXTRACT STRUCTURE CUST
30    PRINT CUST(ADDRESS#1)
40    PRINT CUST(ADDRESS#2)
50  END EXTRACT
60  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.
```

16.4.7 Compound Expressions

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

Conditional expressions
Numeric expressions
String expressions

16.4.8 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 may combine any number of these operators in an expression.

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

You may not generally use two arithmetic operators next to each other. However, you may 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.

String Expressions

String expressions are strings concatenated (joined). String expressions may 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
RUN
NONAME      1-FEB-1985 17:23
MOTHER

INTOUCH
```

In the above example, INTOUCH joins the strings separated by plus signs or ampersands, 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
RUN
NONAME     1-FEB-1985 17:25
THIS IS IT.

INTOUCH
```

16.4.9 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 may 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
RUN
NONAME      1-FEB-1985 17:30
TEST is less than TEXT

INTOUCH
```

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
RUN
NONAME      1-FEB-1985  15:40
 5

INTOUCH
```

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
```

16.4.10 Order of Evaluation

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

```
          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

17 INTOUCH RUNNING UNDER VMS

Every file has a file specification. The file specification is used to access the file. In VAX INTOUCH, a legal file specification looks like this:

```
node::device:[directory]filename.ext;version
```

When you give 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:

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 backspace key was pressed at the last prompt.

_CHANNEL

Returns the next available channel number.

Example:

```
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.

Example:

```
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 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.

Example:

```
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.

Example:

```
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

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.

Example:

```
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$.

Example:

```
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.

Example:

```
10  A$ = 'AB'
    B = CONVERT(A$)
20  END

RNH
16961
```

CONVERT$(int_expr1 [, int_expr2])

Given an integer (int_expr1) and an optional length (int_expr2), which defaults to four, this function returns a string mapping of the integer.

Example:

```
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).

Example:

```
 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
 50  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
100  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.

Example:

```
10 PRINT CPAD$('123', 9, '0')
20 END

RNH
000123000
```

CSC(num_expr)

CSC(x) is the cosecant of X. It's a 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–1 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 letter heading, you can use:

```
PRINT DATE$(DAYS(DATE$),4) gives you

    April 17, 1991
```

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.

Example:

```
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–2 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

Example:

```
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.

Example:

```
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.

Example:

```
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–3 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:

Example:

```
10 LET A$ = ELEMENT$('ADD,DEL,EXIT',2)
20 PRINT A$
30 END

RNH
DEL
```

You may specify a separator other than the comma with str_expr2.

Example:

```
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.

Example:

```
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.

Example:

```
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., FRED.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.

Example:

```
10 PRINT FILESPEC$('x.y', 'ALL')
   PRINT FILESPEC$('', 'ALL')
20 END

RNH
USER:[FRED]X.Y;
USER:[FRED].;
```

Example:

```
10 x$ = 'TTI_RUN:CLIENT'
   PRINT FILESPEC$(x$, 'ALL', '.dat')
20 END

RNH
STORAGE:[INTOUCH]CLIENT.DAT;
```

Example:

```
10 PRINT FILESPEC$('tti_run:client', 'ALL', '.dat')
20 PRINT FILESPEC$('tti_run:client', 'LOCATION')
   PRINT FILESPEC$('tti_run:client', 'LOCATION, NAME')
   PRINT FILESPEC$('tti_run:client.dat;1')
30 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.

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.

Example:

```
10  PRINT findfile$('TTI_RUN:*.INT')

RNH
DUA0:[INTOUCH]ALIENS.INT;5
```

Example:

```
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

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 a '@' 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.

Example:

```
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.

Example:

```
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
```

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–4 FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
0	CCYYMDD HHMMSS
1	MMDDCCYY HHMMSS
2	DDMMCCYY HHMMSS
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.

Examples:

```
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.

Example:

```
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.

Example:

```
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.

Example:

```
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 for a given pattern 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 16 below shows the difference between using '?' and {^}.

```
10 PRINT PATTERN('Mary J. Smith','{^Mar}')
15 PRINT PATTERN('Mary J. Smith','J. {^S}')
16 PRINT PATTERN('Mary J. Smith','J. S?')
20 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$( ).

Example:

```
10 CLEAR
15 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 'Number of lines: '; PIECES(wt$)
60 PRINT WT$
70 PRINT
80 PRINT 'First line was: '; PIECE$(wt$, 1)
100     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.

Example:

```
10  DO
      CLEAR
      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
50    PRINT
      PRINT 'Quoted text using the QUOTE$ function...'
      PRINT quote$(line$)
      DELAY
    LOOP
100 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.

Example:

```
10 INPUT 'Your age': AGE%
20 LET DECIMAL_AGE = REAL(AGE%)
30 PRINT 'Your number is'; DECIMAL_AGE
40 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 =.

Example:

```
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.

Example:

```
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.

Example:

```
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.

Example:

```
10 LET A$ = 'Cameron Whitehorn'
20 LET B$ = 'amr Wtor'
30 LET POSITION = SCAN(A$,B$)
40 PRINT POSITION

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.

Example:

```
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.

Example:

```
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.

Example:

```
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.

Example:

```
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.

Example:

```
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.

Example:

```
10  DO
      CLEAR
      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
50    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 TRUE or FALSE according to the following rules:

- *ALLOW*

```
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*

```
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*

```
MINLENGTH  nn
```

Minimum of nn characters long.

- *MAXLENGTH*

```
MAXLENGTH   nn
```

Maximum of nn characters long.

- *LENGTH*

```
LENGTH nn
```

Exactly nn characters long.

- *CHARACTERS*

```
CHARACTERS   ccc
```

Returns TRUE if the given characters (letters or numbers) are on the 'rule_str' list.

- *NOCHARACTERS*

```
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 lowercase letters.

- *UCASE*

Returns TRUE if given 'text_string' consists of uppercase characters.

- *DIGITS*

Returns TRUE if 'text_str' consists of the numbers 0 - 9.

- *DECIMALS*

Specifies the maximum number of decimal places.

```
DECIMALS nn
```

- *NUMBER*

Indicates that 'text_str' is numeric (validates that number).

- *INTEGER*

Returns TRUE if given integer number is 32 bits long.

- *INTEGER WORD*

Returns TRUE if given integer number is 16 bits long.

- *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 PATTERN$( ).

- *REQUIRED*

Returns TRUE if 'text_str' does not contain any characters or spaces.

- *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 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 uppercase characters.

LCASE$

Convert to lowercase characters.

REPLACE

```
old_t1=new_t1, old_t2=new_t2, ....
```

If 'text_str' contains a few 'old_t1' characters, all of them will be replaced by 'new_t1'.

REMOVE

Removes any characters from the 'text_str'.

CHANGE

Changes specified characters. For details see CHANGE$( ).

```
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
```

VALID returns an error if there is an invalid validation rule.

```
"Illegal validation rule" (-4021)
```

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 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.

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

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 Conventions Used in this Manual

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 must supply. These conventions are always shown in lower case. The conventions are listed below.

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.

Conventions Used in this Manual

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.
[ ]	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.

.
.
.

F Making Libraries for Use with INTOUCH

To make libraries for use with INTOUCH, you will need to follow the steps described below:

1 Write the source code using an editor. The source code may 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
```

2 Compile the source code. For instance, the above BASIC routine would have to be compiled with the BASIC compiler:

```
$ BASIC MULT
```

3 Write an options file.

Use an editor to create an options file. Give the options file a file extension of ".OPT". 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
```

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 can now use this routine in INTOUCH. To use the routine, first name the library with a LIBRARY statement, supplying the complete file specification, INCLUDING DISK 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%
```

G INTOUCH DATABASE INTERFACES

Overview

Multiple VAX database engines can be used with INTOUCH. This chapter explains the interfaces between these databases and INTOUCH. Information on how to use these interfaces is included. The following interfaces are discussed:

- ADABAS
- DBMS
- Rdb
- dBASEIII+
- CDD

Also discussed is the correspondence between INTOUCH and the database field attributes, and limitations of the interfaces to INTOUCH.

G.1 ADABAS

ADABAS is a database engine. ADABAS data is contained in a database, which is a collection of files.

This section explains the interface between ADABAS and INTOUCH. The section describes how to use the ADABAS interface, the correspondence between INTOUCH and ADABAS field attributes, and limitations of the ADABAS interface to INTOUCH.

G.1.1 SETUP

There is no explicit installation required for the ADABAS interface. If your system has ADABAS installed, the ADABAS interface will be activated on access.

G.1.2 Limitations

This implementation of the ADABAS interface supports multi-value and periodic group fields with up to 99 occurrences.

Phonetic, subdescriptor and superdescriptor fields are not supported.

INTOUCH cannot create an ADABAS database or the internal ADABAS files.

G.1.3 An example of Setting Up and Using the ADABAS Interface

When you set up the structure for an ADABAS table with CALL SETUP, the data file name indicates both the file and the database. The data file name must not contain either a disk or directory specification.

In defining an ADABAS file to INTOUCH, you have the option of using either the ADABAS FDT or defining the file using an INTOUCH definition.

Here is a sample dialog setting up a structure and using the ADABAS FDT:

```
Structure name? DB_1_FILE_12
Data file? 12 IN 1
Record system? ADABAS
Definition file? NONE
```

Here is a sample dialog setting up a structure and using INTOUCH to define the fields:

```
Structure name? DB_1_FILE_12
Data file? 12 IN 1
Record system? ADABAS
Definition file? DB_1_FILE_12.DEF
```

When defining the fields through INTOUCH, the ADABAS interface uses the starting position of the field to determine which ADABAS FDT entry is referred to.

G.1.4 Referencing ADABAS Multi-value and Periodic Group Fields

An extension to the INTOUCH structure(field) syntax has been provided to allow access to multi-value and periodic group fields.

```
Z$ = STR1(AC#5)  ! get 5th value for field AC

Z$ = STR1(AF#1#2) ! get the 2nd value for the 1st group of
                    field AF
```

G.1.5 Sample INTOUCH Program Accessing ADABAS

```
! INTOUCH program accessing ADABAS
10   open structure adabas_db : name DB_1_FILE_12
     extract structure adabas_db
       print adabas_db(aa); tab(18); &
         adabas_db(ab); tab(35); &
         adabas_db(am#5)
     end extract
     close all
     end
```

G.1.6 NATURAL Interface

Data definitions created by Natural and Predict can be accessed from INTOUCH. Software AG does not provide a supported method of directly accessing the Natural/Predict data definitions. Because of this, we have written an INTOUCH program to access Natural, extract the data definition and write it to an INTOUCH augmented definition file.

Using an augmented definition file allows a programmer to access both the ADABAS two-character field names and the extended field names that were defined in Natural/Predict. When using the extended fields, the heading and edit mask defined in Natural/Predict are available.

To run the INTOUCH program to create the augmented definition file:

```
$ INTOUCH/SOURCE TTI_RUN:NATURAL_DEF
```

This program asks the following questions:

- Natural data definition filename: FILE-9 where FILE-9 is the name of the data definition to extract.
- Intouch definition name : FILE_9.DEF where FILE_9.DEF is the name of the definition file to create

The program creates a sub-process to run Natural. The sub-process logs onto the SYSDDM library, reads the definition and writes the definition to a temporary disk file. The temporary file is read and each field definition is written to the INTOUCH definition file.

This procedure needs to be performed on each Natural/Predict data definition that is to be accessed from INTOUCH.

G.2 DBMS

Note

For information on the DBMS interface, please refer to the INTOUCH manual, *INTOUCH Interface to DEC DBMS DATABASE ENGINE.*

G.3 Rdb

Rdb is a database engine. Rdb data is contained in a database, which is a collection of tables or relations.

G.3.1 SETUP

There is no explicit installation required for the Rdb interface. If your system has Rdb/VMS installed, the Rdb interface will be activated on access.

G.3.2 Overview

When a record is known to INTOUCH to be current, a transaction is outstanding. When another record is retrieved, that transaction is committed, and another one is started (except in an EXTRACT/END EXTRACT loop [see below]).

The transaction type is either READ_ONLY or READ_WRITE, depending on whether the structure was opened ACCESS INPUT or ACCESS OUTIN.

You may want to add a record to a relation that has another record active. The record is not added until the transaction is committed (usually by getting another record, or closing the relation).

In CALL SETUP, a definition file of NONE is valid for Rdb files. This will retrieve as much of the Rdb metadata as possible, and synthesize much of the rest. (The starting position is calculated as the end position+1 of the previous field, etc.)

G.3.3 Limitations

In the EXTRACT/END EXTRACT loop, a single transaction is used to get the dbkeys for all the records. These records are all locked until the end of the extract. Explicit unlocking is strongly discouraged, as it destroys the record stream, and unpredictable results occur.

You may want to close a structure on a database. Another structure may be open on the same database with a record active (with a transaction outstanding). The transaction isn't committed (and the new record isn't visible to other users) until the modified record is released. It can be released by closing, unlocking, finding another record, etc.

Segmented keys are supported. For any key that uses multiple fields, a single field must be defined that starts at the starting location of the first key and has the length of the entire key (even though this might not map against the data properly).

In RDB, you must use an augmented definition file to create this new field.

Segmented strings are not supported at this time.

When unlocking, the transaction is committed. Transactions are DATABASE wide, so all other records locked on the same database attach (and all the streams) are freed. Use of a different phrasing of the databasename in the structure (or in the DATAFILE clause of the open) causes the database to be attached to twice. This preserves the second structures locks (i.e., to be able to unlock the structure STRA and keep locks or a stream on STRB, define a logical to the second database, and use the logical to open the database.)

Varying text strings (defined by INTOUCH are AC) are read-only. If there is an overlay within INTOUCH of the field by a word/text combination, and the user controls the length argument manually, they may write to varying text strings. I.e.,

```
|VT string      |   <-- Rdb declared
|ct|text        |   <-- INTOUCH declared (read write)
|VT string      |   <-- INTOUCH declared (read only)
```

Locking a record against write requires writing to the record. This is a behavior of Rdb.

G.3.4 Performance Topics

Both OPEN and RELATE use callable RDO for parts of their operations, and so are slower than the rest of the structure access functions.

G.3.5 Sample Description and Use of a Database

G.3.6 Setting Up the Structure

The syntax of the datafile field in SETUP is "relation IN databasename", where relation is the relation name, and databasenname is the physical database name.

INTOUCH keeps a list of databases attached to, and will not physically attach to the database if the names in the databasename clauses are the same. It does not translate any logicals. To attach to two different relations in the same database with two physical attaches, you can use a logical for the database name. INTOUCH treats them as two separate databases.

Here is a sample dialog setting up a structure:

```
Structure name? EMPLOYEE
Data file? EMPLOYEES IN PERSONNEL
Record system? Rdb
Definition file? NONE
```

G.3.7 Sample Rdb Table Definition

Here is a sample definition of an Rdb table. You can get this information interactively at the RDO prompt by typing SHOW FIELDS FOR EMPLOYEE on the standard PERSONNEL demo database supplied with Rdb/VMS.

```
RDO> show fields for employees
 Fields for relation  EMPLOYEES
     EMPLOYEE_ID                     text size is  5
       based on global field  ID_NUMBER
     LAST_NAME                       text size is  14
     FIRST_NAME                      text size is  10
     MIDDLE_INITIAL                  text size is  1
     ADDRESS_DATA_1                  text size is  25
     ADDRESS_DATA_2                  text size is  25
     CITY                            text size is  20
     STATE                           text size is  2
     POSTAL_CODE                     text size is  5
     SEX                             text size is  1
     BIRTHDAY                        Date
       based on global field  STANDARD_DATE
     STATUS_CODE                     text size is  1
```

G.3.8 Sample INTOUCH Program Accessing Rdb

```
! INTOUCH program accessing Rdb
open structure x : name 'employee'
extract structure x
  print x(employee_id),x(first_name),x(last_name)
end extract
```

```
(MAINTAIN, GUIDE, etc., can also be used on this structure)
```

G.4 dBASEIII+

dBASEIII+ is a database engine. dBASEIII+ data is contained in a file with a header, the field information, and the actual data.

This section explains the interface between dBASEIII+ and INTOUCH. The manual describes how to use the dBASEIII+ interface, the correspondence between INTOUCH and dBASEIII+ field attributes, and limitations of the dBASEIII+ interface to INTOUCH.

G.4.1 SETUP

There is no installation required for the dBASEIII+ interface.

G.4.2 Overview

dBASEIII+ is a flat file system, with field information stored within the data file.

G.4.3 Limitations

There is no key access for dBASEIII+ files.

It should be noted that INTOUCH deletes records EXACTLY as dBASEIII+ does, by flagging the prefix byte with a "*", and all adds are done at the end of the file. Repeated adding and deletion of data should be avoided.

INTOUCH does not support Memo files. Memo fields are marked as read-only numeric fields.

Currently, one user may have write access to the datafile. Due to the limitations of RMS block mode I/O, no other users may have access.

G.4.4 dBASEIII+ datatypes and INTOUCH attributes

With dBASEIII+, you may specify NONE at the definition file, or use and define an INTOUCH definition file.

When specifying NONE for the definition file, the native datatype is converted to the closest match to INTOUCH attributes. INTOUCH always defines the datatype as CH, and the field attributes as listed below.

The field name is used as the heading, prompt, and description. If there is a scale on numeric fields, a print mask is generated of the format "######.##", with the total field length being represented, and the number of digits to the right of the period is the number of decimal places stored in the dBASEIII+ file.

dBASEIII+ datatype	INTOUCH field attributes
Character	None
Numeric	NUM:nn, RJ
Logical	None
Date	Date:YMD
Memo	NUM, RJ

1 nn is the scale provided from the DBASE3 file.

2 Memo fields are 10 digit numeric fields. INTOUCH will not allow write access to these fields.

G.5 CDD

CDD/Plus is a general-purpose metadata definition product that allows users to define fields with various attributes (prompt, length, datatype, etc.), and records (which are collections of fields).

The INTOUCH CDD/Plus interface allows people with existing records defined by CDD/Plus to use them in place of the standard INTOUCH definition file.

G.5.1 SETUP

If your system has CDD/Plus installed, the CDD/Plus interface will be activated on access.

In SETUP, the definition file is the CDD/Plus directory and record name prefixed with a "CDD>"

(i.e., "CDD>DEVDISK:[DEVINT.CDDTEST]CUSTOMER")

G.5.2 Overview

CDD/Plus records are the logical equivalent of INTOUCH definition files. The main difference is that CDD/Plus allows nested records (records within records), and they are stored in an abstract form, rather than easily identified discrete files.

G.5.3 Attributes

Most INTOUCH field attributes have directly corresponding CDD/Plus attributes, with the following exceptions:

- The starting position is calculated as the end of the previous field.
- The scale has its sign changed (negative to positive) when retrieved.
- The help attribute takes the second non-blank line from the CDD/Plus description attribute.
- The print mask is converted from the CDD/Plus format as explained below.

- Field access is granted to everyone. It is calculated at open-time as being equal to whatever the structures access value is.

G.5.4 Edit Mask Conversions

The following is the CDD/Plus CDO edit string defining character followed by the INTOUCH print mask character it translates to:

9	@
,	,
+	+
-	-
"x"	~x
$	$
X	#

All other CDD/Plus edit string tokens will be converted to "#".

The first instance of this edit string token will be replaced by the specified text. Subsequent instances will be replaced by a "#".

Each character in the literal is quoted with a tilde.

G.5.5 Datatype Conversions

The following is a list of datatypes that are able to be converted to INTOUCH datatypes. All other datatypes are assigned the datatype of UN (undefined)

cdd datatype	INTOUCH datatype
dsc$k_dtype_t	ch
dsc$k_dtype_vt	ac
dsc$k_dtype_adt	ds
dsc$k_dtype_b	in
dsc$k_dtype_w	in
dsc$k_dtype_l	in
dsc$k_dtype_f	fl
dsc$k_dtype_d	fl
dsc$k_dtype_bu	iu
dsc$k_dtype_wu	iu
dsc$k_dtype_lu	iu
dsc$k_dtype_p	c3
dsc$k_dtype_nro	ro
dsc$k_dtype_q	qs

G.5.6 Limitations

CDD/Plus allows duplicate field names. INTOUCH allows these fields to be read in, but accessing the fields by field name uses only the first instance of the field name. To access the subsequent instances of the fields, access by field number ("STR(#nn)") would be required.

The CDD/Plus path name to the record definition may frequently be over 64 characters (the maximum allowed in SETUP). This can be worked around by assigning a logical to the directory and record path, and defining the definition file by "CDD>dirpath:recordname"

Based on fields are not supported in this release. This will be fixed in a future release.

G.5.7 Sample Description and Use of a CDD/Plus Record

This section briefly shows a CDD/Plus DICTIONARY

G.5.8 Sample CDD/Plus Record Definition

The following is a series of CDO commands to define a set of fields, sub-records, and a CUSTOMER record. The CDO command-line interface does not support multiple lines, so you must go into CDO, and edit the field descriptions of CUSTNBR and NAME, and replace the "<CR>"'s with physical new-lines.

```
define field custnbr description is /*Customer
Number<CR>Next line
of CUSTNBR description*/
 datatype is text
 size is 5
 query_name is "Customer number"
 query_header is "Customer,number"
 edit_string is 99"-"999.

define field name
 description is /*Customer name<CR><CR>Next line of NAME description*/
 datatype is text size is 25  query_name is "Customer name".

define field addr1 description is /*Address line 1*/ datatype is text
size is 25 query_name is "Address line 1".


define field addr2 description is /*Address line 2*/ datatype is text
size is 25 query_name is "Address line 2".

define field city description is /*City*/ datatype is text size is
20 query_name is "City".

define field state description is /*State*/ datatype is text size is
2 query_name is "State".

define field zip description is /*Zip*/ datatype is text size is 5
query_name is "ZIP (post) code".

define field balance description is /*Balance due*/  datatype is text
size is 8 query_name is "Balance due".

define field contact description is /*Contact name*/ datatype is text
size is 20 query_name is "Contact name".

define field phone description is /*Phone number*/ edit_string is
AAA")"AAA"-"AAAA  datatype is text size is 10 query_name is
"Phone number".
```

```
define record address.
addr1.
addr2.
city.
state.
zip.
end address record.
define record customer.
custnbr.
name.
address.
balance.
contact.
phone.
end customer record.
```

G.5.9 Setting Up the Structure

The following is an actual screen showing SETUP for an RMS file using a CDD/Plus description:

Example G–1 Setup

```
INTOUCH 4.1 Program: NONAME
The Next Generation Language                Status :
------------------------------------------------------------------------
              S e t   u p   D a t a   S t r u c t u r e s

Structure name? TEST_CDD_____________________________________________
Data file? TTI_RUN:CUSTOMER.DAT______________________________________
Record system? RMS_______________
Definition file? CDD>DEVDISK:[DEVINT.INTEST_CDD]CUSTOMER_____________

Procedure? define__

                  Procedures:  define, show, create, exit

EXIT=Exit                                    \=BACK    HELP=Help
```

G.5.10 Sample INTOUCH Program Using the CDD Definitions

MAINTAIN may be used to modify this structure, which is based on the CUSTOMER example structure shipped with INTOUCH.

Index

\ • 5–20
#
  In Structure References • 16–13
* • A–20
? • A–20
{^} • A–21
{} • A–20
~ • A–21

A

Abbreviations • 1–4
ABORT • 2–8
ABS • A–3
ACCESS
  Option to OPEN STRUCTURE Statement • 13–4
ACCESSING Rdb
  Sample INTOUCH Program • G–6
ACCESS INPUT
  In files • 6–2
  In OPEN STRUCTURE • 13–4
ACCESS Option • 6–2
ACCESS OPTION
  In files • 6–2
ACCESS OUTIN
  In files • 6–2
  In OPEN STRUCTURE • 13–4
ACOS • A–3
ADABAS • G–1
  CALL SETUP • G–2
  Data Definitions • G–3
  Limitations • G–1
  Sample INTOUCH Program • G–2
  Setup • G–1
ADABAS FDT • G–2
ADDITION • 16–15
ADD STRUCTURE Statement • 13–43
ALPHA • 2–20
Ambiguous constants • 16–6
AMPERSAND (&)
  See STATEMENTS, Continuing • 16–4
AND • 16–17
ANGLE • A–4
APPLICATION • 13–26
Application development tool • xxi
AREA
  Line Input • 5–4
AREA Option • 5–13, 7–18
ARITHMETIC OPERATORS
  Description of • 16–15
  Logical • 16–17
  Relational • 16–16, 16–17
ARRAY
  Defining arrays • 8–1
  Defining with DIM • 8–2
  Defining with DIM TO option • 8–3
  Description of • 16–10
  Dimensions • 8–1, 16–10
  Elements • 16–10
  High bound • 8–2
  High bound–Range • 8–1
  Low bound • 8–2
  Low Bound • 8–3
  Low bound–Range • 8–1
  Multiple Dimensions • 8–3
  Range of dimensions • 16–10
  Redimensioning with REDIM • 8–4
  Subscripts • 16–10
ASCII • A–4
ASCII CHARACTER SET • D–1
ASIN • A–4
ASK • 6–8, 9–1
  About a device • 6–8
  options
    CURRENT • 6–9
    MARGIN • 6–9
  Options • 6–9
    ZONEWIDTH • 6–9
ASK #chnl_num: NAME • 6–10
ASK ERRORS • 9–5
ASK KEYSTROKES • 9–7
ASK MARGIN • 9–8
ASK MESSAGELINE • 9–9
ASK RESPONSES • 9–11
ASK SEED • 9–12
ASK Statement • 6–8
ASK STRUCTURE • 13–40
  CURRENT Option • 13–34
  FIELD expressions • 13–32
  FIELD items • 13–25
ASK STRUCTURE:
  FIELDS Option • 13–35
ASK STRUCTURE: ACCESS • 13–39

ASK STRUCTURE: CAPABILITY • 13–35
ASK STRUCTURE: ENGINE • 13–41
ASK STRUCTURE: ID • 13–36, 13–37
ASK STRUCTURE: POINTER • 13–37
ASK STRUCTURE: RECORD • 13–38
ASK STRUCTURE: RECORDSIZE • 13–39
ASK STRUCTURE:TIMEOUT • 13–40
ASK STRUCTURE Statement • 13–36
ASK STRUCTURE struc_name, FIELD field_name: ACCESS • 13–25
ASK SYSTEM, LOGICAL • 9–16
ASK SYSTEM, PASS: SUCCESS • 9–18
ASK SYSTEM, SYMBOL • 9–20
ASK SYSTEM: COMMENT • 9–13
ASK SYSTEM: DIRECTORY • 9–15
ASK SYSTEM: MODE • 9–17
ASK SYSTEM: PARAMETER • 9–17
ASK SYSTEM: PROCESS • 9–18
ASK SYSTEM: RIGHTS • 9–19
ASK SYSTEM: USER • 9–21
ASK WINDOW • 9–23
ASK WINDOW: COLUMN • 9–22
ASK WINDOW: DATA • 9–25
ASK WINDOW: KEYMAP • 9–26
ASK WINDOW: ROW • 9–29
ASK WINDOW: TYPEAHEAD • 9–30
ASK WINDOW AREA • 9–21
ASK ZONEWIDTH • 9–32
ASSIGNING VALUES
  To structure fields • 13–43
  To substrings • 16–11
ASSIGNING VARIABLES • 4–1
  LSET, RSET and CSET • 4–8
  Numeric values • 4–7
  See also LET Statement • 4–7
  String values • 4–7
AT • 5–18
  As a print option • 7–3
ATN • A–4
AT Option • 5–8
AT OPTION
  In the screen management facility • 5–19
%AT row, column • 5–22
Attributes
  CLEAR AREA • 7–19
ATTRIBUTES • 13–26, 13–27
  Field • 15–8
ATTRIBUTES Option • 5–9
%AUTODISPLAY • 5–28
AUTOJUMP
  See INPUT FROM A TERMINAL
AUTOJUMP
  See INPUT FROM A TERMINAL (cont'd)
    SCREEN Option
      AJ • 5–19
%AUTOVBAR • 5–27
AXP systems • 2–20

B

_BACK • 2–25, 5–4, A–1
BACKING UP • 5–4
BACKSPACE KEY
  In the screen management facility • 5–20
%bar • 5–22
BIT MANIPULATION
  Description of • 16–18
  Example of • 16–18
BITS
  ON/OFF • 16–18
BLOCK OF CODE • E–1
BOOLEAN • 16–7
BOOLEAN OPERATORS • 16–17
BOX OPTION • 7–19
Braces • E–2
BRANCHING • 10–26
  GOSUB • 10–29
  GOTO • 10–26
  ON GOSUB • 10–30
  ON GOTO • 10–27
  Within code blocks • 10–34
BREAK • 14–8, 14–9
BROADCAST TRAPPING • 7–20
BUILT-IN FUNCTIONS • A–1

C

CALL
  Description of • 12–3
  PASSING ARGUMENTS IN
    By Descriptor • 12–4
    By Reference • 12–4
    By Value • 12–4
  PASSING ARGUMENTS WITH • 12–3
  PASSING MECHANISMS • 12–4
  PASSING OPTIONAL ARGUMENTS IN • 12–3
  See also the _REAL and _INTEGER functions • A–1

CALLING ROUTINES • 12–1
CALL SETUP • 15–2
  SECURITY • 15–17
  SHOW STRUCTURE • 15–12
CANCEL ADD Statement • 13–43
CANCEL EXTRACT Statement • 13–9
CASE ELSE • 10–21
CASE Statement • 10–20
CAUSE EXCEPTION • 11–1
CDD • G–1, G–7
  Datatype Conversions • G–8
  Defining fields • G–7
  Duplicate Field Names • G–9
  EDIT MASK CONVERSIONS • G–8
  Field Attributes • G–7
  Setup • G–7
CDD/Plus Records • G–7
CDD Setup • G–10
CEIL • A–4
CHAIN • 10–24
CHAIN Statement • 10–23
CHANGE • A–35
CHANGE$ • A–4
CHANGEABLE • 13–27
_CHANNEL • A–1
CHANNEL NUMBERS
  Assigning • 6–2
CHARACTERS • A–31
CHARSET$ • A–5
CHR$ • A–5
CLASSIFICATION • 13–28
CLEAR • 7–15, 7–18
  AREA Option • 7–18
CLOSE • 6–4
CLOSE Statement • 6–4
CLOSE STRUCTURE • 13–4
CLOSE STRUCTURE Statement • 13–5
CODE • A–33
CODE BLOCKS
  Transfer of control • 10–34
Code change and continue
  Example of • 1–10
CODE CHANGE AND CONTINUE
  Interrupting execution
    HALT • 14–20
Column • 7–18
%COLUMNS • 5–25
Command Completion • 1–3
Command recall • 1–2
Commands
  EXIT and HELP • 1–9
COMMANDS
  Basic description of • 16–2
  Description of • 2–2
COMMENTS
  Description of • 14–13
  Example of • 14–13
  Exclamation point • 14–15
  REM • 14–14
  The tilde • 14–16
COMPARE • A–6
COMPARING STRINGS
  Using relational operators • 16–17
  With relational operators • 16–17
COMPILE • 2–19
COMPILED
  Images • 2–19
COMPILED IMAGES • 2–19
COMPILE ERRORS • C–1
COMPOUND EXPRESSIONS
  Description of • 16–15
CONCATENATION
  See STRING DATA, Concatenation • 16–15
CONCEPTS
  Supported • 9–29
CONDITIONAL EXPRESSIONS
  In DO loops • 10–9
CONDITIONALS • 10–14
CONSTANTS
  Integer numeric
    Description of • 16–7
  Real numeric
    Description of • 16–7
CONTINUE • 11–6
CONTINUE Statement • 11–6
Conventions
  Summary list • E–1
CONVERT • A–5
CONVERT$ • A–5
Correcting errors
  Example of • 1–12
COS • A–6
COSH • A–6
COT • A–6
CPAD$ • A–7
CSC • A–7
CSET • 4–10
CTRL/C
  Breaking out of loops • 10–8
  Description of • 1–9
  Example of • 1–9

CTRL/xxx • 5–29
CTRL/Z • 1–9
CURRENT • 6–9, 6–12

D

DATA
  DESCRIPTION OF • 4–11
Database routines
  Specifications • xxi
Data Structure Manual • xxi
DATA STRUCTURE Statements • 13–9
  ACCESS options • 13–4
  CLOSE STRUCTURE • 13–5
  Description • 13–1
  extract CONSTRUCT
    PARTIAL KEY Lookup • 13–16
  EXTRACT CONSTRUCT • 13–9
DATA STRUCTURE STATEMENTS
  ADD STRUCTURE • 13–43
  DELETE STRUCTURE • 13–42
  FOR EACH • 13–17
  LOCK/UNLOCK STRUCTURE • 13–45
  OPEN STRUCTURE • 13–3
DATATYPE • 13–28
DATA TYPES
  Basic description of • 16–5
DATE • A–7, A–32
DATE$ • A–7
DATE DMONCY • A–32
DATE DMONY • A–32
DAY$ • A–9
DAYS • A–8
dBASEIII+ • G–1, G–6
DBMS • G–1, G–3
DCL Pass thru
  Example of • 1–1
  How to use • 1–1
%DEBUG • 2–24
_DEBUG • A–1
DEBUG OFF • 14–2
DEBUG ON/OFF • 14–1, 14–2
DEBUG SYSTEM
  BREAK • 14–9
  DEBUG ON/OFF • 14–2
  Description • 14–1
  Enabling debug • 14–1
  Monitoring program execution • 14–6, 14–7
    STATISTICS • 14–3
DEBUG SYSTEM
  Monitoring program execution (cont'd)
    TRACE • 14–3, 14–4
    TRACE OFF • 14–5
  STATISTICS
    STATS OFF • 14–7
    STATS ON • 14–6
  STEP • 14–10
DECIMALS • A–32
DECLARE • 4–1
  OPTION REQUIRE DECLARE • 4–5
  Variables • 4–2
DECLARE DYNAMIC • 4–2
DECLARE PREFIX • 4–3
DECLARE STRUCTURE • 4–2
DECODE • A–9
DEFAULT • 5–10
DEFAULT OPTION • 5–10
DEG • A–9
DEGREE • A–9
DELAY • 2–24
DELETE
  FILE
    See UNSAVE • 2–22
DELETE STRUCTURE • 13–42
Delimiters
  See string data, delimiters • 16–8
DESCRIPTION • 13–28
DESCRIPTOR • 12–4
Development cycle time • xxi
DIGITS • A–32
DIM • 8–2
DIMENSIONS
  See ARRAY
    Dimensions • 16–10
DIRECTIVES • 5–22
DISALLOW • A–31
DISPATCH • 10–32
DIV0 • A–9
DIVISION • 16–15
DOLLAR SIGN
  See STRING VARIABLES
    Designation of • 16–10
DO LOOP
  EXIT DO • 10–11
  ITERATE DO • 10–13
  REPEAT DO • 10–12
DO LOOPS
  Conditional • 10–9
  Nested loops • 10–8
  UNTIL option • 10–9
  When to use • 10–1

DO LOOPS (cont'd)
WHILE option • 10–9
DOUBLE COMMAS
CALL Statement • 12–4
DO WHILE Statement • 10–8
DTYPE • A–10
DYNAMIC VARIABLES • 4–7

E

EDIT • 2–14
EDIT$ • A–10
Editing Input • 5–5
EDIT MASK
See INPUT FROM A TERMINAL, SCREEN Option, format • 5–14
Edit string • A–10
ELEMENT$ • A–11
ELEMENTS • A–10
ELSE OPTION
In ON GOSUB • 10–31
In ON GOTO • 10–28
ENABLING DEBUG • 14–1
DEBUG ON/OFF • 14–2
ENCODE$ • A–11
END • 14–16, 14–18
END ADD • 13–43
END EXTRACT • 13–9, 13–10
END HANDLER • 11–4
END Statement • 14–17
END WHEN
In WHEN EXCEPTION IN • 11–3
In WHEN EXCEPTION USE • 11–4
E NOTATION
Description of • 16–7
Printing numbers in • 7–5
ENVIRONMENT
Entering into • 1–1
Examples of • 1–5
Exiting • 1–9
EPS • A–11
EQUAL TO • 16–16
EQV • 16–17
ERASE
As a print option • 7–4
ERASE Option • 5–11
_ERROR • 9–4
ERROR OPTION
See MESSAGE • 7–16
ERRORS
Description of • C–1
List of • C–1
EVAL • A–11
EXAMPLE • 2–1
Trying program examples • 2–1
Exception
line • A–12
EXCEPTION HANDLING • 11–1
CONTINUE • 11–6
EXIT HANDLER • 11–8
HANDLER • 11–4
RETRY • 11–5
See also MESSAGE • 7–15
WHEN EXCEPTION IN • 11–3
WHEN EXCEPTION USE • 11–4
EXCEPTIONS
Description of • C–4
List of • C–4
EXCLAMATION POINT • 14–15
EXCLUDE • 13–9
EXCLUDE Statement • 13–14
EXECUTE • 10–13
Execution of programs
See RUN • 2–5
EXIT • 1–9, 15–13
_EXIT • 2–25, 5–4, A–1
EXIT ADD • 13–44
EXIT COMMAND
Description of • 1–9
EXIT DO • 10–11
EXIT DO Statement • 10–11
EXIT EXTRACT • 13–10
EXIT FOR • 10–5
EXIT for Statement • 10–4
EXIT HANDLER • 11–8
EXITING
FOR loops • 10–5
From DO loops • 10–11
EXIT ROUTINE • 10–33
EXLABEL$ • A–12
EXLINE • A–12
EXP • A–12
EXPONENTIATION • 16–15
"Expr" • 7–1
EXPRESSION • A–34
Conditional
numeric • 16–16
Result of • 16–16
Order of evaluation • 16–19

EXPRESSIONS
Conditional
Description of • 16–16
TRUE/FALSE Values • 16–16
Mixing data types • 16–15
Operating on • 16–15
String • 16–15
Unconditional
Description of • 16–15
EXTERNAL ROUTINES
Calling • 12–3
EXTEXT$ • A–12
EXTRACT • 13–5
EXTRACT CONSTRUCT
Detailed description • 13–9
PARTIAL KEY Lookup • 13–8
_EXTRACTED • A–1
EXTRACT STRUCTURE, SET, FIELD: KEY • 13–13
EXTRACT STRUCTURE, SET, USING • 13–12
EXTRACT STRUCTURE: APPEND • 13–11
EXTYPE • A–12

F

FALSE
As a function • A–12
Definition of • 16–17
In the UNTIL option • 10–9
In the WHILE option • 10–9
See EXPRESSIONS
Conditional • 16–16
Field Expressions • 16–14
FIELD EXPRESSIONS
In Structure References • 16–12
Using variables and expressions • 16–13
FIELDS
Description of • 16–12
Multiple Occurrence • 16–14
Rdb Table • G–5
Variable Length • 15–8
FILES • 6–1
ACCESS INPUT • 6–2
ACCESS Option • 6–2
ACCESS OUTIN • 6–2
CLOSE • 6–4
Closing • 6–4
INPUT # • 6–5
LINE INPUT # • 6–7
Opening • 6–2
FILES (cont'd)
PRINT # • 7–14
FILESPEC$ • A–12
FILE SPECIFICATIONS • 17–1
Extensions
Default • 17–1
.INT • 16–5, 17–1
in INTOUCH statements and commands • 17–1
FILL-IN AREAS • 5–14
FILTER • A–34
FINDFILE$ • A–13
FOR EACH • 13–17
FOR LOOP • 10–1
EXIT FOR • 10–5
ITERATE FOR • 10–7
Nested loops • 10–4
REPEAT FOR • 10–6
FOR LOOPS • 10–2
Index variable • 10–2
Initial expression • 10–2
Limit expression • 10–2
When to use • 10–1
FORMAT • 2–1
FORMAT$ • A–14
FORMAT CHARACTERS • 5–15
• 5–15
@ • 5–15
^ • 5–16
FORMAT OPTIONS
AJ • 5–19
AT • 5–19
DIGITS • 5–16
LCASE • 5–16
REQ • 5–19
UCASE • 5–16
Use of • 5–16
FOR NEXT
STEP option • 10–3
Fourth-generation database language • xxi
FP • A–15
FRAME
Description of • 1–1
Displaying messages in • 7–16
MESSAGE • 7–15
EXIT and HELP • 1–9
FRAME ON/OFF • 7–15
See also FRAME ON/OFF • 2–16
FRAME ON/OFF • 2–16
FULLTIME • A–33
FULLTIME$ • A–15
FUNCTIONS
Built-in • A–1

FUNCTIONS (cont'd)
System • A–1

G

GENERAL INFORMATION • 13–1
GO • 2–21, 14–11, 14–12
GOSUB • 10–29
GOTO • 10–26
GREATER THAN • 16–16
GREATER THAN OR EQUAL TO • 16–16
Guide to the INTOUCH Language
Conventions • E–1
Intended audience • xxii
Purpose • xxi

H

HALT • 14–16, 14–20
HANDLER • 11–4
HASH$ • A–15
HEADING • 13–29
%HEADING • 5–24
HELP • 13–29
_HELP • 2–25, A–2
HELP FACILITY
Description of • 1–9
Example of • 1–7
HIGH BOUND • 8–2
Range • 8–1
HIGHLIGHTING
In fill-in areas • 5–14
See TILDE • 14–16

I

ID
Ask Structure • 13–36
IDE • 2–18
IF Construct • 10–16
Block construct • 10–16
ELSE option
Execution of • 10–17
Execution of • 10–17
ELSEIF option • 10–18
ELSE option • 10–15
IF Construct
ELSE option (cont'd)
Execution of • 10–16
Execution of • 10–16
When to use • 10–14
IMAGES
Compiled • 2–19
IMAGES, Compiled • 2–19
INCLUDE • 2–23, 3–2, 13–9
%INCLUDE • 2–23, 3–2
%INCLUDE CONDITIONAL • 2–24
INCLUDE Statement • 13–13
INCREMENTATION • 10–3
Indenting • 1–3
INDEXED • 13–35
INDEX VARIABLE • 10–2
INITIAL EXPRESSION • 10–2
INPUT • 5–4, 6–5, 13–35
See also INPUT MENU
See also LINE INPUT
INPUT # • 6–5
INPUT AREA • 5–4
INPUT chnl_num, EOF • 6–7
INPUT FROM A SCREEN
SCREEN Option # • 5–15
INPUT FROM A TERMINAL • 5–1
ATTRIBUTES Option • 5–9
_BACK • 5–5
Default input • 5–1
Displaying prompts • 5–8
PROMPT Option • 5–8
ELAPSED Option • 5–19
_EXIT • 5–5
_HELP • 5–5
Inputting multiple variables • 5–6
LENGTH Option • 5–9
SCREEN option
Format Options
digits • 5–16
SCREEN Option • 5–14
angle brackets • 5–15
Example • 5–14
format • 5–14
Format characters • 5–15
Format Options • 5–15, 5–16
AJ • 5–19
LCASE • 5–17
FORMAT OPTIONS
AT • 5–19
REQ • 5–19
Special keys • 5–19

INPUT FROM A TERMINAL
SCREEN Option
Special keys (cont'd)
BACKSPACE • 5–20
RETURN • 5–20
TAB • 5–20
User's response • 5–19
^ • 5–16
SCREEN Option @ • 5–15
See also KEY INPUT • 5–29
TIMEOUT Option • 5–12
To a string variable • 5–5
VALID Option • 5–11
INPUT MENU • 5–21
Input Statement
Simple • 5–6
INPUTTING FROM A FILE • 6–5
LINE INPUT # • 6–7
Inputting Multiple Variables • 5–6
INPUTTING MULTIPLE VARIABLES • 6–6
Inputting Strings • 5–5
INT • A–16
INTEGER • A–16, A–32
Basic description of • 16–5
_INTEGER • A–2
INTEGER NUMERIC DATA
Description of • 16–5
Designation of (%) • 16–5
Size • 16–5
INTEGER VALUES
Representation of • 16–18
INTEGER WORD • A–32
Interactive environment
Example of • 1–12
INTOUCH
/COMPILE • 2–19
Exiting • 1–9
Getting help • 1–9
How to use the system
Examples of • 1–5
INTOUCH Commands • 4–1
INTOUCH FRAME • 1–1
INTOUCH language • xxi
INTOUCH LANGUAGE
Basic description of • 16–1
INTOUCH PROGRAM
Changing • 1–11
Interrupting • 1–9
INTOUCH PROGRAMS
Basic description of • 16–1
INTOUCH PROMPT
Description of • 1–1
INTOUCH SCREEN • 7–15
INTOUCH Tutorial • xxi
.INT_IMG • 2–20
.INT_IMG_AXP • 2–20
IP num_expr • A–16
ITEM • A–16
%ITEMS • 5–25
ITERATE DO • 10–13
ITERATE DO Statement • 10–12
ITERATE FOR • 10–7
ITERATE FOR Statement • 10–6

J

JUSTIFYING STRING VALUES • 4–8

K

KEY • 13–9
KEYED • 13–29
KEY INPUT • 5–29
KEY Option • 13–7
KEYS Option • 13–35
KEYWORDS • 2–1
KILL STR_EXPR • 6–4

L

Labels
on multiple statement lines • 16–4
LABELS
Basic description • 16–4
How to use • 16–4
LBOUND • A–17
LCASE • A–31
See also INPUT FROM A TERMINAL
SCREEN Option
LCASE • 5–16
LCASE$ • A–17, A–35
LEFT[$] • A–17
LEN • A–17
LENGTH • 13–30, A–31
LENGTH OPTION • 5–9
LESS THAN • 16–16
LESS THAN OR EQUAL TO • 16–16

LET • 4–1, 4–7
  Assigning numeric values • 4–7
  Assigning string values • 4–7
LETTERS • A–31
LIBRARY
  STATEMENT
    Description Of • 12–1
LIBRARY NAME
  Description of • 12–1
LIBRARY ROUTINES
  PASSING ARGUMENTS TO • 12–3
    By Descriptor • 12–4
    By Reference • 12–4
    By Value • 12–4
  PASSING OPTIONAL ARGUMENTS TO • 12–3
Library statements
  CALL • 12–3
LIBRARY STATEMENTS
  LIBRARY • 12–1
LIBR_NAME • 12–1
LIMIT EXPRESSION • 10–2
LINE INPUT
  See also INPUT
LINE INPUT # • 6–7
LINE INPUT AREA • 5–4
LINE INPUT Statement • 5–7, 6–6
Line numbers
  As labels • 16–4
LINE NUMBERS
  Description of • 16–3
  Range • 16–3
[LINE] INPUT MENU • 5–21
  See also LINE INPUT
LIST • 2–10
  Suppressing the program header • 2–10
LIST COMMAND
  Example of • 1–11
LIST ERROR • 2–13
LIST STATS • 14–7
LNH
  See LIST • 2–10
LNH COMMAND
  Example of • 1–11
LOAD
  See OLD • 2–9
LOCK
  OPTION TO OPEN STRUCTURE Statement • 13–4
LOCK/UNLOCK STRUCTURE • 13–45
LOCK Option • 13–4
%LOCKSTEP • 5–26
LOG • A–17
LOG10 • A–17
LOG2 • A–17
LOOPS • 10–1
LOW BOUND
  Range • 8–1
LPAD$ • A–17
LSET • 4–9
LSET, RSET, CSET FILL • 4–10
LSET, RSET AND CSET • 4–8
LTRIM$ • A–18, A–34

M

MAIN UNIT
  Description of • 16–2
MARGINS
  Determining margins • 6–9, 9–8
  Setting the margin • 6–11, 9–8
MATCH • A–18
MAX • A–18
MAXLEN • A–18
MAXLENGTH • A–31
MAXNUM • A–18
MAXSIZE • A–18
%MAXWIDTH • 5–24
MENU • A–34
%MENUBAR • 5–27
MENU DIRECTIVES: • 5–22
MENU Option • 5–4, 5–21
MESSAGE • 7–15
  DELAY option • 7–17
  ERROR option • 7–16
%MESSAGE • 3–1
%MESSAGE ERROR • 3–1
MESSAGE Statement • 7–17
MID • A–19
MID$ • A–19
MIN • A–19
MINLENGTH • A–31
MOD • A–19
MONITORING PROGRAM EXECUTION • 14–3
  LIST STATS • 14–7
  Notation of program lines • 14–3
  STATS OFF • 14–7
  STATS ON • 14–6
  TRACE • 14–5
  TRACE ON/OFF • 14–4

%MULTI • 5–22
Multiple Statements • 16–3
  See program lines • 16–3
MULTIPLE VARIABLES • 6–7
MULTIPLICATION • 16–15

N

NAME • 13–30
NESTING
  Of DO loops • 10–8
  Of FOR loops • 10–4
  Of GOSUBs • 10–29
NEW • 2–2
NEXT
  In a FOR EACH loop • 13–17
NOCHARACTERS • A–31
NONAME • 2–3
NOT • 16–17
NOT EQUAL TO • 16–16
NULL STRING • 13–35
  Description of • 16–8
NUMBER • 13–30, A–32
NUMERIC FORMAT CHARACTERS • 7–8
NUMERIC VALUES
  Printing of • 7–4
Numeric Variables • 4–1

O

OBJECT FILE
  See COMPILE • 2–19
OLD • 2–9
ON GOSUB • 10–30
  ELSE option • 10–31
ON GOTO • 10–27
  ELSE Option • 10–28
OPEN Statement • 6–1
OPEN STATEMENT • 6–2
OPEN STRUCTURE
  ACCESS OPTION • 13–4
  DATAFILE option • 13–4
  LOCK ADDITION • 13–4
OPEN STRUCTURE Statement • 13–2, 13–3
OPERANDS
  Basic description of • 16–15
OPERATORS
  See ARITHMETIC OPERATORS • 16–15
OPTIMIZED • 13–30
OPTIMIZE OFF Option • 6–2
OPTIONAL ARGUMENTS
  IN THE CALL STATEMENT • 12–3
OPTION BASE • 4–5, 8–5
OPTION COMMUNICATE ON/OFF • 7–15, 7–20
Option Menus • 1–3
OPTION REQUIRE DECLARE
  Description of • 4–5
OPTIONS • 2–1
OR • 16–17
ORD • A–19
ORDER OF EVALUATION • 16–19
ORDNAME$ • A–19
ORIGINAL FILES • 2–4
OUTPUT • 13–35

P

Parentheses
  in expressions • 16–19
PARSE$ • A–19
PARTIAL KEY Option • 13–8
PASS • 1–1
Passing
  By Value • 12–4
PASSING
  By Descriptor • 12–4
  By Reference • 12–4
PASSING MECHANISMS
  IN THE CALL STATEMENT • 12–4
PASS NORETURN • 10–25
PASS [NOWAIT: | NORETURN:] • 10–24
PATTERN • A–33
PATTERN() • A–20
PAUSE • 2–24
Percent sign • 16–5
PERFORMANCE TOPICS • G–5
PI • A–21
PIECE$() • A–21
PIECES() • A–21
POS • A–22
POSITION • 13–31
POUND SIGN
  In Structure References • 16–13
PREFIX
  DOIT • 4–3
PRETTY$ • A–22

PRINT • 1–5
PRINT # • 7–14
PRINTER TYPE • 9–10
Printing
  INTOUCH programs • 16–5
PRINTING TO A FILE • 7–14
PRINTING TO A TERMINAL • 7–1
  AT • 7–3
  ERASE • 7–4
  Printing numeric values • 7–4
    Precision • 7–5
  Separators • 7–2
    Comma • 7–2
    Semicolons • 7–2
  TAB • 7–3
  WIDE • 7–4
PRINTMASK • 13–31, A–33
PRINT Statement • 7–1
PRINT STATEMENT
  USING • 7–6
PRINT USING • 7–7
  Description of • 7–6
PRINT ZONES • 7–2
  Asking the ZONEWIDTH • 9–32
  Commas • 7–2
  Setting the ZONEWIDTH • 9–32
Program
  Example of • 1–6
PROGRAM • 14–16
PROGRAM EXECUTION
  Ending • 14–16
  Example of • 16–1
  Interrupting • 14–16
  Order of • 16–2
  See RUN • 2–5
Program header
  With the LIST command • 2–10
Program headers • 2–6
  With RUN command • 2–6
PROGRAM HEADING • 2–3
Program lines
  Multiple statement lines • 16–3
PROGRAM LINES
  Description of • 16–2
PROGRAMS
  Writing with the editor
    See EDIT • 2–14
PROGRAM STATEMENT • 14–17
PROGRAM STRUCTURE
  Description of • 16–2
PROGRAM UNITS
  Description of • 16–2
PROMPT • 13–31
PROMPT Option • 5–4, 5–8
PROMPTS
  Displaying • 5–7
Prompt Text • 15–9
"PROMPT TEXT" • 5–7
Purpose
  of this guide • xxi

Q

QUOTE$ • A–23

R

RAD • A–23
Random • A–24
RANDOMIZE • 14–21
Rdb • G–1, G–4
  CALL SETUP • G–4
  Closing a structure • G–4
  Fields
    Table • G–5
  Sample Table Definition • G–5
  Setup • G–4
Rdb>Segemented keys • G–4
READ
  DESCRIPTION OF • 4–11
READING FROM A FILE
  See INPUT # • 6–5
REAL • A–23
_REAL • A–2
REAL NUMBER
  Basic description of • 16–5
REAL NUMERIC DATA
  Description of • 16–6
  Precision • 16–6
  Range • 16–6
  Without a decimal point • 16–6
RECORDS
  Description of • 16–12
REDIM • 8–4
REEXTRACT Statement • 13–10
Reference
  passing by • 12–4

REFRESH • 2–17
RELATIVE • 13–35
REM • 14–14
REMAINDER • A–23
REMOVE • A–35
RENUMBER • 2–15
REPEAT$ • A–24
REPEAT DO • 10–12
REPEAT DO Statement • 10–11
REPEAT FOR • 10–6
REPEAT FOR Statement • 10–5
REPEAT ROUTINE • 10–34
REPLACE • 2–7, A–35
%REPLACE • 5–22
REPLACE$ • A–24
_REPLY • A–2
REQ • 5–18
REQUIRED • A–33
RESERVED WORDS • B–1
Response to the SCREEN Option • 5–20
RESTORE • A–35
  DESCRIPTION OF • 4–13
RESUME Statement • 11–7
RETRY Statement • 11–5
RETURN
  From GOSUB • 10–29
  From ON GOSUB • 10–30
RETURN KEY
  In the screen management facility • 5–20
RIGHT • A–24
RIGHT$ • A–24
RND • A–24
  RANDOMIZE Statement • 14–21
RNH
  See RUN • 2–5, 2–6
RNH COMMAND
  Example of • 1–6
ROUND • A–25
ROUNDING • 16–10
ROUTINE • 10–33
ROUTINE NAME
  Description of • 12–3
RPAD$ • A–25
RSET • 4–9
RTRIM$ • A–25, A–34
RUN • 2–5
  Suppressing the program header • 2–6
RUN COMMAND
  Example of • 1–6
Runtime Commands
  Example of • 1–5

S

SAVE • 2–4
  See also REPLACE • 2–7
  See also Storing INTOUCH programs • 2–4
Scale Factors • 15–8
SCAN • A–25
SCIENTIFIC NOTATION
  See E NOTATION • 16–7
SCREEN
  CLEAR • 7–18
    AREA Option • 7–18
  INTOUCH • 7–15
Screen Format Options • 5–16
SCREEN MANAGEMENT FACILITY
  See also INPUT FROM A TERMINAL, SCREEN Option • 5–14
  Special keys • 5–19
  User's response • 5–19
SCREENMASK • 13–32
SCREEN Option • 5–14
  User's Response to • 5–20
SEC • A–25
Secant • A–25
SECONDS • A–25
SEG$ • A–26
SELECT CASE • 10–19
  CASE ELSE Option • 10–21
  CASE statement • 10–20
  Execution of • 10–20
SELECT CASE CONSTRUCT
  When to use • 10–14
SEPARATORS
  See PRINTING TO A TERMINAL, Separators • 7–2
SET • 6–11, 9–1
  #chnl_num
    MARGIN • 6–11
  Devices • 6–11
  Options • 6–11
    ZONEWIDTH • 6–11
SET #chnl_num
  options
    CURRENT • 6–12
SET AUTOEXIT • 9–1
SET BACK OFF • 9–3
SET BACK ON • 9–2
SET ERROR OFF • 9–4
SET ERROR ON • 9–3

SET EXIT OFF • 9–6
SET EXIT ON • 9–5
SET HELP OFF • 9–7
SET HELP ON • 9–6
SET MARGIN • 9–8
SET MESSAGELINE • 9–9
SET PORT OFF • 9–11
SET PORT ON • 9–9
SET SCROLL • 9–11
SET SEED • 9–12
SET STATEMENT • 6–11
SET STRUCTURE • 13–40
  CURRENT Option • 13–18
  KEY Option • 13–19
SET STRUCTURE, FIELD: PARTIAL KEY • 13–19
SET STRUCTURE, SET, USING • 13–24
SET STRUCTURE, SET, USING, FIELD: KEY • 13–23
SET STRUCTURE, SET, USING: EXPRESSION 'owner' • 13–22
SET STRUCTURE, SET: EXPRESSION 'owner' • 13–23
SET STRUCTURE : EXTRACTED 0 • 13–22
SET STRUCTURE: ID • 13–20
SET STRUCTURE: POINTER • 13–21
SET STRUCTURE: RECORD • 13–21
SET STRUCTURE:TIMEOUT • 13–40
SET SYSTEM, LOGICAL • 9–16
SET SYSTEM, SYMBOL • 9–20
SET SYSTEM: COMMENT • 9–14
SET SYSTEM: DIRECTORY • 9–15
SET SYSTEM: PROCESS • 9–19
SETUP • G–6
  Access Rules • 15–10
  Call • 15–2
  Create • 15–1
  Create Procedure • 15–14
  Data File • 15–10
  Data Type • 15–7
  Define • 15–1
  Define Procedure • 15–5
  Definition File • 15–3
  Definitions • 15–11
  Description • 15–6
  Ending-position • 15–8
  Exit • 15–1, 15–10
  Field Attribute
    Date • 15–7
  FIELD ATTRIBUTES • 15–8
  Field Length • 15–8
  Field Name • 15–6, 15–10
SETUP (cont'd)
  Help Text • 15–10
  Print Mask • 15–9
  Procedure • 15–4
  Record System (RMS) • 15–3
  Report Heading • 15–9
  Screen Mask • 15–10
  Show • 15–1
  Show Procedure • 15–11
  Structure Name • 15–3
SETUP prompts • 15–3
SETUP ROUTINE • 15–1
SET WINDOW • 9–24
SET WINDOW: COLUMN • 9–23
SET WINDOW: DATA • 9–25
SET WINDOW: KEYMAP • 9–27
SET WINDOW: ROW • 9–30
SET WINDOW: TYPEAHEAD • 9–31
SET WINDOW AREA • 9–22
SET WINDOW KEYSTROKE:VALUE • 9–28
SET ZONEWIDTH • 9–32
SGN • A–26
SHOW
  Definitions • 15–11
  Field Definitions • 15–11
SIN • A–26
SINH • A–26
SIZE • A–26
%SIZE number • 5–23
SKIP • A–26
SLEEP • 2–24
SORT$ • A–27
SORT BY • 13–9
SORTING BY KEYS
  See KEY OPTION • 13–16
SORT OPTION • 13–15
  ASCENDING • 13–15
  DESCENDING • 13–15
SPACE$ • A–27
Specifications • xxi
Spelling Correction • 1–4
%SPLIT • 5–25
SQR • A–27
STATEMENTS
  Basic description of • 16–1
  CAUSE EXCEPTION • 11–1
  Continuing • 16–4
  MESSAGE • 7–17
  Presentation of • 2–1
  PRINT • 7–6
  PRINT USING • 7–6

STATISTICS • 14–5
Description • 14–3
LIST STATS • 14–7
Notation of program lines • 14–3
STATS OFF • 14–7
STATS ON • 14–6
STATS OFF • 14–6, 14–7
STATS ON • 14–6
_STATUS • A–2
STEP • 14–8, 14–9, 14–10
STEP OPTION • 10–3
STOP • 14–16, 14–18
See also HALT • 14–20
Storage Locations • 4–1
Storing INTOUCH programs • 16–5
STR$ • A–27
_STRING • 5–10, A–3
STRING DATA
Comparing string values • 16–17
Concatenation • 16–15
Constants • 16–8
Delimiters • 16–8
Delimiters • 16–8
Description of • 16–6
Designation of ($) • 16–6
Length • 16–6
Quotes as part of text • 16–8
STRING FUNCTIONS
ADDED
DAY$ • A–1
STRINGS
Basic description of • 16–5
String Variables • 4–1
STRUCTURE
Closing • G–4
Description of • 16–12
Setting Up • G–5
STRUCTURE REFERENCE
Description of • 16–12
Format of • 16–12
Structure references
As variables • 16–7
STRUCTURE REFERENCES
FIELD Expressions • 16–12
STRUCTURE Statements
ADDED
REEXTRACT • 13–10
OPEN STRUCTURE
ACCESS OPTION • 13–4
STRUCTURE STATEMENTS
EXIT EXTRACT • 13–10
STRUCTURE STATEMENTS (cont'd)
OPEN STRUCTURE
DATAFILE option • 13–4
SUBROUTINES
With GOSUB • 10–29
SUBSCRIPTS
See ARRAY
Subscripts • 16–10
Substrings
As variables • 16–7
SUBSTRINGS
Description of • 16–11
Example of • 16–11
SUBTRACTION • 16–15
SYSTEXT$ • A–27

T

TAB • 5–20, A–28
As a print function • 7–3
TAB key • 1–3
TAB KEY
In the screen management facility • 5–20
TABLES
See ARRAY • 16–10
TAN • A–28
Tangent • A–28
TANH • A–28
TARGET
Of a GOSUB • 10–29
Of a GOTO • 10–26
Of an ON GOTO • 10–27
Of ON GOSUB • 10–30
TERMINATING LINES
How to • 16–3
_TERMINATOR • 2–25, 5–29, A–3
TEXT STRINGS • G–4
TILDE • 14–16
TIME • A–28
TIME$ • A–28
TIME(5) • A–29
TIMEOUT Option • 5–4, 5–12
TRACE • 14–4
Description • 14–3
Notation of program lines • 14–3
TRACE OFF • 14–5
TRACE ON/OFF • 14–4
TRACE OFF • 14–5
TRANSACTION • G–4

TRANSFER OF CONTROL
Within code blocks • 10–34
Translate • A–36
Transparent interface • xxi
Trim
leading spaces • A–18
TRIM$ • A–29, A–34
TRUE
As a function • A–29
Definition of • 16–17
In the UNTIL option • 10–9
In the WHILE option • 10–9
See EXPRESSIONS
Conditional • 16–16
TRUNCATE • A–29
TYPE AHEAD
Purging • 7–16

U

UCASE • A–32
See also INPUT FROM A TERMINAL SCREEN Option
UCASE • 5–16
UCASE$ • A–29, A–35
UNDERSCORE CHARACTER
In fill-in areas • 5–14
UNFORMATTED • 6–3
UNFORMATTED Option • 6–1, 6–2
UNIQUE FILE SPECIFICATIONS • 2–4
UNLOCKING • G–4
UNLOCK STRUCTURE • 13–45
UNQUOTE$() • A–29
UNSAVE • 2–22
UNTIL • 10–9
Placement of • 10–10
UNTIL OPTION • 10–9
USE Statement • 11–3
USING VAX FACILITIES
See DCL PASS THRU • 1–1

V

VAL • A–30
VALID • A–30
VALID OPTION • 5–11
Value
Passing by • 12–4
Variable Length Fields • 15–8
VARIABLES
Description of • 16–9
Integer
Assigning reals to • 16–10
INTEGER
Designation of • 16–10
STRING
Designation of • 16–10
%VBAR • 5–28
VMS prompt
($) • 1–1
VRULES • 13–33, A–33

W

WAIT • 2–24
WHEN EXCEPTION IN • 11–2, 11–3
WHEN EXCEPTION USE • 11–3, 11–4
WHILE • 10–9
Placement of • 10–10
WIDE
As a print option • 7–4
%WIDTH • 5–23
WORDS
Value of • 16–18
WORKSPACE NAME
See also OLD • 2–9
See NEW • 2–2
WRAP$() • A–36
WRITING TO A FILE
See PRINTING TO A FILE • 7–14

X

XLATE$ • A–36
XOR • 16–17

Y

YES/NO • A–33

Z

ZE • 15–7
ZONEWIDTH • 9–32