INTOUCH® 4GL Specifications

This document contains PROPRIETARY and CONFIDENTIAL information and such information may not be disclosed to others for any purpose without written permission from:

Touch Technologies, Inc.
9988 Hibert Street, Suite 310
San Diego, California 92131
Support (800) 525-2527

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

NOTICE

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

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

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

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

DYNAMIC TAPE ACCELERATOR INTOUCH CleanDisk

DYNAMIC LOAD BALANCER PLUS REMOTE DEVICE FACILITY

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

DBMS DCL DECNET RDB RMS VAX OpenVMS

Last Revision date: 20-Oct-1995

Copyright ©1992, 1995 Touch Technologies, Inc.

Contents

PREFACE xv

CHAPTER 1 USING SETUP WITH STRUCTURES, FIELDS 1–1

1.1 SETUP–OVERVIEW 1–1
1.1.1 Entering SETUP 1–1

1.2 SETUP PROMPTS 1–2
1.2.1 Structure Name 1–2
1.2.2 Database engine 1–2
1.2.3 Dataset 1–3
1.2.4 Data dictionary 1–4

1.3 DEFINE PROCEDURE 1–6
1.3.1 Field name 1–7
1.3.2 Classification 1–8
1.3.3 Description 1–11
1.3.4 Data type 1–11
1.3.5 First position 1–12
1.3.6 Field length 1–12
1.3.7 Number of occurrences 1–13
1.3.8 Semantics 1–13
1.3.9 Prompt text 1–14
1.3.10 Report heading 1–14
1.3.11 Print mask 1–14
1.3.12 Screen mask 1–14
1.3.13 Help text 1–15
1.3.14 Access rules 1–15
1.3.15 Validation rules 1–15

1.4 SHOW PROCEDURE 1–16
1.4.1 Field Displays 1–17
1.4.2 Structure Display 1–18

1.5 MODIFY STRUCTURE PROCEDURE 1–19

Contents

1.5.1 Modify Structure General Information 1–19
1.5.2 Modify Structure Security Information 1–20

1.6 CREATE PROCEDURE 1–22

1.7 DEFINING COBOL RECORD FIELDS 1–25
1.7.1 Define Example 1–25

CHAPTER 2 INTOUCH RUNNING UNDER VMS 2–1

2.1 FILE SPECIFICATIONS 2–1

CHAPTER 3 INTOUCH DATA STRUCTURES 3–1

3.1 GENERAL DESCRIPTION 3–1
3.1.1 Using Structures within Expressions 3–2
3.1.2 Using PRINT with Fields 3–2
3.1.3 Field Expressions 3–2
3.1.4 Multiple Occurrence Fields 3–3

3.2 STRUCTURE STATEMENTS 3–3
3.2.1 OPEN STRUCTURE 3–3
3.2.2 CLOSE STRUCTURE 3–4
3.2.3 ADD STRUCTURE 3–5
3.2.4 CANCEL ADD 3–5
3.2.5 EXIT ADD 3–6
3.2.6 DELETE STRUCTURE 3–7
3.2.7 LOCK|UNLOCK STRUCTURE 3–7
3.2.8 EXTRACT STRUCTURE 3–8
3.2.9 REEXTRACT STRUCTURE 3–9
3.2.10 FOR EACH 3–10
3.2.11 CANCEL EXTRACT 3–11
3.2.12 EXIT EXTRACT 3–12
3.2.13 EXTRACT STRUCTURE: KEY 3–12
3.2.14 EXTRACT STRUCTURE: PARTIAL KEY 3–13
3.2.15 EXTRACT STRUCTURE: APPEND 3–14
3.2.16 EXTRACT STRUCTURE, SET, USING 3–15

3.2.17 EXTRACT STRUCTURE, SET, FIELD: KEY 3–16

3.3 ASK STRUCTURE 3–16
3.3.1 ASK STRUCTURE: CURRENT 3–16
3.3.2 ASK STRUCTURE: FIELDS 3–17
3.3.3 ASK STRUCTURE: KEYS 3–18
3.3.4 ASK STRUCTURE FIELD: item 3–18
3.3.5 ASK STRUCTURE: CAPABILITY 3–21
3.3.6 ASK STRUCTURE: ENGINE 3–22
3.3.7 ASK STRUCTURE: EXTRACTED 3–23
3.3.8 ASK STRUCTURE: ID 3–23
3.3.9 ASK STRUCTURE: POINTER 3–24
3.3.10 ASK STRUCTURE: RECORD 3–25
3.3.11 ASK STRUCTURE: RECORDSIZE 3–25
3.3.12 ASK STRUCTURE: ACCESS 3–26
3.3.13 ASK STRUCTURE: TIMEOUT 3–26

3.4 ASK | SET STRUCTURE #STRING_EXPR 3–26

3.5 SET STRUCTURE 3–27
3.5.1 SET STRUCTURE: EXTRACTED 0 3–27
3.5.2 SET STRUCTURE: CURRENT 3–28
3.5.3 SET STRUCTURE FIELD:KEY 3–29
3.5.4 SET STRUCTURE FIELD: PARTIAL KEY 3–29
3.5.5 SET STRUCTURE: ID 3–30
3.5.6 SET STRUCTURE:POINTER 3–31
3.5.7 SET STRUCTURE: RECORD 3–31
3.5.8 SET STRUCTURE: TIMEOUT 3–32
3.5.9 SET STRUCTURE, SET, USING 3–32
3.5.10 SET STRUCTURE, SET, USING: EXPRESSION 'owner' 3–32
3.5.11 SET STRUCTURE, SET: EXPRESSION 'owner' 3–33
3.5.12 SET STRUCTURE, SET, USING, FIELD: KEY 3–33

CHAPTER 4 DEBUG SYSTEM 4–1

4.1 DEBUG ON/OFF 4–1
4.1.1 DEBUG ON 4–1
4.1.2 DEBUG OFF 4–2

4.2 THE TRACE FEATURE 4–2

4.2.1 TRACE ON 4–2
4.2.2 TRACE OFF 4–3
4.2.3 TRACE Feature Example 4–3

4.3 STATISTICS FEATURE 4–4
4.3.1 STATS ON 4–4
4.3.2 STATS OFF 4–5
4.3.3 LIST STATS 4–5

4.4 BREAK AND STEP FEATURES 4–6
4.4.1 BREAK 4–6
4.4.2 STEP 4–7
4.4.3 GO 4–8

CHAPTER 5 INTOUCH COMMANDS 5–1

5.1 THE EXIT, HELP AND BACKUP COMMANDS 5–1

5.2 THE INTOUCH COMMANDS 5–2
5.2.1 RUN 5–2
5.2.2 NEW 5–2
5.2.3 SAVE 5–3
5.2.4 REPLACE 5–3
5.2.5 ABORT 5–4
5.2.6 OLD 5–4
5.2.7 UNSAVE 5–5
5.2.8 LIST 5–5
5.2.9 LIST ERROR 5–6
5.2.10 EDIT 5–6
5.2.11 RENUMBER 5–7
5.2.12 FRAME ON | OFF 5–7
5.2.13 REFRESH 5–7
5.2.14 IDE 5–8
5.2.15 COMPILE 5–8
5.2.15.1 Compiled Images • 5–9
5.2.15.2 Running Compiled Images • 5–9
5.2.15.3 Developing INTOUCH Programs • 5–10

CHAPTER 6 PROGRAM DIRECTIVES AND GENERAL STATEMENTS 6–1

6.1 COMPILE DIRECTIVES 6–1
6.1.1 %INCLUDE 6–1
6.1.2 %INCLUDE CONDITIONAL 6–1
6.1.3 %DEBUG 6–2
6.1.4 %MESSAGE 6–2
6.1.5 %MESSAGE ERROR 6–2

6.2 GENERAL USE STATEMENTS 6–3
6.2.1 PROGRAM 6–3
6.2.2 END 6–4
6.2.3 STOP 6–4
6.2.4 HALT 6–5
6.2.5 RANDOMIZE 6–5
6.2.6 DELAY 6–6

CHAPTER 7 ASSIGNING VARIABLES 7–1

7.1 STATEMENTS USED TO ASSIGN VARIABLES 7–1
7.1.1 LET 7–1
7.1.2 LSET 7–2
7.1.3 RSET 7–2
7.1.4 CSET 7–3
7.1.5 LSET, RSET, CSET FILL 7–3
7.1.6 DATA and READ 7–4
7.1.7 RESTORE 7–4
7.1.8 DECLARE STRING | INTEGER | REAL | DYNAMIC 7–5
7.1.9 DECLARE STRUCTURE 7–6
7.1.10 DECLARE PREFIX 7–7
7.1.11 OPTION REQUIRE DECLARE 7–9
7.1.12 OPTION BASE 7–9

CHAPTER 8 INPUTTING DATA FROM A TERMINAL 8–1

8.1 THE INPUT STATEMENT 8–1

8.2 STATEMENT TYPES AND INPUT STYLES 8–2

8.3 RESPONSE COMMANDS AND KEYSTROKES AVAILABLE TO THE USER 8–3
8.3.1 Editing Input 8–4

8.4 INPUTTING STRINGS 8–4

8.5 INPUTTING MULTIPLE VARIABLES 8–5
8.5.1 Simple Input Statement 8–5
8.5.2 LINE INPUT Statement 8–5

8.6 INPUT OPTIONS 8–6
8.6.1 AREA Option 8–6
8.6.2 AT Option 8–7
8.6.3 ATTRIBUTES option 8–8
8.6.4 DEFAULT Option 8–8
8.6.5 ELAPSED Option 8–9
8.6.6 ERASE option 8–9
8.6.7 LENGTH Option 8–9
8.6.8 MENU Option 8–10
8.6.8.1 Menu Directives • 8–11
8.6.9 PROMPT Option 8–13
8.6.10 SCREEN Option 8–13
8.6.10.1 Screen Format Characters • 8–14
8.6.10.2 Screen Format Options • 8–15
8.6.11 TIMEOUT Option 8–16
8.6.12 VALID Option 8–16

8.7 KEY INPUT 8–16

CHAPTER 9 PRINTING DATA TO A TERMINAL 9–1

9.1 PRINT 9–1
9.1.1 PRINT Options 9–2
9.1.1.1 TAB Function • 9–2
9.1.1.2 AT row, column • 9–2
9.1.1.3 BLINK | UNDERLINE | BOLD | REVERSE • 9–2
9.1.1.4 WIDE Option • 9–3
9.1.1.5 ERASE Option • 9–3
9.1.1.6 PRINT USING Option • 9–3

9.2 USING THE INTOUCH SCREEN 9–6

9.3 MESSAGE [ERROR: | DELAY:] 9–6

9.4 CLEAR 9–7
9.4.1 AREA Option 9–8
9.4.2 BOX Option 9–9

9.5 OPTION COMMUNICATE ON | OFF 9–10

CHAPTER 10 DEFINING ARRAYS 10–1

10.1 ARRAYS 10–1
10.1.1 DIM 10–1
10.1.2 REDIM 10–2
10.1.3 OPTION BASE 10–3

CHAPTER 11 SET AND ASK STATEMENTS 11–1

11.1 GENERAL SET, ASK STATEMENTS 11–1
11.1.1 SET AUTOEXIT 11–1
11.1.2 SET BACK ON | OFF 11–2
11.1.3 SET ERROR ON | OFF 11–2
11.1.4 ASK ERRORS 11–3
11.1.5 SET EXIT ON | OFF 11–4
11.1.6 SET HELP ON | OFF 11–4
11.1.7 ASK KEYSTROKES 11–5
11.1.8 ASK | SET MARGIN 11–5
11.1.9 ASK | SET MESSAGELINE 11–6
11.1.10 ASK PAGESIZE 11–7
11.1.11 SET PORT ON | OFF 11–7
11.1.12 ASK RESPONSES 11–8
11.1.13 SET SCROLL 11–9
11.1.14 ASK | SET SEED 11–9
11.1.15 ASK | SET ZONEWIDTH 11–10

11.2 ASK | SET SYSTEM 11–11

11.2.1 ASK | SET SYSTEM: COMMENT 11–11
11.2.2 ASK | SET SYSTEM: DIRECTORY 11–11
11.2.3 ASK | SET SYSTEM, LOGICAL: VALUE 11–12
11.2.4 ASK SYSTEM: MODE 11–13
11.2.5 ASK SYSTEM: PARAMETER 11–13
11.2.6 ASK SYSTEM, PASS: SUCCESS 11–14
11.2.7 ASK | SET SYSTEM: PROCESS 11–14
11.2.8 ASK SYSTEM: RIGHTS 11–15
11.2.9 ASK | SET SYSTEM, SYMBOL: VALUE 11–15
11.2.10 ASK SYSTEM: USER 11–16

11.3 ASK | SET WINDOW 11–16
11.3.1 ASK | SET WINDOW AREA 11–17
11.3.2 ASK | SET WINDOW: COLUMN 11–18
11.3.3 ASK | SET WINDOW: CURRENT 11–18
11.3.4 ASK | SET WINDOW: DATA 11–19
11.3.5 ASK | SET WINDOW: KEYMAP 11–20
11.3.6 SET WINDOW KEYSTROKE 11–21
11.3.7 ASK | SET WINDOW: ROW 11–22
11.3.8 ASK | SET WINDOW: TYPEAHEAD 11–23

CHAPTER 12 LOOPS, CONDITIONALS, CHAINING, BRANCHING 12–1

12.1 LOOPS 12–1
12.1.1 FOR . . . NEXT Loop 12–1
12.1.2 EXIT FOR 12–2
12.1.3 REPEAT FOR 12–3
12.1.4 ITERATE FOR 12–3
12.1.5 DO Loop 12–4
12.1.6 EXIT DO 12–5
12.1.7 REPEAT DO 12–6
12.1.8 ITERATE DO 12–6

12.2 CONDITIONALS 12–7
12.2.1 IF/THEN . . . END IF 12–7
12.2.2 SELECT CASE . . . END SELECT 12–8

12.3 BRANCHING 12–9
12.3.1 GOTO 12–9
12.3.2 ON . . . GOTO . . . ELSE 12–10

12.3.3 GOSUB . . . RETURN 12–11
12.3.4 ON . . . GOSUB 12–12

12.4 EXECUTE 12–13

12.5 DISPATCH 12–14

12.6 ROUTINE/END ROUTINE 12–15
12.6.1 EXIT ROUTINE 12–15
12.6.2 REPEAT ROUTINE 12–16

12.7 CHAINING PROGRAMS 12–16
12.7.1 CHAIN 12–16

12.8 PASS COMMANDS TO THE OPERATING SYSTEM 12–17
12.8.1 PASS [NOWAIT: | NORETURN:] 12–17

CHAPTER 13 FILE HANDLING 13–1

13.1 OPEN #CHNL_NUM: 13–1

13.2 CLOSE #CHNL_NUM 13–2

13.3 PRINT #CHNL_NUM 13–3

13.4 INPUT #CHNL_NUM 13–3

13.5 LINE INPUT #CHNL_NUM 13–4

13.6 ASK #CHNL_NUM: 13–5

13.7 SET #CHNL_NUM: 13–6

13.8 DELETING FILES 13–7

13.8.1 KILL 13–7

CHAPTER 14 EXCEPTION HANDLING 14–1

14.1 CAUSE EXCEPTION 14–1

14.2 WHEN EXCEPTION IN 14–1

14.3 WHEN EXCEPTION USE 14–2

14.4 HANDLER ROUTINE ACTIONS 14–3
14.4.1 EXIT HANDLER 14–3
14.4.2 RETRY 14–4
14.4.3 CONTINUE 14–5
14.4.4 RESUME 14–6

CHAPTER 15 PROGRAM ELEMENTS AND DATA IN INTOUCH 15–1

15.1 INTOUCH PROGRAM ELEMENTS 15–1
15.1.1 Continuing Program Lines 15–3
15.1.2 Labels 15–4
15.1.3 Storing INTOUCH Programs 15–5

15.2 COMMENTS 15–5
15.2.1 REM 15–5
15.2.2 ! comment_text 15–6

15.3 DATA IN INTOUCH 15–7
15.3.1 Data Types 15–7
15.3.1.1 Integers • 15–7
15.3.1.2 Real Numbers • 15–8
15.3.1.3 String Data • 15–8
15.3.2 BOOLEAN 15–9
15.3.3 Expressions 15–9
15.3.4 Constants 15–9
15.3.4.1 Numeric constants • 15–9
15.3.4.2 E Notation • 15–10
15.3.4.3 String Constants • 15–10

15.3.5 Variables 15–11
15.3.5.1 Arrays • 15–12
15.3.5.2 Substrings • 15–13
15.3.6 Structure References 15–13
15.3.7 Multiple Occurrence Fields 15–15
15.3.8 Compound Expressions 15–16
15.3.8.1 Numeric Expressions • 15–16
15.3.8.2 String Expressions • 15–17
15.3.8.3 Conditional Expressions • 15–17
15.3.9 Order of Evaluation 15–20

CHAPTER 16 CALLING ROUTINES WRITTEN IN OTHER LANGUAGES 16–1

16.1 LIBRARY 16–1

16.2 CALL 16–2
16.2.1 Optional Arguments 16–2
16.2.2 Passing Mechanisms 16–3

APPENDIX A BUILT-IN FUNCTIONS A–1

A.1 SYSTEM FUNCTIONS A–1

A.2 OTHER FUNCTIONS A–3

APPENDIX B CREATING LIBRARIES FOR USE WITH INTOUCH B–1

INDEX

EXAMPLES
1–1 SETUP Prompts 1–2
1–2 SETUP Procedures 1–6
1–3 Field Definition Input Screen 1–7
1–4 DEFINE Screen showing field class generated defaults 1–10
1–5 DEFINE Screen showing SETUP generated defaults 1–10
1–6 SHOW - Brief Field Display Example 1–17

1–7 SHOW - Full Field Display Example 1–18
1–8 SHOW - Fields List Display Example 1–18
1–9 SHOW - Structure Display Example 1–19
1–10 CREATE Procedure RMS Input Screen 1–23
4–1 Debug TRACE ON Example 4–4
4–2 Listing Program Statistics 4–6
4–3 BREAK in DEBUG System 4–7
4–4 STEP Example in DEBUG System 4–8
4–5 Using GO in DEBUG System 4–9
15–1 Executing an INTOUCH program 15–2
15–2 Structures 15–14

FIGURES

1–1 Character Positions 1–12

TABLES

11–1 Supported Keystroke Concepts 11–22
15–1 E Notation 15–10
A–1 CONVERT$ Function - Supported Data Types A–5
A–2 DATE$ Function - Integer Values A–7
A–3 DAYS Function - Integer Values A–8
A–4 EDIT$ Function - Operation Values A–10
A–5 FORMAT$ Function - Date Arguments A–15
A–6 FULLTIME$ Function - Integer Values A–16

Preface

Introduction

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

The INTOUCH language:

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

INTOUCH includes a transparent interface to the RMS file management system, and interfaces to other database engines (DBMS, Rdb, ORACLE, etc.). The interface includes:

- simple structure statements for complex manipulations of data.
- 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–as an interpreter–while code is optimized by the compiler.
- has been integrated into the TPU editor.

Development cycle time is the time it takes to write, 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. These 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 INTOUCH Language*
- *Specifications*

A Guide to the INTOUCH 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* contains a brief description of the INTOUCH system, statements, commands, etc. and some background on how the system operates.

Purpose

The purpose of the *Specifications* manual is to provide a summary description of the INTOUCH system, language and features. This manual should be used solely for reference and informational purposes.

Intended Audience

This manual is written for experienced programmers who have had a thorough exposure to computers and programming languages.

Conventions Used in this Manual

User Conventions

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

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

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

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

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

Program Examples

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

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

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

Other Conventions

Other conventions used in this manual are:

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

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

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

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

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

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

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

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

dashes Three sets of dashes indicate a block of code.

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

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

.
.
.

1 Using SETUP with Structures, Fields

1.1 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.

Procedures

The procedures available in SETUP and described in this chapter are:

- **DEFINE** - Define fields in a structure. Use DEFINE when you want to add new fields, modify existing fields or delete fields.
- **SHOW** - Show field definitions or structure information. Use SHOW to display or print lists of the fields defined for this structure or to display information on this structure.
- **MODIFY** - Modify the structure file. Use MODIFY to change the structure file and/or the structure security and access levels.
- **CREATE** - Create a DATA File.

1.1.1 Entering SETUP

To enter SETUP, get into INTOUCH and type **CALL SETUP** at the INTOUCH prompt.

SETUP will ask you a number of questions including:

structure name
database engine (record system being used)
dataset (data file name)
data dictionary (definition file name)

You can create a new structure by entering new file names or you can modify existing structures by selecting them at the prompts.

Once you have specified the structure you want to work with, you can define fields in the structure or display field definitions.

1.2 SETUP Prompts

This section describes the SETUP prompts and your options for answering them.

When you are using SETUP and are at any of the prompts, you can enter **HELP** or press the Help key to get information about the prompt.

Example 1–1: SETUP Prompts

```
SETUP V4.2              Set Up INTOUCH 4GL Data Structures

Structure name :  CUSTOMER.STR
Database engine:  RMS
Dataset        :  CUSTOMER.DAT
Data dictionary:  CUSTOMER.DEF



Structure name? _____________________________________________________________

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

1.2.1 Structure Name

Enter a structure name. This can be a new name or the name of an existing structure. If you are creating a new structure, this will be the name of the structure file. The default extension for the file is .STR.

Your structure name can be any VMS file specification. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

1.2.2 Database engine

Use the arrow keys to select the database engine (record management system) used by this structure. INTOUCH supports these systems:

RMS
RDB
DBMS
USERBASE
POISE
FASTFILE
S1032
INGRES
DBASE3
ADABAS
ORACLE

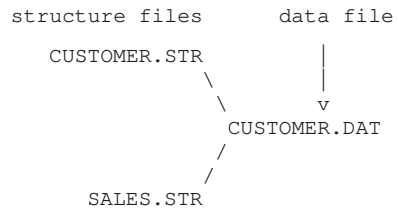
(You can refer to the *INTOUCH Interfaces to Database Engines* manual for more information.)

If you are creating a new structure, the default will be RMS. If you are modifying an existing data structure, the default will be the database engine (record management system) specified in the structure file. FASTFILE is an INTOUCH record management system designed for speed.

1.2.3 Dataset

Enter the name of the data file (dataset) you will be using with this structure. If this is a new structure, the default data file name will be the same as the structure name. The default extension is .DAT.

You can use the same data file for a number of structures. For example, you might use the CUSTOMER data file for both the CUSTOMER and the SALES structures:



Your dataset name can be any VMS file specification. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

If the database engine is **Rdb**, the input format will be different. An Rdb database consists of tables. Each structure corresponds to one table. For Rdb, the dataset is the name of a table in the Rdb database.

```
FORMAT:         table_name IN rdb_database
EXAMPLE:        employee in rdb_data:personnel
```

For a **DBMS** dataset, you need to specify the DBMS record name and the name of the DBMS root file. Optionally, you can add the name of the subschema.

If the subschema is not given, the DEFAULT_SUBSCHEMA is assumed.

```
FORMAT:         record_name IN [subschema_name FOR ] root_file
EXAMPLES:       part in datadisk:[parts]dbmparts
                part in ss0001 for datadisk:[parts]dbmparts
```

For an **INGRES** dataset, you need to specify the INGRES table name. The INGRES database to use is specified by the symbol INT_INGRES_DBNAME.

```
FORMAT:         table_name
EXAMPLE:        employee
```

For an **ADABAS** dataset, you need to specify the file number and the database number.

```
FORMAT:         file_number IN database_number
EXAMPLE:        6 IN 1
```

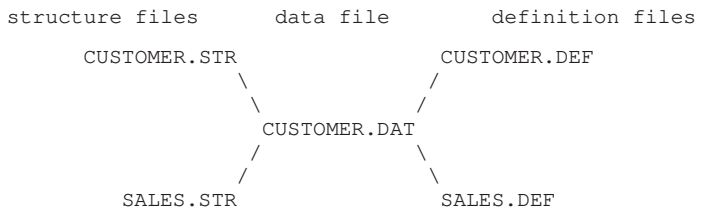
For an **ORACLE** dataset, you need to specify the ORACLE table name.

```
FORMAT:         table_name
EXAMPLE:        emp
```

1.2.4 Data dictionary

Enter the name of the definition file you will be using with this structure. If you are creating a new structure, the default definition file name will be the same as the structure name. The default extension is .DEF.

As with the data file, you can use the same definition file for a number of structures. In fact, you can create any combination of definition, data and structure files:



Your definition file name can be any VMS file specifications. (See the chapter *INTOUCH Running Under VMS* for information on VMS file specifications.)

There are three special definition file names allowed. If the database engine for this structure can supply the field information and you want to use these native definitions, enter **NONE** as the definition file name and INTOUCH will use the definitions embedded in the database engine with the fields.

These database engines can supply the field definitions:

Rdb
DBMS
dBASE III
INGRES
ADABAS
ORACLE

If you are using a database engine that supports native field definitions (Rdb, for example) and you would like to augment the native definitions with additional definitions, you can enter a definition file name prefixed with **AUG>**. This will create an augmented definition file. The native field names will be available along with any fields defined in the augmented definition file. Typically, augmented definitions are used to supply report headings, prompt text and print masks.

For example, if you have an Rdb table which contains of the following native fields:

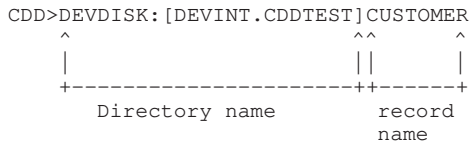
```
Name            Start           Length
-----           -----           ------
fielda            1               10
fieldb           11                5
fieldc           16               10
```

and you want to define a field (an index) consisting of "fielda" and "fieldb", you can create an augmented data dictionary and define the index you want.

Data dictionary? AUG>RDBTEST.DEF

If you are using **CDD**, you can enter **CDD>** and a name of a CDD/Plus Directory and recordname:

Data dictionary? CDD>DEVDISK:[DEVINT.CDDTEST]CUSTOMER



CDD/Plus access is read only at this time.

After the structure information has been entered, the SETUP Procedure Menu will be displayed and you can select the SETUP procedure you want to perform.

Example 1–2: SETUP Procedures

```
SETUP V4.2              Set Up INTOUCH 4GL Data Structures

Structure name :  CUSTOMER.STR
Database engine:  RMS
Dataset        :  CUSTOMER.DAT
Data dictionary:  CUSTOMER.DEF


                        +---Setup Procedure---+
                        |  Define fields      |
                        |  Show             [>|
                        |  Modify structure [>|
                        |---------------------|
                        |  Create data file   |
                        |  EXIT               |
                        +---------------------+

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

Note

Depending on the database engine chosen and the definition file selected, you might not have all of these options.

1.3 DEFINE Procedure

You can use the **DEFINE** procedure to define the fields in a structure. The DEFINE procedure is also used to modify and delete field definitions.

You will be asked first for the name of the field you wish to add, modify or delete. You will then be prompted for information about the field.

Example 1–3: Field Definition Input Screen

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :
Classification  :
Description     :
Data type       :
First position  :          length:          Occurs:
Semantics       :

Prompt text     :
Report heading  :
Print mask      :
Screen mask     :
Help text       :
Access rules    :
Validation rules:


Field name? ____________________________________

                        Enter -fieldname to delete a field
EXIT = Exit                                          \ = Back  HELP = Help
```

1.3.1 Field name

Enter the field name you wish to add, modify or delete. If you are adding a new field, the field name will be used to identify this field. If there are existing fields, you can use one of the **SHOW** options to get a list of the current field names.

If you wish to delete a field, enter a minus sign (**-**) followed by the field name. For example: **-phone**

The field name:

must begin with a letter.
can consist of numbers, letters, underscores and dollar signs.
can be up to 32 characters long.
The last character of the field name can be a "$" or "%".

If you are modifying a field, the defaults for the rest of the DEFINE prompts will be your original responses. You can keep these defaults by pressing the Return key at the prompt. You can change the defaults by typing over the responses.

To return to a previous prompt, enter a "\" (backslash) and then press the Return key. SETUP will return you to the previous prompt where you can change your response.

1.3.2 Classification

Enter the field class or press Return to accept the default of **none**.

The classification is optional.

Classification refers to a field class in the CLASSIFICATION data dictionary. Field classes are pre-defined commonly used data fields.

Refer to the SETUP chapter in *A Guide to the INTOUCH Language* manual for detailed information on setting up and using classification fields.

Using Classifications

There are cases when you use the same field type in a record or in several datasets. When this occurs, setting up a field class will save time in defining fields and help to ensure data integrity.

If, for example, you had three phone fields, you would need to define the three fields using different names but the same characteristics. The phone fields would all be 10 numeric characters in length and be printed in the format (123) 456-7890.

To ensure that all the phone numbers are defined exactly the same, you could define a phone *class* in the CLASSIFICATION data dictionary. The class name could be PHONE and be defined with the length (10), semantics (num), print mask ((###~)~ ###~-####) and validations (digits; minlength 10). Then, whenever you want to define a phone number type field, you would enter the field name, HOME_PHONE for example, with PHONE as the classification and the classification data would automatically become the default answers to the remaining DEFINE prompts.

You would also be able to use the PHONE classification when defining phone fields in other dataset records.

Before you can use field classifications, you must first define the field classes in the CLASSIFICATION data dictionary.

Creating Field Classes

Field classes are defined the same way as any other data field. However, these fields are located in a specific data file called TTI_RUN:CLASSIFICATION. To create a field class, perform the following steps:

1. Get into SETUP.
2. Enter **TTI_RUN:CLASSIFICATION** as the structure name. The following information will be displayed:

```
Structure name :  TTI_RUN:CLASSIFICATION.STR
Database engine:  RMS
Dataset        :  NONE
Data dictionary:  TTI_RUN:CLASSIFICATION.DEF
```

3 Select the *Define fields* procedure.

4 At the field name prompt, enter the field class name you want to use (i.e. PHONE).

5 Go through each of the prompts and set up the characteristics you want this field class to have.

Using an Alternate Classification File

If you do not want to use the provided TTI_RUN:CLASSIFICATION data file, you can create one of your own. For example, you could create a structure such as CLASS_NAMES for your field classes. However, if you choose to do this, you **MUST** define the file as follows:

DEFINE TTI_CLASSIFICATION device:[directory]CLASS_NAMES.DEF

SETUP will then retrieve classification data from CLASS_NAMES instead of TTI_RUN:CLASSIFICATION.

Generated Defaults

After the classification prompt is answered, SETUP displays either the class defaults or defaults generated by SETUP to assist you in defining the field.

Example 1–4: DEFINE Screen showing field class generated defaults

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :  HOME_PHONE
Classification  :  PHONE
Description     :  Home phone
Data type       :  CH
First position  :  14      length:  10      Occurs:  1
Semantics       :  NUM, RJ

Prompt text     :  Home phone
Report heading  :  Home Phone
Print mask      :  (### )  ### -####
Screen mask     :  none
Help text       :  none
Access rules    :  NORMAL
Validation rules:  digits; minlength 10


Description? Home phone_____________________

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

Example 1–5: DEFINE Screen showing SETUP generated defaults

```
SETUP V4.2                Definitions for CUSTOMER.STR

Fieldname       :  HOME_PHONE
Classification  :  none
Description     :  Home phone
Data type       :  CH
First position  :  14      length:          Occurs:  1
Semantics       :  UC

Prompt text     :  Home phone
Report heading  :  Home Phone
Print mask      :  none
Screen mask     :  none
Help text       :  none
Access rules    :  NORMAL
Validation rules:  none


Description? Home phone_____________________

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

1.3.3 Description

Enter a description of the field. The description can consist of any characters on the keyboard and be up to 30 characters long. If you are adding this field, the default description is the field name with any "$", "%", "_" characters changed to spaces.

1.3.4 Data type

Enter the data type for this field. The data type is the internal representation of the data. The data type can be:

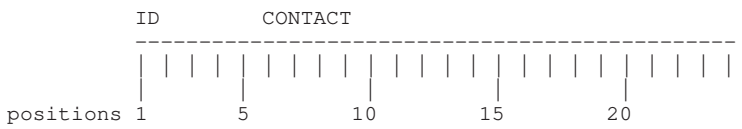
AC	ASCII counted string
AP	Pointer to ASCII counted
C3	COBOL comp-3, also known as packed decimal; the C3 data type also supports the VAX DECIMAL data type
CH	Character
DS	Date stamp (length=8)
EB	EBCDIC
FL	Floating point
GF	G-Float
IN	Integer (signed)
IU	Integer (unsigned)
PF	Packed floating
PZ	Packed zipcode
QS	Quadword (signed)
RO	Right overpunch
RS	Right sign separate
UN	Undefined
ZE	Zoned EBCDIC
ZN	Zoned numeric; used by DIBOL

The DATE (DS) attribute is currently only used by applications and has no effect within INTOUCH.

1.3.5 First position

Enter the position at which this field will start. Character positions are numbered sequentially. For example:

Figure 1–1 Character Positions



In the above diagram, the ID field starts at position 1 and ends at position 5. The CONTACT field starts at position 6, etc.

Fields generally follow each other directly. Therefore, the first position of a new field is usually one more than the last position used. If the last position used was ten, the next field would start at eleven. INTOUCH automatically defaults to the correct sequential position.

Defining existing RMS file fields

If you are defining fields for an existing RMS file, you must keep in mind that the starting position of an RMS file record is 0 and in INTOUCH, the starting position is 1. (RMS is 0 oriented, whereas INTOUCH is 1 oriented.) For example, if the RMS file field starts in position 0, when you define this field in INTOUCH, you must define the field as starting in position 1. If the RMS field starts in position 20, you would use 21 as the starting position, etc.

1.3.6 Field length

Enter the length of this field. The length should be the number of character positions this field will contain. Fields can be any length. The length must be given as an integer number.

You can also enter the ending position for the field length. You enter "P" followed by the ending position. For example: P5

Variable Length Fields

If a CH or UN data type field has a first position that starts at or before the end of a logical record, that field will have its length adjusted upon reference. The new length will reflect the actual end of the logical record.

For example, a field could start in position one, and have its length set to the maximum record length. When the field is referenced, its length will be adjusted to the actual record length. This field is variable length.

1.3.7 Number of occurrences

Enter the number of times that this field occurs in the data record. The default number of occurrences is "1".

If the field being defined is an array, enter the number of elements in the array.

1.3.8 Semantics

Enter the semantics (attributes) that apply to this field. The semantics describe what type of data will be in the field. For example, if this is a dollar amount field you would select NUM:2 which means that this is a numeric field with two decimal places.

The semantics are entered as a list of descriptors separated by commas. When you have completed your list, enter GOLD/F (i.e. press PF1 and then the F key). The valid semantics are:

NUM	Indicates a numeric field which allows only numbers. The semantic *RJ*, right-justified, is automatically appended when NUM is selected.
NUM:scale	SCALE specifies the scale factor—how many decimal places to use. Scale factors can be positive or negative (from -99 to 99).
RJ	The data will be right-justified in the field.
UC	All letters will be stored in upper-case format. Additionally, field comparison data will be upper-cased prior to any comparison.
ZF	The data will be zero filled.
ZS	Causes all leading zeros to be suppressed
DATE[:YMD|MDY]	This field contains a date. The field must be six or eight digits long. The default date format is YMD. You can enter DATE or optionally, specify the date format. The date format is currently only used by applications. INTOUCH does NOT automatically switch date values into or out of the format specified.
FULLTIME	Indicates that the field contains a date-time stamp and the time value is to be preserved. By default, INTOUCH stores/retrieves only the date portion of a date stamp field.
APPLICATION:name	This is a user defined semantic and is not used by INTOUCH. A user can assign a name to indicate that the field is related to a specific application (i.e. accounting). Application programs can then ask for the application name and perform some action based on the data.
DBFLD:fieldname	This is a special semantic for augmented definition files only. The DBFLD semantic is used to associate an augmented field definition with an actual database field. Fieldname is the actual name of the field in the database.

If you do not want any of the semantics, you can enter NONE. If you are modifying an existing field, the default will be the semantics you have previously specified.

1.3.9 Prompt text

Enter the prompt text for this field. The prompt text will be used when data is entered for this field. The prompt text can be any characters on the keyboard and can be up to 30 characters long.

If you are adding this field, the default will be the *description* you entered for this field.

1.3.10 Report heading

Enter the heading you want to appear on reports. This heading will appear at the top of the report column containing this field's data. If you are adding a new record, the default report heading is the *field name* you entered.

The heading can be up to 30 characters long. To make a multi-line heading, separate the heading for each line with a comma (i.e. Cust,No).

1.3.11 Print mask

Enter a print mask for this field. The print mask is used to display information entered into this field. The print mask will be used whenever you use INTOUCH's PRINT statement to print this field's information, or when you use one of the INTOUCH programs such as Guided Query Language.

Print masks are created with the #, @ and % characters. SETUP will automatically provide the tilde (~) character for any literal (non-masking) characters found in the entered mask. Some examples:

```
                          Example 1              Example 2
                          ----------------       -------------
Data to be masked:        AB1234XY               AB1234XY
Data to print as :        (AB-1234-XY)           AB 1234 XY
Mask you enter   :        (##-####-##)           ## #### ##
SETUP displays   :        (##~-####~-##~)        ##~ ####~ ##
```

You can refer to Chapter 9, Printing Data to a Terminal for more information on print masks.

1.3.12 Screen mask

The screen mask is used by applications to enter data into this field. A program can ask for the screen mask and use it in whatever manner that is desired. The format should be the same as for the print mask.

1.3.13 Help text

HELP text can be entered for this field. The help text can be displayed during data entry if the user types "HELP" or presses the Help key at this field prompt when in **MAINTAIN**. Custom applications can also display this help text.

The help text can be up to 60 characters long and can consist of any characters on the keyboard.

1.3.14 Access rules

Enter the access rules for this field. The access rules determine who will have access to this field. NORMAL allows users to read, add and change field information.

Access rules have the following formats:

READ:read_access, WRITE:write_access

NORMAL

where *read_access* and *write_access* are each represented by a single letter from (A-Z).

NORMAL denotes READ:N, WRITE:N.

A is the most restricted access level. **Z** is the least restricted access level.

The access_rules are used in conjunction with the structure security level to determine whether or not a user can access the field.

When a user attempts to access a field, the structure security level is compared to the read or write access level defined for the field. If the field's access level is lower than the structure security level, access is allowed. For example:

> If the structure READ access is J and the field READ access is T, the field data CAN be read. However, if the structure WRITE access is J and the field WRITE access is E, the field CANNOT be written to or changed.

1.3.15 Validation rules

You can enter validation rules for this field. The validation rules will be used to validate data entered by a data entry program.

The data entry program can access these validation rules to make sure that the data entered is valid prior to storing the data into the field.

Any rule that is valid to the INTOUCH VALID function can be entered here.

The validation rules are entered as a list of items separated by semicolons. For example:

required; digits; minlength 10

When you have completed your list, enter GOLD/F (i.e. press PF1 and then the F key).

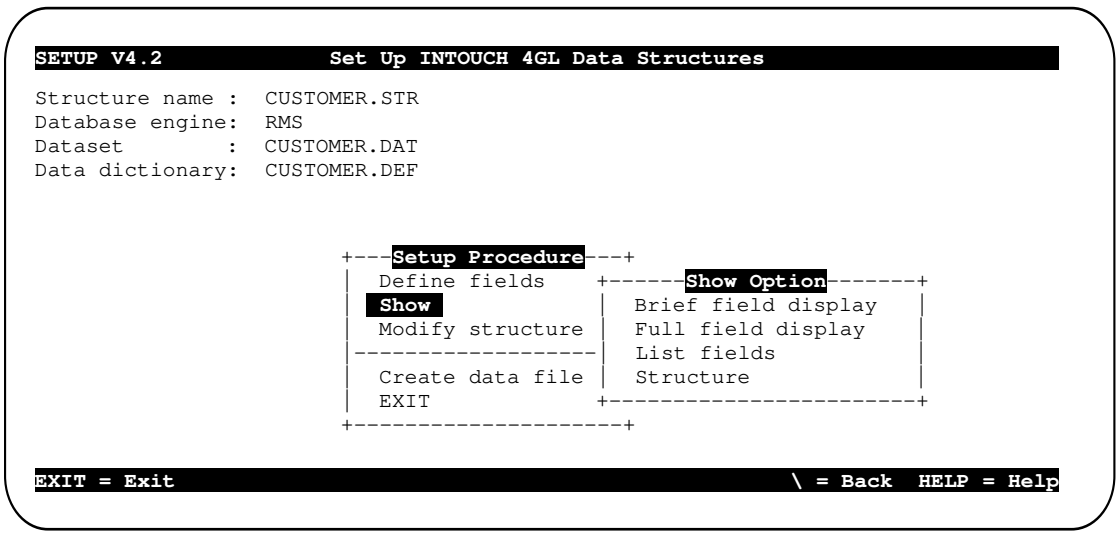
Please refer to Appendix A of this manual for detailed information on the validation rules.

When you have answered all the field prompts, the field definition will be added to the data dictionary. The screen will clear and you can define the next field or enter "\" or "EXIT" to return to the SETUP procedures menu.

1.4 SHOW Procedure

The SHOW procedure displays information about the fields defined in the data dictionary and also displays structure information.

You can use the arrow keys to select the **Show** procedure from the SETUP procedures menu.



The SHOW procedure has four options:

- The **Brief field display** creates a listing of the fields showing some of the definition information.
- The **Full field display** creates a listing of the fields showing all of the definition information.
- The **List fields** option creates a listing of the defined fields.
- The **Structure** option displays information about the structure file.

SETUP creates the listings in report format so that you can display the field and structure information on your screen, print a hardcopy or save the information on disk.

After a report is created, you can select an output option from the Output Options menu. The default is **Screen** which will display the report to your screen. The line under the Output Options menu tells you the name of the temporary report file that was created.

```
+--------------------- Output Options ---------------------+
| Screen          |   System Printer   |   Laser Printer   |
|  Printer Port   |   Save to Disk     |   EXIT            |
+----------------------------------------------------------+

Ready to print user:[tester]setup_show_definition_210006.tmp
```

If you are in the middle of displaying a report and want to exit out of the display, enter CTL/Z and you will return to the Output Options menu.

To return to the SETUP procedures menu, you can use the arrow keys to select the EXIT option or press "\" (backslash key).

1.4.1 Field Displays

The **Brief Field Display** shows the field name, description, first position of the field, field length, number of occurrences, last field position, data type, print mask and semantics.

Example 1–6: SHOW - Brief Field Display Example

```
08-Mar-1995            Definitions for: CUSTOMER.DEF                 Page:   1

Field Name      Description       First  Length  Occurs  Last  Dtype  Print Mask
--------------- ----------------- -----  ------  ------ -----  -----  ----------
CUSTNBR         Customer number       1       5      1      5  CH     none
      Semantics  : none

NAME            Customer name         6      25      1     30  CH     none
      Semantics  : none
```

The **Full Field Display** shows all of the definition information.

Example 1–7: SHOW - Full Field Display Example

```
08-Mar-1995             Definitions for: CUSTOMER.DEF                  Page:   1

Field Name      Description       First  Length  Occurs  Last  Dtype  Print Mask
--------------- ----------------- -----  ------  ------ -----  -----  ----------
CUSTNBR         Customer number       1       5       1      5  CH     none
      Semantics  : none
      Prompt text: Customer number
      Heading    : Cust#
      Help       : none
      Screen mask: none
      Validations: none
```

The **List Fields Display** shows the field name, description, first position of the field, field length and data type.

Example 1–8: SHOW - Fields List Display Example

```
08-Mar-1995                  CUSTOMER.DEF - FIELDS                     Page:   1

Field Name                        Description                     First Length  DT
--------------------------------- ------------------------------- ----- ------  --
CUSTNBR                           Customer number                     1      5  CH
NAME                              Customer name                       6     25  CH
ADDR1                             Address line 1                     31     25  CH
ADDRESS                           Address array                      31     25  CH
ADDR2                             Address line 2                     56     25  CH
CITY                              City                               81     20  CH
```

1.4.2 Structure Display

The Structure option of SHOW displays information about the structure and security access levels.

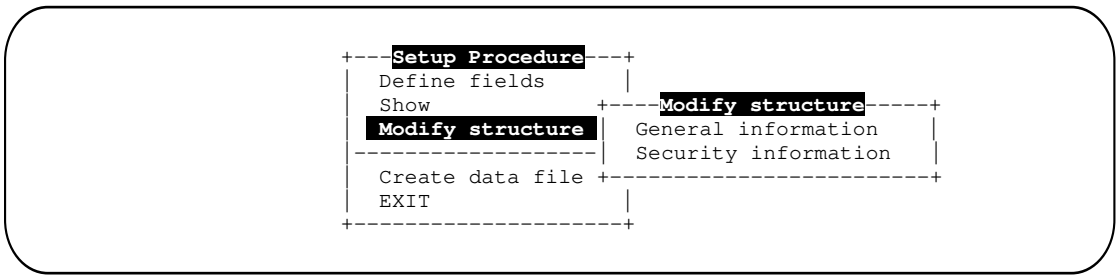
Example 1–9: SHOW - Structure Display Example

```
08-Mar-1995                Structure: CUSTOMER.STR                      Page: 1

Structure name : CUSTOMER.STR
Dataset        : CUSTOMER.DAT
Record system  : RMS
Data dictionary: CUSTOMER.DEF
Access levels  : SECURITY:N, READ:N, WRITE:N, UPDATE:N, DELETE:N
```

1.5 MODIFY STRUCTURE Procedure

The MODIFY STRUCTURE procedures allow you to:

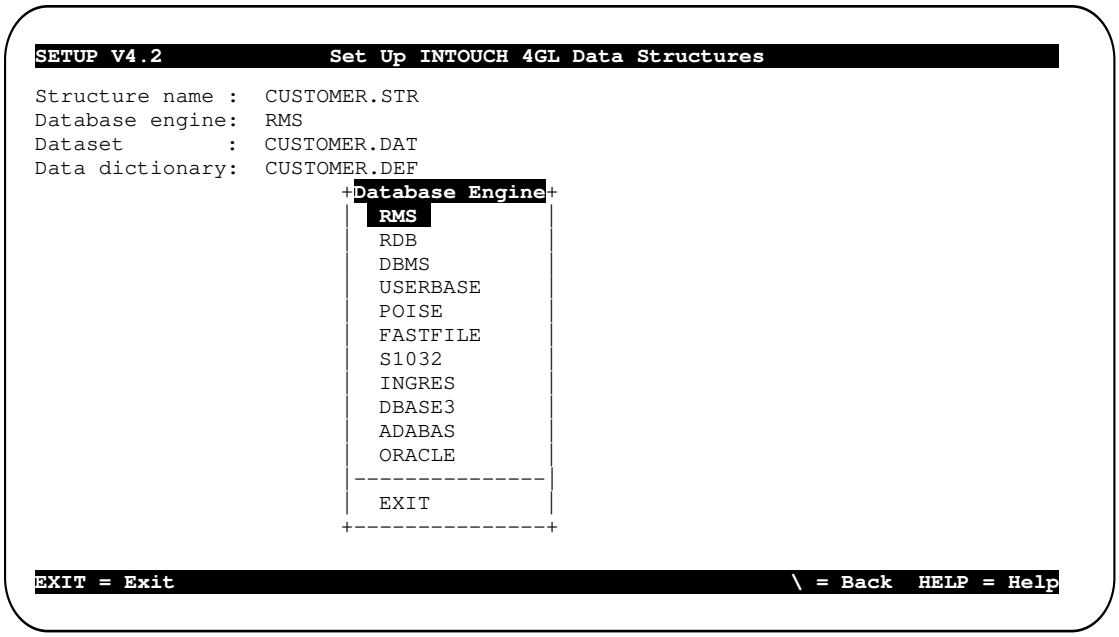
- change the general structure information.
- change the structure security level information and the read, write, update and delete access levels.



You can use the arrow keys to select the **Modify structure** procedure from the SETUP procedures menu and then select one of the two modify structure options.

1.5.1 Modify Structure General Information

When you select the modify structure **General information** option, SETUP will display the current structure information.



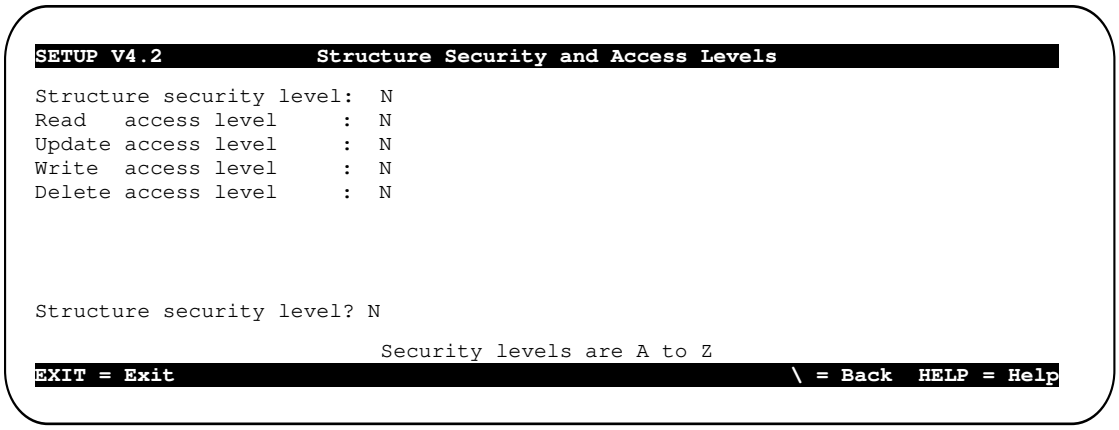
You will be prompted for: Database engine, Dateset, Data dictionary

You can either press Return at the prompts to keep the current information or enter new information. If you need clarification on any of the prompts, you can refer back to the **SETUP Prompts** section in this chapter.

After the last prompt is answered, the **STR** file will be updated with the new information. You will be returned to the SETUP procedures menu where you can select another option or exit SETUP.

1.5.2 Modify Structure Security Information

When you select the modify structure **Security information** option, SETUP will display the current security information.



You will be prompted for:

Structure security level information
Read, update, write and delete access level information

You can either press Return at the prompts to keep the current information or enter new information.

Structure security level

Enter the security level associated with this structure. The security level is a single letter in the range of **A** to **Z**. Security level **A** is the highest level of security. Security level **Z** is the lowest level of security.

Read, Update, Write, Delete access levels

Enter the access level required to *read* a data record in this structure. The *read access level* is a single letter in the range **A** to **Z**. **A** is the most restricted access level. **Z** is the least restricted.

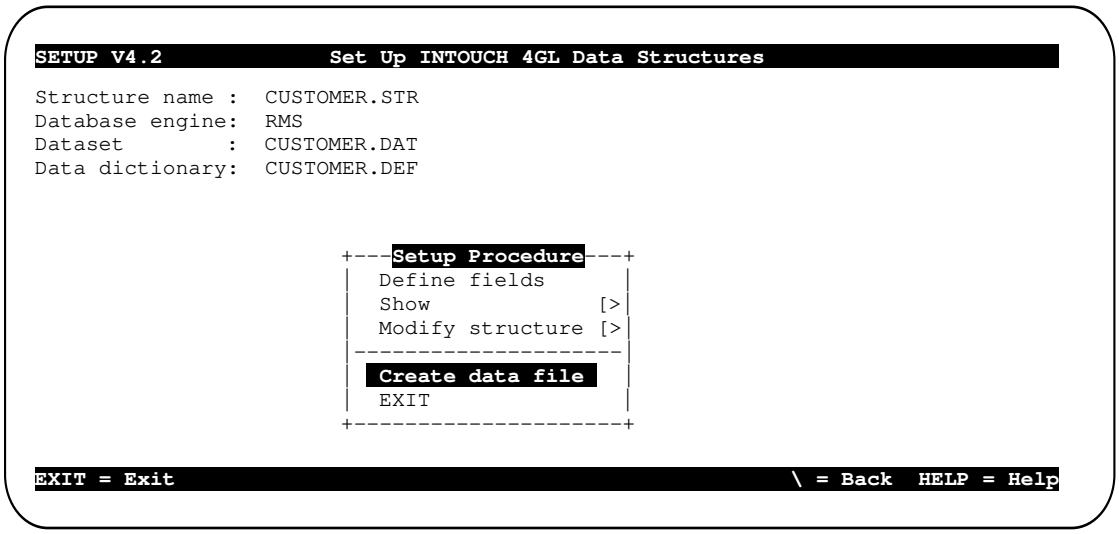
When an attempt is made to read a record from this structure, INTOUCH will compare the structure security level to the read access level. If the read access level is lower than the structure security level, the read request will be allowed. For example:

Structure level	Read access level	Access
K	R	Allowed
K	E	Denied
K	K	Allowed

Enter the access levels required to *update, write, delete* data records in this structure. Answer each of the prompts with a letter from **A** to **Z**.

After the last prompt is answered, the **STR** file will be updated with the new security information. You will be returned to the SETUP procedures menu where you can select another option or exit SETUP.

1.6 CREATE Procedure



Selecting the **Create data file** option from the SETUP procedures menu will allow you to create a data file. After the CREATE option has been selected, the create screen will be displayed and you will be prompted for some information.

Note

If the database engine is RMS, the following screen is displayed. If the database engine is not RMS, a similar screen will be displayed. The screen heading shows the database engine and name of the file that will be created.

Example 1–10: CREATE Procedure RMS Input Screen

```
SETUP V4.2            Creating RMS data file: CUSTOMER.DAT

File organization:
Recordsize       :


                      +RMS File Organization+
                      |  Indexed            |
                      |  Sequential         |
                      +---------------------+

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

RMS File Organization

Select how you want your file to be organized. The options are:

- **INDEXED** - the data file record contains one or more *key* fields
- **SEQUENTIAL** - the data file record contains *no* key fields

If you are going to be sorting and/or extracting data from the RMS file you are creating, the indexed file organization is more efficient than the sequential organization. This is especially true if you have large files.

An *indexed file* contains one or more indexed key fields which allow programs to locate and access file data very quickly without having to read through the file to locate records.

A *sequential file* has no key fields and accessing file data can be very slow and time consuming. Locating data in a sequential file requires reading every record in the file to find all occurrences of the data.

If you select **Indexed** file organization, you will be asked for the *key* fields in later prompts.

Record size

Enter the record size that you want for this data file. The default record size is 25% larger than the actual length of the fields defined. This allows room for adding new fields to the record without having to expand the record size.

The legal record sizes depend on the database engine (file management system) you use. For RMS, sizes range from 1 to approximately 32000.

Note

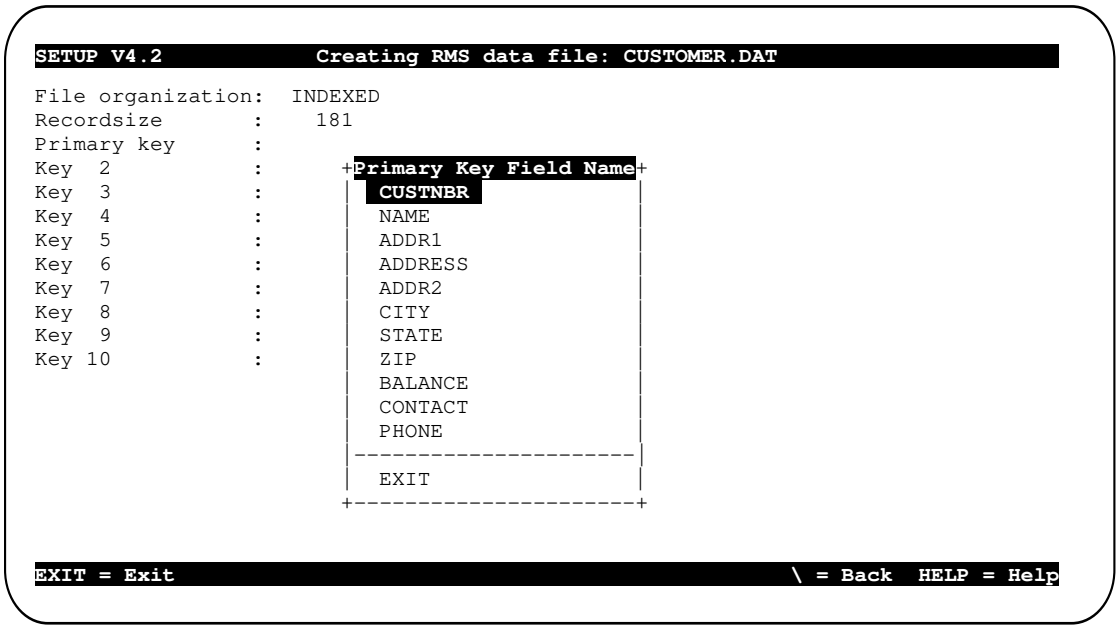
Depending on the database engine and file organization, SETUP might ask for additional information.

If no additional information is required to create this data file, you can enter "Y"es at the *Proceed with file creation* prompt and your data file will be created using the appropriate database engine.

Primary key field

Select the field from the list of defined fields, that you want to use as the primary key field. The first field is the default.

RMS indexed files require at least one key field, the *primary* key field.



Allow duplicates

You can control whether the data to be stored in the RMS key field will be *unique* (only one occurrence of the data in the key field) or allow *duplicates* (multiple occurrences of the data in the key field). Data access works most efficiently with unique key fields.

Normally, the PRIMARY key field is set up to allow only unique data in the field (i.e. NO duplicates) and the other key fields (ALTERNATE key fields) can be unique or not.

If you want more than one key field, select the fields you want to use as ALTERNATE key fields. Selecting **All keys defined** ends key field selection.

You will be asked if you want to proceed with file creation. Enter **Y** to create the data file. After the file has been created, you will be returned to the SETUP procedures menu.

1.7 Defining COBOL Record Fields

The **TTI_RUN:COBOL_DEF.INT** program allows COBOL users to define COBOL record fields. This program processes a text file that contains a COBOL record definition statement and creates an INTOUCH data dictionary (definition file).

The program expects the text file to contain a *single* COBOL record definition statement. The file must be in one of the following formats:

— An ANSI formatted file which contains line numbers in columns 1-6 and comments in columns 72-80.

— A TERMINAL formatted file which has no reserved columns at the beginning and end of the lines.

The file name or file specification that is to be defined can be up to 63 characters in length if the screen margin is set at 80. If the margin is set to 132, 115 characters are allowed.

1.7.1 Define Example

Here is an example of how to use the **TTI_RUN:COBOL_DEF** program. This example shows the input file and the created file. The sample text file COBOL_TEST.RECORD contains the following record definition statement:

```
01  test_rec.
  05  code pic x.
  05  desc pic x(20).
  05  amount pic s9(5)v99 comp.
  05  tbl  occurs 3 times.
      10  tbl-code pic x(5).
      10  tbl-desc pic x(25).
```

To run the program, enter the following at the "$" prompt:

$ **INTOUCH/SOURCE TTI_RUN:COBOL_DEF**

The program screen will be displayed:

```
COBOL_DEF V4.2   Create INTOUCH Definition from COBOL Definition

COBOL copy-lib:
ANSI format   :




COBOL copy-lib? _____________________________________________________________

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

At the "COBOL copy-lib?" prompt, enter the file name. In this example, the file name is **COBOL_TEST.RECORD**.

At the "Is file in ANSI format (Y/N)? NO__" prompt, press the Return key to take the "NO" default because the COBOL_TEST.RECORD file is not in ANSI format, it is in TERMINAL format.

To proceed, enter "Y" at the "Proceed with definition creation (Y/N)?" prompt. The program processes the COBOL_TEST.RECORD file and creates the COBOL_TEST.DEF file. You are returned to the "COBOL copy-lib?" prompt. Enter EXIT to exit the program.

Checking the Output

To see what the COBOL_TEST.DEF file looks like, you can:

1. Get into INTOUCH by entering **INTOUCH**
2. Enter **CALL SETUP** to activate the SETUP routine.
3. Answer the following prompts:

   Structure name : **COBOL_TEST**
   Database engine: **RMS**
   Dataset : **COBOL_TEST**
   Data dictionary: **COBOL_TEST**

   After the prompts are answered, a COBOL_TEST.STR file is created.
4. Use the arrow keys to select the **Show** menu option and the **Brief field display** option

The following definition listing is created:

```
11-Apr-1995              Definitions for: COBOL_TEST.DEF              Page:   1

Field Name       Description      First  Length  Occurs  Last  Dtype  Print Mask
---------------- ---------------- -----  ------  ------ -----  -----  ----------
TEST_REC         Test rec             1     115       1    115  CH     none
      Semantics  : UC

CODE             Code                 1       1       1      1  CH     none
      Semantics  : UC

DESC             Desc                 2      20       1     21  CH     none
      Semantics  : UC

AMOUNT           Amount              22       4       1     25  IN     none
      Semantics  : NUM:2, RJ, ZF

TBL_A1           Tbl a1              26      30       1     55  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A1      Tbl code a1         26       5       1     30  CH     none
      Semantics  : UC

TBL_DESC_A1      Tbl desc a1         31      25       1     55  CH     none
      Semantics  : UC

TBL_A2           Tbl a2              56      30       1     85  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A2      Tbl code a2         56       5       1     60  CH     none
      Semantics  : UC

TBL_DESC_A2      Tbl desc a2         61      25       1     85  CH     none
      Semantics  : UC

TBL_A3           Tbl a3              86      30       1    115  CH     none
      Semantics  : NUM, RJ, ZF

TBL_CODE_A3      Tbl code a3         86       5       1     90  CH     none
      Semantics  : UC

TBL_DESC_A3      Tbl desc a3         91      25       1    115  CH     none
      Semantics  : UC
```

2 INTOUCH Running Under VMS

2.1 File Specifications

Every file has a file specification. The file name is used to access the file. In INTOUCH, a legal file specification looks like this:

```
node::device:[directory]filename.ext;version
```

A file name and extension:

- can each be up to, but not longer than 39 characters.
- can consist of letters, numbers, underscores and hyphens.

The extension must be preceded by a period (.).

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 include the legal file specification according to the rules described in the *Command Language and DCL User's Guide* of the OpenVMS documentation set.

Note

You do not need to enter an extension with a file name in the INTOUCH commands. If you do not give an extension, INTOUCH uses the default extension ".INT".

3 INTOUCH Data Structures

3.1 General Description

INTOUCH has the ability to perform database operations as a transparent part of the language. These operations are performed on "fields" of given "records" within an INTOUCH structure.

A structure consists of three files: the structure file, the dataset or data file, and the data dictionary.

- The structure file contains the name of the data file, the database engine (record management system), the name of the data dictionary, and security and other information.
- The data file contains the actual data to be accessed.
- The data dictionary contains the field definitions and any other information required to access data in a given field.

A structure must be opened before it is accessed. Otherwise, an exception occurs. A structure is considered open if both the structure definition file and the data file are opened. The structure file contains the descriptions of these files, including the file names used to open them.

Record manipulation is performed as follows:

- A structure record is read, when the first access (either read or write) of a field in that record occurs.
- When a record is read, any previously locked record that has been modified is written to disk and unlocked. The new record is then read and unlocked.
- When you modify a record that is not currently locked, it is re-read and locked. A flag is set so the record will be written to disk before it is unlocked.
- When the structure is closed (explicitly by CLOSE, or implicitly by END), any locked records are written, as necessary, and unlocked.

Fields in the current record of a structure may be referenced as **structure-name(field)**.

Structure references can appear anywhere a normal variable can. In certain circumstances, treatment of the field is special. These are outlined below.

Upper and lower case letters are treated equivalently in a field name.

3.1.1 Using Structures within Expressions

As an expression is evaluated, INTOUCH keeps track of the type of the expression (numeric or string). When a structure(field) is referenced, INTOUCH makes sure that the field is the appropriate type.

If the field is compatible with the rest of the expression, then its contents are used. Otherwise, an exception is signaled.

3.1.2 Using PRINT with Fields

When printing a field from a structure, the field is printed using the edit mask from the structure definition file. The data from the field is unpacked if necessary and strings are formatted appropriately for output.

If the field is used within an expression, none of the above special formatting is done.

3.1.3 Field Expressions

To reference a field, you must indicate both the structure name and the expression of the field whose contents you want to read.

```
struc_name(field_expr)
```

The *field_expr* can be given either a string or numeric expression. You must specify a field by either: giving the field name as a string expression, or the field number as a numeric expression.

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

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

If you specify the field as a numeric expression or other than a string constant, you 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 interprets the characters as a field name.

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

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

3.1.4 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.
```

3.2 Structure Statements

3.2.1 OPEN STRUCTURE

FORMAT:

```
OPEN STRUCTURE struc_name: NAME 'filename'
    [, ACCESS INPUT | OUTIN] [, LOCK]
    [, DATAFILE filename] [, OPTIMIZE OFF]
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', &
      ACCESS INPUT, LOCK
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California clients by last name'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END
```

```
RNH
List of California clients by last name
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

SUMMARY:

OPEN STRUCTURE opens an INTOUCH structure. You must open a structure before you reference fields, change field data or add records to the structure. *struc_name* is the name you associate with the structure (similar to a channel number). **NAME** is the actual name of the structure file. **ACCESS INPUT** opens the structure for input only. ACCESS INPUT is the default OPEN mode for structures. **ACCESS OUTIN** opens the structure for output and input. The **LOCK** option causes INTOUCH to lock the structure from writing access by others. This can have the effect of speeding up INTOUCH's structure statements (EXTRACTs especially). The **DATAFILE** option overrides the default datafile as specified by the structure.

3.2.2 CLOSE STRUCTURE

FORMAT:

```
CLOSE STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California Clients'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

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

SUMMARY:

CLOSE STRUCTURE closes a structure from further access. You should close your structure before your program ends. You can use the **CLOSE ALL** statement to close *all* open structures and other files.

3.2.3 ADD STRUCTURE

FORMAT:

```
ADD STRUCTURE struc_name
        ---
  [LET] struc_name(field) = expr
        ---
END ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  INPUT 'Enter ID number': id%
    INPUT 'Enter last name': last$
    INPUT 'Enter first name': first$
    INPUT 'Enter state': state$
    INPUT 'Enter phone': phone$
30  ADD STRUCTURE cl
      PRINT
      PRINT 'Adding '; last$; ', '; first$
      LET cl(id) = id%
      LET cl(last) = last$
      LET cl(first) = first$
      LET cl(state) = state$
      LET cl(phone) = phone$
    END ADD
40  CLOSE STRUCTURE cl
50  END
```

```
RNH
Enter ID number? 12233
Enter last name? Jones
Enter first name? Tom
Enter state? NV
Enter phone? 2345556161

Adding Jones, Tom
```

SUMMARY:

The **ADD STRUCTURE** construct is used to add a record to the structure. **LET** statements assign values to the fields. INTOUCH writes the record when the **END ADD** is executed.

3.2.4 CANCEL ADD

FORMAT:

```
CANCEL ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
    ADD STRUCTURE cl
      INPUT 'Client ID': cl(id)
      IF  _EXIT  THEN  CANCEL ADD
      INPUT 'Last name': cl(last)
      IF  _EXIT  THEN  CANCEL ADD
      INPUT 'First name': cl(first)
      IF  _EXIT  THEN  CANCEL ADD
      PRINT 'Adding client'
    END ADD
20  END

RNH
Client ID? 14422
Last name? WHITE
First name? EXIT
```

SUMMARY:

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.

3.2.5 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
```

SUMMARY:

EXIT ADD transfers control immediately to the corresponding END ADD statement *and* performs the add.

3.2.6 DELETE STRUCTURE

FORMAT:

```
DELETE STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
30  ! Delete all clients from California
    FOR EACH cl
      PRINT 'Deleting '; cl(first); ' '; cl(last) ;'...';
      DELETE STRUCTURE cl
      PRINT 'record deleted'
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
Deleting Earl Errant...record deleted
Deleting Kathy Cass...record deleted
Deleting Dale Derringer...record deleted
```

SUMMARY:

DELETE STRUCTURE is used to delete the current record from a structure.

3.2.7 LOCK | UNLOCK STRUCTURE

FORMAT:

```
LOCK | UNLOCK STRUCTURE struc_name
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS OUTIN
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
30  FOR EACH cl
      PRINT
      PRINT cl(first); ' '; cl(last)
      LOCK STRUCTURE cl   ! Give us exclusive access
      LINE INPUT DEFAULT cl(phone), PROMPT 'Enter new phone ': phone$
      IF  _EXIT  THEN  EXIT FOR
      cl(phone) = phone$
      UNLOCK STRUCTURE cl ! Put the record out to disk
                          ! and release it
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH

Keith Kent
Enter new phone 6199675021

Paul Johnson
Enter new phone EXIT
```

SUMMARY:

INTOUCH automatically locks and unlocks structures when you work with them. You can use the **LOCK STRUCTURE** and **UNLOCK STRUCTURE** statements to manually control this process.

3.2.8 EXTRACT STRUCTURE

FORMAT:

```
EXTRACT STRUCTURE struc_name
        ---
        ---  block of code
        ---
END EXTRACT

           or

EXTRACT STRUCTURE struc_name [:KEY field=expr1 [TO expr2]]
     [INCLUDE | EXCLUDE] cond_expr
     [SORT [ASCENDING | DESCENDING] BY expression]

      ...

END EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California Clients'
    PRINT
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(state)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of California Clients

Dale Derringer      CA
Earl Errant         CA
```

SUMMARY:

EXTRACT STRUCTURE is used to create a list of structure records. You can then work with the records in this list. **KEY** causes INTOUCH to use the structure key list to search for the key specified. KEY with the **TO** option causes INTOUCH to search for all keys in the range specified. **INCLUDE** causes INTOUCH to include the specified records. **EXCLUDE** causes INTOUCH to exclude the specified records. **SORT** causes INTOUCH to sort the list in **ASCENDING** or **DESCENDING** order according to the specified field.

3.2.9 REEXTRACT STRUCTURE

FORMAT:

```
REEXTRACT STRUCTURE struc_name
       ---
       ---  block of code
       ---
END EXTRACT

            or

REEXTRACT STRUCTURE struc_name
  [INCLUDE | EXCLUDE] cond_expr
  [SORT [ASCENDING | DESCENDING] BY expression

   ...

END EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
    END EXTRACT
    REEXTRACT STRUCTURE cl
      EXCLUDE cl(phone)[1:3] <> '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
30  PRINT 'List of California Clients in Area Code 619'
    FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50  END

RNH
List of California Clients in Area Code 619
Cathy Cass          (619) 743-8582
Paul Johnson        (619) 489-5551
Keith Kent          (619) 967-5021
Pete Porter         (619) 778-6709
Wayne Waters        (619) 564-1231
```

SUMMARY:

REEXTRACT STRUCTURE is used to do a second extract on the previous extract list. This lets you choose more and more specific records through a series of REEXTRACTs. It functions exactly like EXTRACT, except it chooses its records from the existing extract list.

Note

Extract operations by key cannot be performed with REEXTRACT.

3.2.10 FOR EACH

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
      INCLUDE cl(state) = 'CA'
      EXCLUDE cl(phone)[1:3] = '619'
      SORT ASCENDING BY cl(last)
    END EXTRACT
    PRINT 'List of California Clients'
    PRINT
30  FOR EACH cl
      PRINT cl(first); ' '; cl(last), cl(phone)
    NEXT cl
40  CLOSE STRUCTURE cl
50 END

RNH
List of California Clients

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

SUMMARY:

FOR EACH is used to execute a block of code for each record in the extract list. **NEXT** marks the end of the loop. Use FOR EACH to manipulate the structure information in your program. The **REPEAT**, **ITERATE** and **EXIT FOR** statements can also be used in FOR EACH loops.

3.2.11 CANCEL EXTRACT

FORMAT:

```
CANCEL EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
    EXTRACT STRUCTURE cl
      PRINT 'Client: '; cl(last)
      LINE INPUT 'Press return to continue': z$
      IF  _EXIT  THEN  CANCEL EXTRACT
    END EXTRACT
    PRINT 'Records extracted:'; _EXTRACTED
20  CLOSE STRUCTURE cl
30  END

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

SUMMARY:

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

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

3.2.12 EXIT EXTRACT

FORMAT:

```
EXIT EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
      PRINT 'Client: '; cl(last)
      LINE INPUT 'Press return to continue': z$
      IF  _EXIT  THEN  EXIT EXTRACT
    END EXTRACT
    PRINT 'Records extracted:'; _EXTRACTED
20  END

RNH
Client: Smith
Press return to continue? <Return>
Client: Kent
Press return to continue? EXIT
Records extracted: 1
```

SUMMARY:

EXIT EXTRACT passes control to the corresponding END EXTRACT statement, performs final sorting (if any), and creates the extracted collection.

3.2.13 EXTRACT STRUCTURE: KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
20  EXTRACT STRUCTURE cl: KEY id = 80561
      PRINT 'Client:',
      PRINT cl(first); ' '; cl(last), cl(id)
    END EXTRACT
    CLOSE STRUCTURE cl
30  END

RNH
Client:            Dale Derringer        80561
```

SUMMARY:

The **KEY** option includes records using the record's *key*. The key is a field which has an index for fast access. The key option can considerably speed up extractions.

The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

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.

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
20  INPUT 'Enter the lowest ID to check': lowest
    INPUT 'Enter the highest ID to check': highest
    EXTRACT STRUCTURE cl: KEY id = lowest TO highest
      PRINT cl(id); TAB(10); cl(last)
    END EXTRACT
    CLOSE STRUCTURE cl
30  END

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

3.2.14 EXTRACT STRUCTURE: PARTIAL KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'TTI_RUN:CLIENT'
    EXTRACT STRUCTURE cl, FIELD last: PARTIAL KEY 'Ros'
    END EXTRACT
    PRINT 'List of clients with last name starting with Ros'
    PRINT
    FOR EACH cl
      PRINT cl(first); ' '; cl(last)
    NEXT cl
20  CLOSE STRUCTURE cl
30  END

RNH
Bud Roske
Earl Rost
Dale Rosty
```

SUMMARY:

PARTIAL KEY is used in the EXTRACT STRUCTURE to look for part of a key value. This statement finds all records with a key value starting with *str_expr*.

3.2.15 EXTRACT STRUCTURE: APPEND

FORMAT:

```
EXTRACT STRUCTURE struc_name : APPEND
```

EXAMPLE:

```
10  OPEN STRUCTURE detail : name 'tti_run:detail'
    SET STRUCTURE detail : EXTRACTED 0
20  EXTRACT STRUCTURE detail, FIELD lineid : &
        KEY '10301001' TO '10301999', APPEND
      SORT BY detail(prodnbr)
      SORT BY detail(invnbr)
    END EXTRACT
30  EXTRACT STRUCTURE detail, field lineid : &
        KEY '10311001' to '10311999', APPEND
      SORT BY detail(prodnbr)
      SORT BY detail(invnbr)
    END EXTRACT
    PRINT 'Prod'; TAB(7); 'Line ID'; TAB(17); 'Qty'
40  FOR EACH detail
      PRINT detail(prodnbr); TAB(7); detail(lineid); &
            TAB(17); detail(qty)
    NEXT detail
50  END
```

```
RNH
Prod  Line ID   Qty
22800 10301-002      2
22800 10301-004      1
22800 10301-006      2
24100 10311-003      1
24200 10301-003      1
24200 10311-009      1
28400 10311-001      2
28800 10301-009      2
28800 10311-002      9
28800 10311-005      1
28800 10311-006      1
31020 10301-005      1
31040 10311-010      2
31150 10301-001      1
31150 10301-008      8
31150 10311-004      1
31150 10311-008      1
33090 10301-007      2
33090 10311-007      1
```

SUMMARY:

The **EXTRACT STRUCTURE: APPEND** statement adds records to the last collection of extracted records rather than creating a new collection.

3.2.16 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
```

SUMMARY:

Used for DBMS handling, this statement fetches all structures (*struc_name1*) owned by another structure (*struct_name2*) within a given set.

3.2.17 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
```

SUMMARY:

Used for DBMS handling, this statement allows you to extract all structures specified by *struc_name1* owned by the current record within the given set (*set_name*) with field a *field_expr* equal to the given key.

3.3 ASK STRUCTURE

ASK STRUCTURE allows you to ask about various device and structure characteristics from within your programs. After you do an ASK STRUCTURE option, you can do a SET STRUCTURE option in many cases. The SET STRUCTURE options are explained in the next section. The format and description of the various ASK STRUCTURE options are explained in this section.

3.3.1 ASK STRUCTURE: CURRENT

FORMAT:

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

EXAMPLE:

```
10  DIM a$(100)
    LET i = 0
20  OPEN STRUCTURE cl: NAME 'tti_run:client'
30  EXTRACT STRUCTURE cl
    END EXTRACT
40  FOR EACH cl
      PRINT cl(last); ', '; cl(first)
      INPUT 'Would you like to see this record (Y/N)': yn$
      IF  yn$ = 'Y'  THEN
        LET i = i + 1
        ASK STRUCTURE cl: CURRENT a$(i)
      END IF
    NEXT cl
50  PRINT
    FOR j = 1 TO i
      SET STRUCTURE cl: CURRENT a$(j)
      PRINT cl(last); ','; cl(first), cl(state), cl(phone)
    NEXT j
60  END
```

```
RNH
Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

SUMMARY:

The **CURRENT** option assigns the current record to the *str_var*. The *str_var* is assigned the concept of current record for the record management system you are using. INTOUCH returns a zero length string if there is no CURRENT record.

3.3.2 ASK STRUCTURE: FIELDS

FORMAT:

```
ASK STRUCTURE struc_name: FIELDS num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: FIELDS z
    PRINT z
20  END
```

```
RNH
 18
```

SUMMARY:

The **FIELDS** option returns a number specifying the number of fields in the structure. This number is assigned to the numeric variable specified.

3.3.3 ASK STRUCTURE: KEYS

FORMAT:

```
ASK STRUCTURE struc_name: KEYS num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: KEYS z
    PRINT z
20  END

RNH
 2
```

SUMMARY:

KEYS returns the number of available key fields. It returns the value of 0 if no keys are available.

3.3.4 ASK STRUCTURE FIELD: item

FORMAT:

```
ASK STRUCTURE struc_name, FIELD field_expr: ITEM var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  ASK STRUCTURE cl: FIELDS num_fields
    FOR i = 1 TO num_fields
      CLEAR
      ASK STRUCTURE cl, FIELD #i: DESCRIPTION b$
      ASK STRUCTURE cl, FIELD #i: PROMPT a$
      ASK STRUCTURE cl, FIELD #i: POSITION a%
      ASK STRUCTURE cl, FIELD #i: LENGTH b%
      ASK STRUCTURE cl, FIELD #i: HEADING f$
      ASK STRUCTURE cl, FIELD #i: PRINTMASK c$, &
                                  SCREENMASK d$, &
                                  HELP e$
      PRINT AT 5,5: ''
      PRINT 'Description   : '; b$
      PRINT 'Prompt        : '; a$
      PRINT 'Position      : '; a%
      PRINT 'Field length  :' ; b%
      PRINT 'Rpt. heading  : '; f$
      PRINT 'Print mask    : '; c$
      PRINT 'Screen mask   : '; d$
      PRINT 'Help          : '; e$
      DELAY 5
    NEXT i
    CLOSE STRUCTURE cl
30  END
```

```
RNH
Description   : Client ID number
Prompt        : Client ID number
Position      :  1
Field length  : 5
Rpt. heading  : CLNT,ID
Print mask    : >####
Screen mask   : digits:#####
Help          : Client ID number
```

SUMMARY:

The **FIELD** option allows you to get information about a specific field in a structure. The following items are available with ASK STRUCTURE FIELD:

- ACCESS str_var

  Returns the access rules of this field in the string variable specified.

- APPLICATION str_var

  Returns a name of an application for this structure and field in the string variable specified.

- ATTRIBUTES str_var

  Returns the INTOUCH attributes NUM (number), UC (upper-case), etc., in the string variable specified.

- CHANGEABLE num_var

Returns a value of TRUE or FALSE in the numeric variable specified. If the field can be changed, the value is TRUE. If the field cannot be changed, the value is FALSE.

- CLASSIFICATION str_var

  Returns the classification for this field in the string variable specified.

- DATATYPE str_var

  Returns the INTOUCH data type such as CH (character), IN (integer), etc., in the string variable specified.

- DESCRIPTION str_var

  Returns the description for this field in the string variable specified.

- HEADING str_var

  Returns the heading for this field in the string variable specified.

- HELP str_var

  Returns the help text for this field in the string variable specified.

- KEYED num_var

  Returns a value of TRUE or FALSE in the numeric variable specified. If it is a keyed field, the value is TRUE; otherwise, it is FALSE.

- LENGTH num_var

  Returns the length of this field in the numeric variable specified.

- NAME str_var

  Returns the name of the field in the string variable specified.

- NULL num_var

  Returns a value of TRUE or FALSE in the numeric variable specified. If a field is null, the value is TRUE. If the field is not null, the value is FALSE.

- NUMBER num_var

  Returns the field number of this field in the numeric variable specified. Fields are numbered sequentially. If the field does not exist, a value of 0 will be returned.

- OPTIMIZED num_var

  Returns a value of TRUE or FALSE in the numeric variable specified. If the key field is optimized, the value is TRUE; otherwise, the value is FALSE.

- POSITION num_var

  Returns the starting position for this field in the numeric variable specified.

- PRINTMASK str_var

  Returns the print mask for this field in the string variable specified.

- PROMPT str_var

  Returns the prompt for this field in the string variable specified.

- SCREENMASK str_var

  Returns the screen mask for this field in the string variable specified.

- VRULES str_var

  Returns the validation rules for this field in the string variable specified. You can refer to the "VALID" function item in Section A.2, Other Functions for information on validation rules.

3.3.5 ASK STRUCTURE: CAPABILITY

FORMAT:

```
ASK STRUCTURE struc_name: CAPABILITY str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client', ACCESS INPUT
    ASK STRUCTURE cl: CAPABILITY z$
    PRINT z$
20  END

RNH
INDEXED,INPUT
```

SUMMARY:

Given a structure expression, **ASK STRUCTURE: CAPABILITY** stores in *str_var* a comma delimited string containing one or more of the following values:

- *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.

3.3.6 ASK STRUCTURE: ENGINE

FORMAT:

```
ASK STRUCTURE struc_name : ENGINE str_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl : name 'tti_run:vendor'
20  ASK STRUCTURE  cl : ENGINE ename$
30  PRINT 'DATABASE ENGINE is: '; ename$
40  END

RNH
DATABASE ENGINE is: RMS
```

SUMMARY:

The **ASK STRUCTURE: ENGINE** statement returns the name of the database engine being used for the specified structure in *str_var*.

3.3.7 ASK STRUCTURE: EXTRACTED

FORMAT:

```
ASK STRUCTURE struc_name: EXTRACTED num_var
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    EXTRACT STRUCTURE cl
    END EXTRACT
    ASK STRUCTURE cl: EXTRACTED z
    PRINT 'Records found: '; z
20  END

RNH
Records found:  13
```

SUMMARY:

ASK STRUCTURE: EXTRACTED asks the operating system for the last extracted count for the specified structure.

3.3.8 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
```

SUMMARY:

This statement asks the operating system for the ID of the structure and stores it into the string variable given.

3.3.9 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
```

SUMMARY:

From within a FOR EACH...NEXT struc_name block, **ASK STRUCTURE: POINTER** asks the structure for the number of the current pointer.

3.3.10 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
```

SUMMARY:

ASK STRUCTURE: RECORD asks for the relative record number of the current record and places that number in the numeric variable given.

3.3.11 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
```

SUMMARY:

This statement returns the record size of the structure data file.

3.3.12 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
```

SUMMARY:

This statement retrieves the access rules for the specified structure. Security level, data file read, write, update, and delete rules are returned.

3.3.13 ASK STRUCTURE: TIMEOUT

FORMAT:

```
ASK STRUCTURE str_name: TIMEOUT int_var
```

SUMMARY:

This statement asks for the current TIMEOUT setting.

The **SET STRUCTURE: TIMEOUT** statement is used when a record is locked for over the *int_expr*. INTOUCH then returns a TIMEOUT error.

3.4 ASK | SET STRUCTURE #string_expr

FORMAT:

```
ASK STRUCTURE #string_expr. . .
SET STRUCTURE #string_expr. . .
```

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    str$ = 'CL'
    fld$ = 'ID'
    do_work
    STOP

20  ROUTINE do_work
    ASK STRUCTURE #str$, FIELD #fld$: DESCRIPTION dsc$
    PRINT 'Description is: '; dsc$
    END ROUTINE

30  END

RNH
Description is: Client ID number
```

SUMMARY:

You can use a string expression for the structure name in an ASK STRUCTURE or SET STRUCTURE statement. This allows you to write generalized code to work for several structures.

3.5 SET STRUCTURE

SET STRUCTURE allows structure manipulation. You can use the SET structure statement to change various device and structure characteristics of INTOUCH structures. The format and description of the various SET STRUCTURE options are explained in this section.

3.5.1 SET STRUCTURE: EXTRACTED 0

FORMAT:

```
SET STRUCTURE struc_name: EXTRACTED 0
```

EXAMPLE:

```
10  OPEN STRUCTURE vend: NAME 'tti_run:vendor'
    SET STRUCTURE vend: EXTRACTED 0
20  END
```

SUMMARY:

Setting the number of records extracted to zero causes a new collection to be started. The **SET STRUCTURE: EXTRACTED 0** statement is used in conjunction with the EXTRACT STRUCTURE: APPEND statement.

3.5.2 SET STRUCTURE: CURRENT

FORMAT:

```
SET STRUCTURE struc_name: CURRENT str_var
```

EXAMPLE:

```
10  DIM a$(100)
    LET i = 0
20  OPEN STRUCTURE cl: NAME 'tti_run:client'
30  EXTRACT STRUCTURE cl
    END EXTRACT
40  FOR EACH cl
      PRINT cl(last); ', '; cl(first)
      INPUT 'Would you like to see this record (Y/N)': yn$
      IF  yn$ = 'Y'  THEN
        LET i = i + 1
        ASK STRUCTURE cl: CURRENT a$(i)
      END IF
    NEXT cl
50  PRINT
    FOR j = 1 TO i
      SET STRUCTURE cl: CURRENT a$(j)
      PRINT cl(last); ','; cl(first), cl(state), cl(phone)
    NEXT j
60  END
```

```
RNH
Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

SUMMARY:

The **CURRENT** option sets the current record to the value specified by the string variable *str_var*.

If you use a null string for the value, INTOUCH sets the structure to no current record.

```
SET STRUCTURE struc_name: CURRENT ''
```

3.5.3 SET STRUCTURE FIELD:KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    LINE INPUT 'Enter an ID': id$
    SET STRUCTURE cl, FIELD id: KEY id$
    IF  _EXTRACTED = 0  THEN
       MESSAGE ERROR: 'Not found'
    ELSE
       PRINT cl(id), cl(last)
    END IF
20  END

RNH
Enter an ID? 80561
80561               Derringer
```

SUMMARY:

The **FIELD** option lets you specify a field in a structure. The FIELD option is currently used in conjunction with the **KEY** and PARTIAL KEY options. The *field* name must be specified exactly as it appears in the structure.

The KEY option specifies the key to look for. The unique key is contained in the string expression *str_expr*. The KEY must be a string expression.

_EXTRACTED contains the number of records extracted. If no records are extracted, _EXTRACTED will be 0 and an error message will be displayed.

3.5.4 SET STRUCTURE FIELD: PARTIAL KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
    INPUT 'Name': name$
    SET STRUCTURE cl, FIELD last: PARTIAL KEY name$
    PRINT cl(id); ' '; cl(last)
20  END

RNH
Name? D
80561 Derringer
```

SUMMARY:

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

3.5.5 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
```

SUMMARY:

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 above). By using these statements, you can write generalized routines when you do not know which structure you are going to access until run time.

3.5.6 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
```

SUMMARY:

This statement sets the structure to the nth (*num_expr*) 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.

3.5.7 SET STRUCTURE: RECORD

FORMAT:

```
SET STRUCTURE struc_name: RECORD num_expr
```

EXAMPLE:

```
10  OPEN STRUCTURE ml: NAME 'tti_run:maint_log'
    SET STRUCTURE ml: RECORD 3
    PRINT ml(cost)
20  END

RNH
    $370.00
```

SUMMARY:

SET STRUCTURE: RECORD sets the structure to the record number given. The statement works only for structures that support relative record access.

3.5.8 SET STRUCTURE: TIMEOUT

FORMAT:

```
SET STRUCTURE str_name: TIMEOUT int_expr
```

SUMMARY:

If a record is locked for over *int_expr*, INTOUCH returns a **TIMEOUT** error.

3.5.9 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
```

SUMMARY:

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

3.5.10 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
```

SUMMARY:

This statement used for DBMS handling, fetches the owner of the structure specified by *struc_name1*. The structure specified by *struc_name2* is the owner of *struc_name1* with the set given in *set_name*.

3.5.11 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
```

SUMMARY:

This statement is used for DBMS handling.

3.5.12 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
```

SUMMARY:

This statement fetches the first *struc_name1* record using the key given in *str_expr* within the set, *set_name*, where *struc_name2* is the owner.

4 DEBUG System

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's TRACE and STATISTICS features can be used to monitor your program's execution. These features take effect only during program execution.

TRACE traces your program line by line and lists the number of each 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.

4.1 DEBUG ON/OFF

4.1.1 DEBUG ON

FORMAT:

```
DEBUG ON
```

EXAMPLE:

```
DEBUG ON              (from INTOUCH prompt)
1050 DEBUG ON         (from within program)
```

SUMMARY:

DEBUG ON lets you use INTOUCH's debug system. The DEBUG system helps you find problems in the way your program runs. DEBUG ON enables the system. It can be used as a statement in your program or as a command at the INTOUCH prompt.

DEBUG ON turns on the frame if it was off.

4.1.2 DEBUG OFF

FORMAT:

```
DEBUG OFF
```

EXAMPLE:

```
DEBUG OFF
1050 DEBUG OFF
```

SUMMARY:

DEBUG OFF disables INTOUCH's DEBUG system. You should turn DEBUG OFF when you have finished correcting your program. DEBUG OFF can be used as a statement in your program or as a command at the INTOUCH prompt.

4.2 The TRACE Feature

4.2.1 TRACE ON

FORMAT:

```
TRACE ON
```

EXAMPLE:

```
TRACE ON
1050 TRACE ON
```

SUMMARY:

Use **TRACE ON** to turn INTOUCH's trace feature on. The trace feature lists each program line number, as it is executed, at the bottom of the screen. That way you can tell 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.

4.2.2 TRACE OFF

FORMAT:

```
TRACE OFF
```

EXAMPLE:

```
TRACE OFF
1050 TRACE OFF
```

SUMMARY:

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

4.2.3 TRACE Feature Example

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 statement. The following program is executed with the trace feature on:

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

Example 4–1: Debug TRACE ON Example

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


INTOUCH

TRACE ON

INTOUCH

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

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

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

4.3 Statistics Feature

4.3.1 STATS ON

FORMAT:

```
STATS ON
```

EXAMPLE:

```
STATS ON
1020 STATS ON
```

SUMMARY:

STATS ON turns on the statistics feature. STATISTICS stores the execution time and number of times each program line is executed. That way you can tell if your statements are being executed the correct number of times, and which parts of your program are taking longest to execute. STATS is especially useful for locating bottleneck areas in application programs.

4.3.2 STATS OFF

FORMAT:

```
STATS OFF
```

EXAMPLE:

```
STATS OFF
1020 STATS OFF
```

SUMMARY:

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

4.3.3 LIST STATS

FORMAT:

```
LIST STATS
```

EXAMPLE:

```
LIST STATS
```

SUMMARY:

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

Example 4–2: Listing Program Statistics

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

INTOUCH

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

INTOUCH


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

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

4.4 BREAK and STEP Features

4.4.1 BREAK

FORMAT:

```
BREAK
```

EXAMPLE:

```
1050 BREAK
```

SUMMARY:

Use **BREAK** to interrupt program execution. For example, you might use BREAK to stop program execution if a variable is assigned an illegal value.

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

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

Example 4–3: BREAK in DEBUG System

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

INTOUCH

RNH
1                   2                   BREAK at 10.2

INTOUCH

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

4.4.2 STEP

FORMAT:

```
STEP [number]
```

EXAMPLE:

```
STEP
STEP 10
```

SUMMARY:

STEP executes a specific number of program statements and then stops execution. That way, you can "step through your program" to find bugs. If no number is specified, the default is one. For example:

Example 4–4: STEP Example in DEBUG System

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

RNH
1                    2                   BREAK at 10.2

INTOUCH

STEP 2
3                    4
INTOUCH


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

4.4.3 GO

FORMAT:

```
GO
```

EXAMPLE:

```
GO
```

SUMMARY:

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 STOP or BREAK statement was used, execution will resume at the first statement after the STOP or BREAK.

Example 4–5: Using GO in DEBUG System

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

INTOUCH

PRINT SQR(i)
 2

INTOUCH

25  PRINT i;
GO
 4  5  6
INTOUCH

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

5 INTOUCH COMMANDS

5.1 The EXIT, HELP and Backup Commands

At the bottom of the INTOUCH screen, a summary message appears. The commands described at the bottom of the screen can be used at any time while you are in INTOUCH. These universal commands are:

EXIT

You can exit an INTOUCH procedure, routine or the INTOUCH environment by typing **EXIT** at an available prompt.

Typing EXIT at the INTOUCH prompt will cause you to exit INTOUCH and return to the system prompt (usually $). Typing EXIT, when in an INTOUCH routine will cause one of the following:

- exit the routine and return to the INTOUCH prompt.
- exit a procedure within the routine and return to the previous procedure.

HELP

You can get help by typing **HELP**. Typing HELP at the INTOUCH prompt or in an INTOUCH routine will provide you with helpful text information.

INTOUCH has an extensive library of HELP text on many topics. Typing HELP by itself will cause INTOUCH to display some information, a list of topics and the prompt "Topic?". When a topic is selected, INTOUCH provides information about that topic. INTOUCH can display a list of sub-topics and the prompt "Subtopic?". This allows you to get more detailed information.

To ask for specific information, you can enter:

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

Pressing the Return key causes INTOUCH to exit a topic level. Pressing the Return key at the "Topic?" prompt will cause INTOUCH to exit the HELP facility.

\

When in an INTOUCH routine, you can use the \ (backslash) to back up to a previous prompt or menu.

5.2 The INTOUCH Commands

The following sections describe many of the INTOUCH commands.

5.2.1 RUN

FORMAT:

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

EXAMPLE:

```
RUN 'display_name'
    or
RNH 'display_name'
```

SUMMARY:

The **RUN** command is used to execute a program after it has been written. RUN executes the program specified. If no file specification is given, it executes the program in current memory. **RNH** will run the INTOUCH program without displaying the program header. Other than suppressing the header, RNH operates the same as the RUN command.

5.2.2 NEW

FORMAT:

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

EXAMPLE:

```
NEW 'display_name'
```

SUMMARY:

To write a new program, enter **NEW** followed by the name of the new program. NEW clears current memory and assigns a name to the current workspace. If no name is specified, INTOUCH uses the default "NONAME". If you specify a file name with an extension, INTOUCH ignores the extension. The workspace name can be up 19 characters in length and consist of letters, numbers and underscore characters.

5.2.3 SAVE

FORMAT:

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

EXAMPLE:

```
SAVE 'display_name'
```

SUMMARY:

After a program has been written, the **SAVE** command saves it for later use. SAVE is used to save original files. An original file is one which has a unique file specification. Typing SAVE by itself causes INTOUCH to save your file under the current workspace name with an extension of **.INT**.

5.2.4 REPLACE

FORMAT:

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

EXAMPLE:

```
REPLACE 'display_name'
```

SUMMARY:

After you make changes in a program, use **REPLACE** to save the changed version. The REPLACE command replaces the latest version of your file with the version in current memory. If the file does not exist, INTOUCH creates a new file with the name you specify.

5.2.5 ABORT

FORMAT:

```
ABORT [int_expr]
```

EXAMPLE:

```
ABORT
```

SUMMARY:

ABORT aborts INTOUCH without saving your latest changes. This statement passes the ABORT status to the operating system. In this way, INTOUCH applications can force a command procedure to abort.

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

```
ABORT 200
```

5.2.6 OLD

FORMAT:

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

EXAMPLE:

```
OLD 'display_name'
```

SUMMARY:

Use the **OLD** command to load a program you have saved. OLD searches for the file specified and loads it into current memory.

5.2.7 UNSAVE

FORMAT:

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

EXAMPLE:

```
UNSAVE 'display_name'
```

SUMMARY:

Use the **UNSAVE** command to delete programs you no longer want. UNSAVE deletes or unsaves the specified file. If no extension is given, INTOUCH uses the default extension **.INT**.

5.2.8 LIST

FORMAT:

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

EXAMPLE:

```
LIST 300, 510-570, do_print
LNH 300, 510-570, do_print
```

SUMMARY:

Use the **LIST** command to display all or part of your program code. LIST displays lines from the program in current memory. **LNH** works the same as LIST except the program header, time and date are suppressed. Typing LIST by itself will cause all of the program lines in current memory to be displayed. You can list specific sections of a file by referencing the program's line numbers, labels and routines.

5.2.9 LIST ERROR

FORMAT:

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

EXAMPLE:

```
LIST ERROR
LNH ERROR
```

SUMMARY:

LIST ERROR lists compile-time errors to the terminal. LIST ERROR displays the line number and the source code of all lines with errors. **LNH ERROR** works the same as LIST ERROR except that the program header, date and time are suppressed. The options available with LIST are available with this command.

5.2.10 EDIT

FORMAT:

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

EXAMPLE:

```
EDIT 510-570, do_print
```

SUMMARY:

EDIT transfers control to the editor. The entire program or a specified part of it is placed in the editor. When the program is exited, you return to the INTOUCH prompt. Typing EDIT by itself will cause all of the program lines in current memory to be placed in the editor. You can edit specific sections of a file by referencing the program's line numbers, labels and routines. Only the sections you reference will be moved into the editor.

5.2.11 RENUMBER

FORMAT:

```
RENUMBER
```

EXAMPLE:

```
RENUMBER
```

SUMMARY:

RENUMBER operates on the program in current memory. This command renumbers or resequences the program in current memory. It renumbers the first line number to 1000. Subsequent line numbers are incremented by 10 and all references to the line numbers (GOTOs, GOSUBs, etc.) are changed to match the new sequence.

5.2.12 FRAME ON | OFF

FORMAT:

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

EXAMPLE:

```
FRAME ON
FRAME OFF
```

SUMMARY:

FRAME OFF and **FRAME ON** turn the INTOUCH screen or window frame–the bars across the top and bottom of your screen–OFF or ON.

5.2.13 REFRESH

FORMAT:

```
REFRESH
```

EXAMPLE:

```
REFRESH
```

SUMMARY:

You can use **REFRESH** when your screen gets garbled. REFRESH repaints the screen, the frame (if it is on) and the INTOUCH prompt.

5.2.14 IDE

FORMAT:

```
IDE
```

EXAMPLE:

```
IDE
```

SUMMARY:

You can use **IDE** if you want to find out the current version number for your copy of INTOUCH. Use the version number when contacting Touch Technologies, Inc. with suggestions problems.

5.2.15 COMPILE

FORMAT:

```
COMPILE ['file_spec']
```

EXAMPLE:

```
COMPILE 'display_name'
```

SUMMARY:

Use **COMPILE** to compile your finished INTOUCH programs. COMPILE will convert the program in current memory into an INTOUCH runnable file. Runnable files have the extension **.RUN**. The .RUN file can then be executed from the system prompt by typing INTOUCH followed by the filename. For example:

```
$ INTOUCH display_name
```

The above command starts INTOUCH and runs the compiled version of the program "display_name". When the program has finished running, control will return to the system prompt.

5.2.15.1 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 approximately 500 disk blocks. Image sizes increase in 256 block increments.

INTOUCH images are compiled from *outside* the INTOUCH environment. To compile a program/procedure into an image, use the following command:

```
$ INTOUCH/COMPILE filename
```

Example:

```
$ INTOUCH/COMPILE display_name
```

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

```
Compiling:
    $INTOUCH/COMPILE CUSTOMER

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

5.2.15.2 Running Compiled Images

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

```
$ INTOUCH/IMAGE filename
```

Example:

```
$ INTOUCH/IMAGE display_name
```

In order to set up a DCL symbol to activate an INTOUCH compiled image, use the following command format:

```
$ symbolname :== '$INTOUCH_IMAGE filename/IMAGE'
```

Example:

```
$ MAINTAIN :== '$INTOUCH_IMAGE TTI_RUN:MAINTAIN/IMAGE'
```

5.2.15.3 Developing INTOUCH Programs

The following command entered at the system prompt:

```
$ INTOUCH/DEVELOP filename.INT
```

Example:

```
$ INTOUCH/DEVELOP sample_print.int
```

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

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

```
$ INTOUCH               gets into the INTOUCH environment

OLD 'filename'          brings the file into the workspace

REPLACE 'filename'      after you make your changes and enter
                        EXIT, INTOUCH adds "1" to the version
                        number and replaces the file
```

6 Program Directives and General Statements

6.1 Compile Directives

The following sections describe the directives that are available for use in your programs. These directives are invoked when the compiler OLDs in a program and/or when a program is compiled.

6.1.1 %INCLUDE

FORMAT:

```
%INCLUDE 'file_spec'
```

EXAMPLE:

```
100  %INCLUDE 'tti_run:example'
```

SUMMARY:

You can use **%INCLUDE** to include source code from another file into the current INTOUCH program. The included file cannot contain line numbers. The default extension for the *file_spec* is **.INC**.

6.1.2 %INCLUDE CONDITIONAL

FORMAT:

```
%INCLUDE CONDITIONAL: 'file_spec'
```

SUMMARY:

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

6.1.3 %DEBUG

FORMAT:

```
%DEBUG INTOUCH_statement
```

EXAMPLE:

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

SUMMARY:

This directive gives an error *only* if an image compile (/COMPILE) is being done. Use this directive to make sure that DEBUG code does not show up in production applications.

6.1.4 %MESSAGE

FORMAT:

```
%MESSAGE 'quoted_text'
```

EXAMPLE:

```
10  %MESSAGE 'Including HELP module'
20  %INCLUDE 'tti_run:help'
30  END
```

SUMMARY:

This compiler directive displays a message, *quoted_text*, on the message line. The message is displayed when a program is OLD'ed in or compiled.

6.1.5 %MESSAGE ERROR

FORMAT:

```
%MESSAGE ERROR: 'quoted_text'
```

EXAMPLE:

```
10  %MESSAGE ERROR: 'Using experimental HELP module'
20  %INCLUDE 'tti_run:help'
30  END
```

SUMMARY:

This compiler directive displays an error message, *quoted_text*, on the message line. The message is displayed when a program is OLD'ed in or compiled.

6.2 General Use Statements

6.2.1 PROGRAM

FORMAT:

```
PROGRAM prog_name
```

EXAMPLE:

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

SUMMARY:

Use the **PROGRAM** statement to name your program. *prog_name* is the program name. The program name must meet the specifications for variable names. It must:

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

6.2.2 END

FORMAT:

```
END
```

EXAMPLE:

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

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

SUMMARY:

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

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

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

6.2.3 STOP

FORMAT:

```
STOP
```

EXAMPLE:

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

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

SUMMARY:

The **STOP** statement will terminate program execution. It executes exactly the same as the END statement, but does NOT mark the physical end of your program.

6.2.4 HALT

FORMAT:

```
HALT
```

EXAMPLE:

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

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

INTOUCH
```

SUMMARY:

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

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.

6.2.5 RANDOMIZE

FORMAT:

```
RANDOMIZE
```

EXAMPLE:

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

RNH
 .244013674718

INTOUCH

RNH
 .524856061388
```

SUMMARY:

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.

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

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

6.2.6 DELAY

FORMAT:

```
DELAY [num_expr]
```

EXAMPLE:

```
10  PRINT 'Waiting a bit'
20  DELAY 4.5
30  PRINT 'Done'
40  END

RNH
Waiting a bit
Done
```

SUMMARY:

DELAY causes INTOUCH to pause for the specified number of seconds before continuing program execution. The numeric expression, *num_expr*, does not have to be a whole number, you can use fractions. For example:

```
DELAY 3.5
```

If *num_expr* is omitted, INTOUCH prints this message at the bottom of the screen:

```
Press the RETURN key to continue
```

and waits for the user to respond.

7 Assigning Variables

7.1 Statements Used to Assign Variables

Variables specify storage locations. This chapter describes the INTOUCH statements used to assign variables.

7.1.1 LET

FORMAT:

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

EXAMPLE:

```
10  INPUT 'Total sales': total
20  INPUT 'Number of sales': number
30  LET average=total/number
40  PRINT average
50  END

RNH
Total sales? 2590
Number of sales? 20
 129.5
```

SUMMARY:

Use the **LET** statement to store information into a variable or data structure field. LET stores the value of the expression into the storage location indicated by the variable,*var*. The data types of the variable and the expression must match or an exception will be generated. String variables are dynamic in length.

7.1.2 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              )
```

SUMMARY:

To left-justify string data, use the **LSET** statement. When you use LSET, the string variable is not re-allocated. The operation takes place within the data of the current string. Leading and trailing spaces are carried over.

7.1.3 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)
```

SUMMARY:

To right-justify string data, use the **RSET** statement. When you use RSET, the string variable is not re-allocated. The operation takes place within the data of the current string. Leading and trailing spaces are carried over.

7.1.4 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      )
```

SUMMARY:

To center string data, use the **CSET** statement. When you use CSET, the string variable is not re-allocated. The operation takes place within the data of the current string. Leading and trailing spaces are carried over.

7.1.5 LSET, RSET, CSET FILL

FORMAT:

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

EXAMPLE:

```
10  heading$ = REPEAT$('.', 20)          ! Twenty dots
20  PRINT '('; heading$; ')'
30  CSET FILL '*': heading$ = 'Page 12'
    PRINT '('; heading$; ')'
40  END

RNH
(.....................)
(*******Page 12******)
```

SUMMARY:

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.

7.1.6 DATA and READ

FORMAT:

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

EXAMPLE:

```
10  DIM months$(3)
20  DATA January, February, March
30  FOR i = 1 TO 3
      READ months$(i)
      PRINT months$(i)
    NEXT i
40  END
```

```
RNH
January
February
March
```

SUMMARY:

Use **READ** and **DATA** statements to assign data to variables in cases where the data will not change with successive runs of the program.

DATA specifies a list of data to be assigned. The data must be given as constants. Multiple data items must be separated by commas. READ specifies a list of variables to assign data to. When READ is executed, INTOUCH goes to the first DATA statement and assigns the items to the variables in the READ list. The variable types and data types must match or an exception will result. When a second READ is executed, INTOUCH starts reading from the first unread item in the DATA list. The READ and DATA statements must occur in the same program unit.

7.1.7 RESTORE

FORMAT:

```
RESTORE
```

EXAMPLE:

```
10  DIM months$(3), more_months$(3)
20  DATA January, February, March
30  FOR i = 1 TO 3
      READ months$(i)
      PRINT months$(i)
    NEXT i
40  RESTORE
50  FOR i = 1 TO 3
      READ more_months$(i)
      PRINT more_months$(i)
    NEXT i
60  END
```

```
RNH
January
February
March
January
February
March
```

SUMMARY:

Use **RESTORE** when you want to use the same set of data (from a DATA statement) for a number of READ statements. RESTORE restores the DATA statements in a program unit so that they can be used again by the next READ statement.

7.1.8 DECLARE STRING | INTEGER | REAL | DYNAMIC

FORMAT:

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

EXAMPLE:

```
10  DECLARE STRING name, sex
    DECLARE INTEGER age
    DECLARE REAL amount
    DECLARE DYNAMIC anything
20  INPUT 'Enter your name': name
    INPUT 'Enter your age': age
    INPUT 'Enter your sex': sex
    INPUT 'Enter an amount': amount
30  PRINT
    PRINT name; ' is a'; AGE; 'year old '; sex
    PRINT name; ' entered the amount'; amount
40  anything = 4 + 5
    PRINT 'This dynamic variable contains: '; anything
    anything = 'kitty'
    PRINT 'Now it contains a string value: '; anything
50  END
```

```
RNH
Enter your name? Sammy
Enter your age? 28
Enter your sex? male
Enter an amount? 25.38
```

```
Sammy is a 28 year old male
Sammy entered the amount 25.38
This dynamic variable contains:  9
Now it contains a string value: kitty
```

SUMMARY:

Use the **DECLARE** statement to declare the data type of a variable. That way you do not have to repeat it every time you use the variable. **STRING** indicates the following are string variables. **INTEGER** indicates integer numeric variables and **REAL** indicates real numeric variables. Once you have declared the data type of a variable or function, you do not have to designate them with a trailing $ or %.

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.

7.1.9 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
```

SUMMARY:

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.

7.1.10 DECLARE PREFIX

FORMAT:

```
DECLARE PREFIX str_expr
```

EXAMPLE:

```
irp$total   prefix is IRP
doit$now$   prefix is DOIT
doing_it$   there is no prefix
```

SUMMARY:

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)
```

SUMMARY:

The **DECLARE PREFIX** statement causes all non-prefixed variable references, structure name references and field name references to inherit a prefix (i.e. "DOIT").

Declared prefixes are local to the routine that they are declared in. For this first release, nested declared prefixes are not supported. The following statements turn OFF any declared prefixing:

```
ROUTINE routine_name
END ROUTINE
DECLARE PREFIX none
```

In addition, at run time, prefixing is turned off before and after a string is **EXECUTE**d. (See the EXECUTE statement for information on how EXECUTE works.) This allows you to locally prefix executed code:

```
10  a = 100
20  EXECUTE 'DECLARE PREFIX excode \ a = 45 \ PRINT a'
30  PRINT a
40  EXECUTE 'PRINT excode$a'
50  END

RNH
45
100
45
```

If you want the same prefix to be used throughout an entire procedure, you need to put a DECLARE PREFIX at the beginning of the procedure and after every ROUTINE statement.

When referencing structure field names and using a prefix, you must:

```
STRUC(#'field_name')

       so:

VEND(city)  --> vend(#'city')
```

Without the quotes, CITY would receive a prefix and would be an invalid field name.

Prefixing also applies to structure names. Therefore, if the structure is not opened within the prefixed routine that references it, you must have opened the structure with an explicit prefix.

```
 10  OPEN STRUCTURE ar$cust : NAME 'tti_run::customer', ACCESS INPUT
     GOSUB find_customer
     STOP

100  ROUTINE find_customer
       DECLARE PREFIX find
       c$ = '13727'
       SET STRUCTURE ar$cust, FIELD custnbr : KEY c$
       PRINT ar$cust(#'name')
     END ROUTINE
999  END
```

7.1.11 OPTION REQUIRE DECLARE

FORMAT:

```
OPTION REQUIRE DECLARE
```

EXAMPLE:

```
10  OPTION REQUIRE DECLARE
20  DECLARE STRING name, comment
30  INPUT 'Please enter your name': name
40  LINE INPUT 'Enter a comment in quotes': comment
50  PRINT name; ' says, '; comment
60  END

RNH
Please enter your name? George
Enter a comment in quotes? 'Have a nice day!'
George says, 'Have a nice day!'
```

SUMMARY:

OPTION REQUIRE DECLARE causes INTOUCH to require all variables in a program be declared. If the OPTION REQUIRE DECLARE is used and a variable is left undeclared, INTOUCH will return an error when program execution is attempted.

7.1.12 OPTION BASE

FORMAT:

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

SUMMARY:

OPTION BASE sets the lowest subscript or base for arrays. The base can be either zero or one. If you use OPTION BASE 0, the lowest element of an array has the subscript zero (0). If you use OPTION BASE 1, the lowest element is subscript one (1).

See Section 10.1.3 for an example and detailed information on this statement.

8 Inputting Data from a Terminal

The **INPUT** statement is used to ask questions at the terminal and store the answers for use in programs. This chapter describes the INPUT statement and all of the options that can be used with this statement.

8.1 The INPUT Statement

FORMAT:

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

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

SUMMARY:

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 provided and waits for the user's response. The user's response is then assigned to the variable(s) specified.

For information on INPUT from a text file, see Chapter 13, File Handling.

When the user enters data in response to an INPUT prompt, the input data must be the same data type as the destination 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 (" ").

8.2 Statement Types and INPUT Styles

Types of INPUT Statements

There are three types of INPUT statements. They are:

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

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

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

INPUT styles

There are four styles or ways of inputting data. These are:

1 Simple input from the terminal. For example:

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

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

2 Formatted data entry screens (see Section 8.6.10, SCREEN Option). For example:

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

RNH
Please enter your name? Sam
Soc. sec.:  ___-__-____
```

3 Free format multiline text input (see Section 8.6.1, AREA Option). For example:

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

RNH

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

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

4 Pop-up menus (see Section 8.6.8). For example:

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

RNH

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

Menu option was CHANGE
```

8.3 Response Commands and Keystrokes Available to the User

At an input prompt, there are several commands and keystrokes that you can enter instead of typing in a data response. These options are:

EXIT or CTRL/Z

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

\ or UP arrow

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

HELP

You can get help by typing the word **HELP** or by pressing the Help key. If you enter this option, INTOUCH sets the internal variable **_HELP** to TRUE (1).

Note

All input statements set the internal variables _BACK, _EXIT and _HELP to TRUE or FALSE regardless of the actual response. This allows you to examine the variables within your program and take appropriate action.

8.3.1 Editing Input

If you want to edit your input response, you can use the following keys:

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

8.4 Inputting Strings

If the input is a string variable, you can enter an unquoted or a quoted string. If you enter an unquoted string, INTOUCH ignores any leading spaces, trailing spaces and tabs. An unquoted string cannot contain commas. If you enter a quoted string, INTOUCH removes the quotes, and includes any leading and/or trailing spaces and/or tabs.

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

RNH
Enter your name? Sam
Enter your name in quotes? 'Sam'
Enter your name in quotes with spaces? '  Sam  '
Enter last name, comma, space, first name in quotes? 'Jones, Sam'

Sam
Sam
  Sam
Jones, Sam
```

8.5 Inputting Multiple Variables

8.5.1 Simple Input Statement

A single input statement can be used to input several variables. The input variables must be separated by commas.

```
10  INPUT 'Enter name, age, sex separated by commas': name$, age%, sex$
20  PRINT name$, age%, sex$
30  END

RNH
Enter name, age, sex separated by commas? Tommy Tune, 31, Male
Tommy Tune           31                Male
```

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

You can enter the data for a variable list in one of two ways:

1. You can enter the data on one line by separating the data with commas as shown in the example above.
2. You can enter each piece of data on a separate line by typing a comma as the last character on each continuing line. The comma tells INTOUCH that there is more data to come.

```
RNH
Enter name, age, sex separated by commas? Tommy Tune,
? 31,
? Male
Tommy Tune           31                Male
```

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

8.5.2 LINE INPUT Statement

The **LINE INPUT** statement gets a full line of data from the terminal. This statement is used in cases where an entire line from the terminal must be returned to the program. LINE INPUT accepts everything the user types until a line terminator is pressed, as the input line. LINE INPUT accepts commas, semicolons, etc. All of the options available with the INPUT statement are available with LINE INPUT.

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

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

RNH
? This is the first comment
? This is the second comment
This is the first comment
This is the second comment
```

INTOUCH asks for input until all of the variables have received a value. Unlike the simple 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 taken as part of the text.

8.6 INPUT Options

The INPUT statement has a number of options. These options are:

AREA	does free format multiline text input from an area on the screen
AT row, col	positions the cursor on the desired row and column
ATTRIBUTES	asks for input in one or more of these forms: BOLD, BLINK, REVERSE, UNDERLINE
DEFAULT	lets you provide defaults for INPUT statements
ELAPSED	keeps track of the time it takes a user to respond to an INPUT prompt
ERASE	clears the input line prior to accepting input and after the input has been completed
LENGTH nn	limits the number of characters that can be entered
MENU	displays and receives input from "pop-up" menus
PROMPT	displays the prompt text
SCREEN	creates formatted data entry screens
TIMEOUT	limits the time the user has to respond to the INPUT prompt
VALID	validates user responses

8.6.1 AREA Option

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

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

The **AREA** option does LINE INPUT from a specified area on the screen. During 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 keys:

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/BT	moves cursor to the top of the area
GOLD/F	complete (finish) and save the input
F10	exit (abort) the input
CTRL/Z	exit (abort) the input
\	EXIT

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

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

RNH

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

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

8.6.2 AT Option

```
AT row, column
```

The **AT** option positions the cursor on the specified row and column. *row* and *column* can be any integer numeric constants.

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

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

8.6.3 ATTRIBUTES option

```
[LINE] INPUT ATTRIBUTES attr_list: str_var
LINE INPUT AREA row_1, col_1, row_2, col_2, ATTRIBUTES attr_list: str_var
INPUT SCREEN str_expr1, ATTRIBUTES attr_list: str_var
```

The **ATTRIBUTES** option does input with attributes. The available attributes are: BOLD, BLINK, REVERSE, UNDERLINE. You can use multiple attributes in one INPUT statement by separating them with commas.

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

RNH
Enter your name? Sandy
Hello Sandy
```

8.6.4 DEFAULT Option

```
DEFAULT str_expr
```

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

```
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
```

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

Upon completion of an INPUT MENU) statement, the function **_STRING** contains the menu path taken by the user when selecting the menu item. E.g. "#2;#3" means the 3rd item of the 2nd submenu.

8.6.5 ELAPSED Option

```
ELAPSED num_var
```

The **ELAPSED** option keeps track of the time it takes the user to respond to an INPUT prompt. *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). 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? John
John, your elapsed time is 1.39
```

8.6.6 ERASE option

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

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

RNH
Please enter a name? James

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

8.6.7 LENGTH Option

```
LENGTH num_expr
```

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

```
10  INPUT 'Enter your name', LENGTH 15: name$
    PRINT name$
20  END

RNH
Enter your name? Junior_________
Junior
```

8.6.8 MENU Option

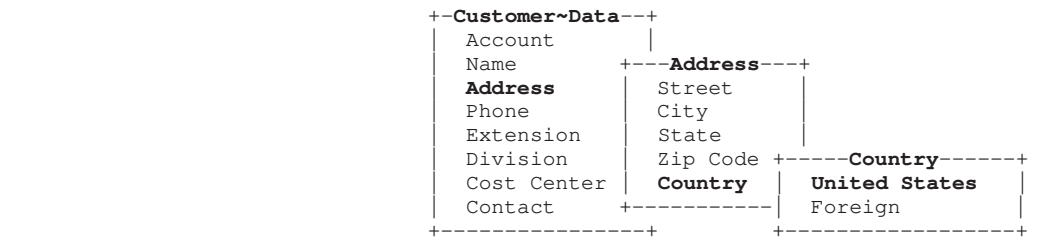
```
[LINE] INPUT MENU str_expr: str_var
```

The **MENU** option is used to create pop-up menus. The *str_expr* is a string expression that describes the menu. It consists of multiple, comma-separated elements. *str_var* contains the name of the selected item. If the **%multi** menu directive is used and the Select key is pressed, *str_var* will contain a list of selected items, with each item separated by a linefeed character.

Menu elements can be either items or directives. An item consists of either a description or a description and an = sign followed by either a data value or a submenu indicator. If no = sign is given, the data value is the same as the description. If the description contains spaces or special characters, it *must* be enclosed within quotes.

A submenu consists of a "{" followed by menu elements, followed by "}". The following example shows a menu with submenus and a submenu which has a submenu.

```
10  clear
20  customer_data$ = '%title "Customer Data", "Account",' &
        + '"Name" = {"First", "Middle", "Last"},' &
        + '"Address" = {"Street", "City", "State", "Zip Code",' &
        + '"Country" = {"United States", "Foreign"}},' &
        + '"Phone", "Extension", "Division", "Cost Center", "Contact"'
30  line input menu customer_data$: selection$
40  print at 21,1 : 'The menu selection was:  '; selection$
50  end

RNH
```




```
The menu selection was:  United States
```

The *INTOUCH - A Guide to the Language* manual contains detailed information on menus. The *INTOUCH Application Development Guide* contains many menu examples that you can try out.

8.6.8.1 Menu Directives

There are various directives that you can use with your menus. These are:

%AT row, column

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

%AUTODISPLAY [ON | OFF]

Controls display of submenus. Unless specifically turned OFF, INTOUCH automatically displays submenus.

%AUTOVBAR [ON|OFF]

Turns vertical bars ON and OFF in menus.

%BAR ["text_str"]

Separates DESCRIPTIVE-items. 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.

%COLUMNS number

Sets the number of columns for the menu.

%HEADING ["text_str"]

Displays a blank line and a "text_str" between menu items. If no "text_str" is given, the line derfaults to a dashed line.

%ITEMS number

Creates a multi-column menu.

%LOCKSTEP [ON | OFF]

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.

%MAXWIDTH number

Controls the maximum width of the entire menu.

%MENUBAR

Creates menu bars (menus with choices listed horizontally) and pull-down submenus.

%MULTI

Allows multiple items to be selected.

%REPLACE

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

%SIZE number

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

There is no limit to the number of the items that can be used. Vertical scrolling can be done using the UP, DOWN, NEXT and PREV keys.

%SPLIT

Instructs INTOUCH to start a new column at that specified point in the menu.

%TITLE "....."

Assigns a title to the menu.

%VBAR

Inserts a vertical bar between menu items in a bar menu.

%WIDTH number

Controls the minimum width of the current column.

User Interface

The following user actions are acceptable with [LINE] INPUT MENU:

- UP and DOWN arrow keys move the menu cursor from one item to another. LEFT and RIGHT arrow keys are used in a bar menu.
- \ (backslash) or LEFT arrow key returns to the calling menu.
- CTRL/Z exits the menu.
- SPACE bar, Return, RIGHT arrow are used to select an item if the item is on a submenu.
- SPACE bar or Select keys are used with the **%multi** option to select multiple menu items.
- Remove key can be used with the **%multi** option to remove an item, or press the SPACE bar a second time.
- The DEL key moves the menu cursor back to the first item of the menu.

8.6.9 PROMPT Option

By default, INTOUCH prints a question mark and a space, then waits for the user's response. You can display some prompt text before the question mark 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. When INTOUCH executes the INPUT statement, it prints the prompt text, followed by a question mark and a space. For example:

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

RNH
Please your name? Sally
Hello Sally
```

The **PROMPT** option displays the prompt text *without* the question mark and space. The format is:

```
PROMPT str_expr
```

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

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

RNH
Please your name:  Sally
Hello Sally
```

8.6.10 SCREEN Option

```
SCREEN str_expr: input_list
```

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.

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

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

format options
format characters

Format options come first. They are followed by a colon and then by format characters. If no format options are used, place a colon after the first angle bracket.

```
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? Skipper
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 input is assigned to the variable SSN.

8.6.10.1 Screen Format Characters

#

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

@

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

.

A decimal point is used to align the fractional portion of a floating point number. It also displays as part of the data.

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.

^

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.

~

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

8.6.10.2 Screen Format Options

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

UCASE	Upper-cases all letters that are typed.
LCASE	Lower-cases all letters that are typed.
NOECHO	Typed characters are not echoed (printed) on the screen.
DIGITS	Allows only digits to be entered in the field. The format character # also allows the minus sign and the decimal point.
AJ	Causes INTOUCH to jump automatically to the next field when the current field has been filled in. The field must be completely filled in before INTOUCH will jump to the next field.
REQ	Specifies that input is required and you must enter something.
AT row, column	Displays the field on the screen at the row and column position specified. *row* specifies the row to print at. *column* specifies the column to print at. You can supply defaults for each field in the screen. The format of the defaults is a list of values separated with line feeds.
VALID	Allows full validation of each field on the screen. (See the VALID function in Appendix A for a list of validation rules.) The format of the VALID option is: VALID 'rule_str' *rule_str* is the list of validation rules. Multiple validation rules are separated by semi-colons.
ELAPSED time	Keeps track of the time it takes the user to respond to an INPUT prompt. INTOUCH assigns the elapsed time to the variable specified. The ELAPSED option starts marking time when INTOUCH finishes displaying the prompt. Time stops when all the variables listed in the INPUT statement have values. The format is: ELAPSED num_var. *num_var* is the numeric variable the elapsed time is assigned to. INTOUCH returns the elapsed time in seconds. If a real numeric variable is specified, INTOUCH returns the time to a hundredth of a second (1/100). If an integer variable is specified, INTOUCH rounds the time to the nearest second.
BOLD, BLINK, UNDERLINE, REVERSE	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.

You can refer to *INTOUCH A Guide to the Language* for a list of format option examples.

8.6.11 TIMEOUT Option

```
TIMEOUT num_expr
```

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

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

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

RNH
Name? Timeout on input at 10

INTOUCH
```

8.6.12 VALID Option

```
VALID str_expr
```

where *str_expr* is the list of validation rules.

The **VALID** option validates user responses according to validation rules. You can refer to the VALID function in Appendix A, Built-in Functions for information on all the validation rules.

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

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

8.7 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
```

SUMMARY:

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

KEY INPUT does not echo the key or generate a line feed. KEY INPUT returns one character at a time from the file open on the given channel.

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

9 Printing Data to a Terminal

This chapter describes the various ways that data can be presented on a terminal screen.

9.1 PRINT

FORMAT:

```
PRINT [[AT row, col] [,ERASE] [,WIDE] [,BLINK] [,UNDERLINE] [,REVERSE]
      [,BOLD] [,USING "print_mask"]:] expr [{, | ;} expr...] [, | ;]
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    PRINT BOLD, UNDERLINE: 'Hello, '; name$
    PRINT name$, 'Have a good day'
20  END

RNH
Please enter your name? James
Hello, James
James               Have a good day
```

SUMMARY:

The **PRINT** statement is used to print text on the screen. The PRINT statemnt is also used to print to files. For information on printing to files, see Section 13.3, PRINT #chnl_num.

When printing *comma* separated items on the screen, INTOUCH will print the items in separate print zones. If a *semicolon* is used to separate the items, the items will immediately follow one another (i.e. there will be no spaces between items). If the message line is altered and a message requiring a delay is present, the print will be delayed.

The **?** can be used as a synonym for PRINT at the command prompt. For example:

```
INTOUCH

? ROUND((10000. * .072 * 30.) / 365., 2)
 59.18

INTOUCH
```

9.1.1 PRINT Options

The PRINT statement has a number of options. These options are explained in the following subsections.

9.1.1.1 TAB Function

```
TAB(column)
```

TAB positions the cursor at the column specified. *column* is the column position to print at. TAB always moves forward (to the right). 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 'Please enter your name': name$
    PRINT 'Hello, '; name$; TAB(30); 'Have a good day'
20  END
```

```
RNH
Please enter your name? Sam
Hello, Sam                  Have a good day
```

9.1.1.2 AT row, column

```
AT row, column
```

The **AT** option prints the first variable at the coordinates specified.

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

```
    RNH
    Please enter your name? Tom

                    Hello, Tom
```

9.1.1.3 BLINK | UNDERLINE | BOLD | REVERSE

These options are used to highlight expressions when they are printed on the screen. These options can be combined.

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

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

9.1.1.4 WIDE Option

The **WIDE** option prints *double-width* characters on the terminal screen.

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

9.1.1.5 ERASE Option

The **ERASE** option erases from the end of the printed text to the end of the line.

```
10  CLEAR
20  PRINT AT 10, 1: 'Mary had a little lamb'
    DELAY 2
30  PRINT AT 10, 1: 'Jeff'
    DELAY 2
40  PRINT ERASE, AT 10, 1: 'Caroline'
    DELAY 2
50  END
```

```
RNH
Mary had a little lamb
.
Jeff had a little lamb
.
Caroline
```

9.1.1.6 PRINT USING Option

The **USING** option allows you to format numbers and text by using masks consisting of format characters or by using directives.

Using Format Characters

The USING option lets you print text to the screen using an edit mask. This allows you to format output uniformly–even when you do not know what that output is going to be. Here is an example showing some ways you can use the format characters.

```
10  PRINT USING '####': 25
    PRINT USING '@@@@': 25
    PRINT USING '%%%%% %%%%%': 23, 4567
    PRINT USING '***,***,***.**': 1222333.44
20  z$ = '$###,###,###.##'
    PRINT USING z$: 1222333.44
30  END
```

```
RNH
  25
0025
00023 04567
**1,222,333.44
  $1,222,333.44
```

A number of format characters can be used to control how expressions are printed. The available format characters are:

#	Used to indicate a character position. Can also be used to specify numeric characters.
@	Indicates one character position with no translation.
<	The less than sign left-justifies text within a field.
>	The greater than sign is used to right-justify text within a field.
.	You can include a decimal point in the number by typing a period or decimal point in the format.
,	You can include commas in your numbers by typing a comma in the format.
%	Zero fill. Pad on the left with zeros.
*	Pad on the left with asterisks.
+	Causes INTOUCH to print a leading plus or minus sign.
-	Causes INTOUCH to print a leading or trailing minus sign for negative numbers, and a leading space for positive numbers.
$	Causes INTOUCH to print a floating dollar sign.
+$	Causes INTOUCH to print a plus sign before positive numbers, a minus sign before negative numbers, and a floating dollar sign.
-$	Causes INTOUCH to print a floating dollar sign. Will cause INTOUCH to print a minus sign before negative numbers and a space before positive numbers.
$+	Causes INTOUCH to print a floating dollar sign. Will cause INTOUCH to print a minus sign before negative numbers, and a plus sign before positive numbers.
$-	Causes INTOUCH to print a floating dollar sign. Will cause INTOUCH to print a minus sign before negative numbers, and a space before positive numbers.

Using Directives

There are several **DIRECTIVES** that you can use with the PRINT USING option. The directives tell INTOUCH what to do with the text. The format is:

```
PRINT USING '{directive [argument]}?' : str_expr
```

{UCASE}? and {LCASE}?

The **UCASE** directive converts the *str_expr* to upper-case characters.

The **LCASE** directive converts the *str_expr* to lower-case characters.

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

RNH
MARCH
april
```

{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
```

{DATE [argument]}?

Given a *str_expr* that contains a date in the format YYMMDD or CCYYMMDD, the **DATE** directive converts the *str_expr* to a default or specified, optionally-masked, date format.

The date arguments are: YMD, CYMD, MDY, MDCY, DMY, DMCY, DMONY, DMONCY, MONTHDY, MONTHDCY. If no argument is provided, the default is MDCY. (See FORMAT$(expr, str_expr) for examples of date argument usage.)

You can format the resulting date by replacing the **?** with a print mask.

```
10  PRINT USING '{DATE}?': '950401'
    PRINT USING '{DATE DMY}?': '950401'
    PRINT USING '{DATE DMCY}?': '950401'
    PRINT
    PRINT USING '{DATE MDY}?': '19950215'
    PRINT USING '{DATE MDY}##/##/##': '19950215'
    PRINT USING '{DATE MDCY}##/##/####': '19950215'
20  END

RNH
04011995
010495
01041995

021595
02/15/95
02/15/1995
```

{ROTATE n}?

The **ROTATE** directive rotates the last *n* characters in a *str_expr* to the first position in the *str_expr*. Optionally, the resulting *str_expr* can be masked by replacing the **?** with a print mask.

```
10  PRINT USING '{ROTATE 3}?': '5552527800'
    PRINT USING '{ROTATE 3}###~ ###~-####': '5552527800'
    PRINT
    PRINT USING '{ROTATE 5}?': 'TuneTommy'
    PRINT USING '{ROTATE 5}#####~ ####': 'TuneTommy'
20  END

RNH
8005552527
800 555-2527

TommyTune
Tommy Tune
```

{TIME}?

Given a *str_expr* containing a 4 digit time in HHMM or HH:MM format or a 6 digit time in HHMMSS or HH:MM:SS format, the **TIME** directive converts the *str_expr* to HH:MM AM/PM or HH:MM:SS AM/PM.

```
10  PRINT USING '{TIME}?': '1022'
    PRINT USING '{TIME}?': '19:45'
    PRINT
    PRINT USING '{TIME}?': '102255'
    PRINT USING '{TIME}?': '19:45:36'
20  END

RNH
10:22 AM
07:45 PM

10:22:55 AM
07:45:36 PM
```

{ZIPCODE}?

Given a *str_expr* containing a 5, 6 or 9 digit zipcode, the **ZIPCODE** directive converts the *str_expr* to an appropriate zipcode format.

```
10  PRINT '5 character zipcode : ';
    PRINT USING '{ZIPCODE}?': '92126'
    PRINT '6 character zipcode : ';
    PRINT USING '{ZIPCODE}?': 'K8A3P9'
    PRINT '9 character zipcode : ';
    PRINT USING '{ZIPCODE}?': '931327845'
20  END

RNH
5 character zipcode : 92126
6 character zipcode : K8A 3P9
9 character zipcode : 93132-7845
```

9.2 Using the INTOUCH Screen

INTOUCH has features that allow you to control the appearance of the screen. The following sections describe some of these features.

9.3 MESSAGE [ERROR: | DELAY:]

FORMAT:

```
MESSAGE [ERROR: | DELAY:] str_expr [{, | ;} expr]
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:
20  MESSAGE 'Please enter your name'
    INPUT 'Name': name$
30  WHEN EXCEPTION IN
      INPUT 'How old are you': age
    USE
      MESSAGE ERROR: 'Not a valid age'
      RETRY
    END WHEN
40  PRINT
    PRINT name$; ' is'; age
50  END

RNH
```

```
Name?
                              Please enter your name

Name? Steven
How old are you? w
                              Not a valid age

How old are you? 39

Steven is 39
```

SUMMARY:

INTOUCH uses the message line (defaults to line 23) to display messages in bold print. You can use the **MESSAGE** statement to display your own messages and errors on this line. When the INTOUCH frame is ON, your messages will appear within the lower frame. If the frame is OFF, the messages will appear at the bottom of the screen.

Multiple message items must be separated with a comma or a semicolon. The separator determines where the *next* expression will be printed.

The **ERROR** option causes INTOUCH to ring the terminal bell when the message is printed and disregard any characters that were typed ahead.

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 longer messages.

9.4 CLEAR

FORMAT:

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

EXAMPLE:

```
10  CLEAR
20  INPUT 'Enter your name': name$
    PRINT 'Hello, '; name$
30  END

RNH

Enter your name? John
Hello, John
```

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.

SUMMARY:

CLEAR, by itself, clears all text from the screen. CLEAR does not remove the INTOUCH frame if it is on. However, it does remove any message text that is displayed within the frame. If the frame is on, INTOUCH positions the prompt within the frame.

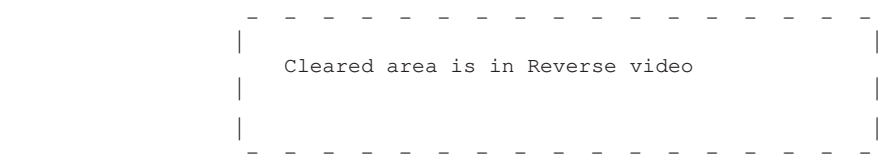
9.4.1 AREA Option

FORMAT:

```
CLEAR AREA attr_list: row_1, col_1, row_2, col_2
```

EXAMPLE:

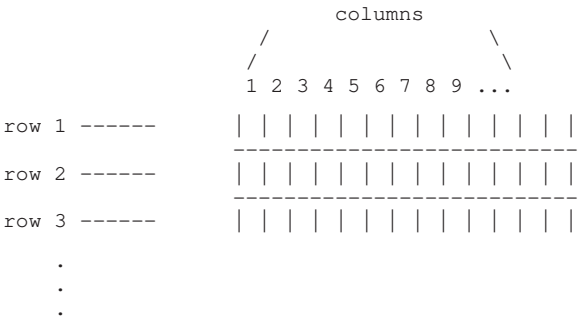
```
10  CLEAR
    CLEAR AREA REVERSE: 5, 10, 11, 60
    PRINT AT 7, 20: 'Cleared area is in Reverse video'
20  END

RNH
```



SUMMARY:

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 typically 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. 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 example, 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.

The CLEAR AREA statement allows the following attributes to be used when clearing an area: **BOLD**, **BLINK**, **REVERSE**, **UNDERLINE**. Multiple attributes used in one statement must be separated by commas.

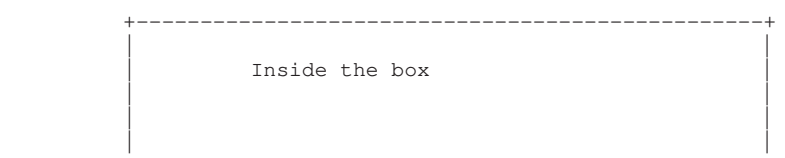
9.4.2 BOX Option

FORMAT:

```
CLEAR AREA BOX, attr_list: row_1, col_1, row_2, col_2
```

EXAMPLE:

```
10  CLEAR
    CLEAR AREA BOX, BOLD: 5, 10, 11, 60
20  PRINT AT 7, 20: 'Inside the box'
    PRINT AT 12, 1: 'Outside the box'
30  END

RNH
```




```
Outside the box
```

SUMMARY:

The **BOX** option creates an empty box with a frame.

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

9.5 OPTION COMMUNICATE ON | OFF

FORMAT:

```
OPTION COMMUNICATE ON | OFF
```

EXAMPLE:

```
10  CLEAR
    OPTION COMMUNICATE OFF
    PRINT AT 1,1:
20  INPUT 'Enter your name': name$
    PRINT name$
    OPTION COMMUNICATE ON
30  INPUT 'Enter your name': name$
    PRINT name$
40  END

RNH

Enter your name? Tester
Tester
Enter your name?

   .New mail on node FAST from FAST::MOLLY        (12:01:08)
```

SUMMARY:

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

10 DEFINING ARRAYS

10.1 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. About arrays:

- The highest bound allowed is 2147483648.
- The lowest bound allowed is -2147483647.
- Each array can have up to 32 dimensions.
- If an array element outside of the specified bounds is referenced, an exception is generated.

10.1.1 DIM

FORMAT:

```
DIM [data_type] array_name ([int_expr TO] int_expr [,...])
```

EXAMPLE:

```
10  DIM name$(4)
20  FOR i = 1 TO 4
      INPUT name$(i)
    NEXT i
30  PRINT
    FOR i = 1 TO 4
      PRINT name$(i)
    NEXT i
40  END

RNH
? John
? Joan
? Jack
? Jenny

John
Joan
Jack
Jenny
```

SUMMARY:

The **DIM** statement is used to dimension arrays. Arrays are used to store and manipulate tables of variable information. DIM describes the number of dimensions in the array and the high bound of each dimension. DIM can also be used to give a low bound. The **OPTION BASE** statement changes the default low bound. By default, the low bound is 1.

data_type is the type of array being dimensioned. The type can be INTEGER, REAL or STRING.

10.1.2 REDIM

FORMAT:

```
REDIM array_name ([int_const TO] int_const [, ...])

                or

REDIM array_name [( [num_expr TO] num_expr,
    [num_expr TO] (num_expr...)] ...
```

EXAMPLE:

```
10  DIM name$(2)
    INPUT 'How many names': num
    REDIM name$(num)
20  FOR i = 1 TO num
      INPUT name$(i)
    NEXT i
30  PRINT
    FOR i = 1 TO num
      PRINT name$(i)
    NEXT i
40  END

RNH
How many names? 3
? Tony
? Charles
? Suzy

Tony
Charles
Suzy
```

SUMMARY:

REDIM redimensions arrays. REDIM can only be used on arrays that have already been dimensioned with the DIM statement. The REDIM statement has the same rules, options and limits as the DIM statement.

You can dynamically expand arrays as needed. If you REDIM a single dimension array or the first dimension of a multi-dimensioned array to a larger size, the old values are kept. If you REDIM any array to a smaller size or REDIM two or more dimensions in a multi dimensioned array to a larger size, the old values are lost.

If your application depends on REDIM initializing all array values, change your code as follows:

```
Old Code:       REDIM X(100)

New Code:       REDIM X(1)
                REDIM X(100)
```

The REDIM X(1) forces all array values to be initialized by the second REDIM statement.

10.1.3 OPTION BASE

FORMAT:

```
OPTION BASE  0 | 1
```

EXAMPLE:

```
10  OPTION BASE 0
    DIM name$(5)
20  FOR i = 0 TO 4
      INPUT name$(i)
      PRINT 'Hello, '; name$(i)
    NEXT i
30  END
```

```
RNH
? Susan
Hello, Susan
? Sandy
Hello, Sandy
? Sam
Hello, Sam
? Sunny
Hello, Sunny
? Sharon
Hello, Sharon
```

SUMMARY:

You can use **OPTION BASE** to set the default *low bound* for arrays to suit your needs. You have the option of starting the array with element O or element 1.

When no low bound is specified for a dimension, the default is 1. The OPTION BASE statement lets you specify a default low bound of 0 or 1. When any following DIM or REDIM statements are executed, INTOUCH defaults the low bound to 0 or 1 as specified.

11 Set and Ask Statements

ASK and SET find and change characteristics within an INTOUCH program. SET sets various characteristics, and ASK returns the value of various characteristics. SET and ASK have many options. These options are described in this chapter.

Note

For information on the SET #chnl and ASK #chnl statements, please refer to Chapter 13.

For information on SET STRUCTURE and ASK STRUCTURE statements, please refer to Section 3.3 and Section 3.5.

11.1 General SET, ASK Statements

The following SET and ASK statments change and return various types of characteristics.

11.1.1 SET AUTOEXIT

FORMAT:

```
SET AUTOEXIT num_expr
```

EXAMPLE:

```
10  SET AUTOEXIT 1
20  DO
      INPUT 'Who': a$
      IF  _EXIT  OR  _BACK  THEN  EXIT DO
30  LOOP
    PRINT 'Finished'
40  END

RNH
Who? Greg
Who? Sammy
Who?              (User fails to respond within 1 minute.)

INTOUCH
```

SUMMARY:

The **SET AUTOEXIT** statement is used to slowly back a user out of a program if the terminal is left idle for a specified period of time. *num_expr* is expressed in minutes.

11.1.2 SET BACK ON | OFF

FORMAT:

```
SET BACK OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _BACK
30  SET BACK ON
40  PRINT _BACK
50  END

RNH
Name? TESTER_____________________________
0
1
```

SUMMARY:

SET BACK OFF sets the internal variable **_BACK** to FALSE (0). **SET BACK ON** sets the internal variable _BACK to TRUE (1).

11.1.3 SET ERROR ON | OFF

FORMAT:

```
SET ERROR ON | OFF
```

EXAMPLE:

```
10  DO
      INPUT 'Enter the age': age
      IF  age < 1  THEN
        PRINT 'Too young:'; age
        SET ERROR ON
      ELSE
        SET ERROR OFF
      END IF
    LOOP WHILE _ERROR
30  END

RNH
Enter the age? .5
Too young: .5
Enter the age? 38
```

SUMMARY:

The **SET ERROR** statement sets or clears the **_ERROR** internal variable.

The following methods can be used to **SET** the _ERROR variable:

- SET ERROR ON statement
- MESSAGE ERROR: statement
- Execution of the END WHEN statement

The following methods can be used to **CLEAR** the _ERROR variable:

- SET ERROR OFF statement
- GOSUB or DISPATCH statement
- WHEN EXCEPTION IN statement

11.1.4 ASK ERRORS

FORMAT:

```
ASK ERRORS num_var
```

EXAMPLE:

```
10  DO
      INPUT 'Enter the age': age
      IF  age < 1  THEN
        MESSAGE ERROR: age; ' Too Young'
        REPEAT DO
      ELSE
        EXIT DO
      END IF
    LOOP
20  ASK ERRORS num_errors
    PRINT 'Errors:'; num_errors
30  END

RNH
Enter the age? 0                  0 Too Young
Enter the age? .5                 .5 Too Young
Enter the age? 21
Errors: 2
```

SUMMARY:

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

11.1.5 SET EXIT ON | OFF

FORMAT:

```
SET EXIT OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _EXIT
30  SET EXIT ON
40  PRINT _EXIT
50  END

RNH
Name? JANE_____________________________
 0
 1
```

SUMMARY:

SET EXIT OFF sets the internal variable **_EXIT** to FALSE (0). **SET EXIT ON** sets the internal variable _EXIT to TRUE (1).

11.1.6 SET HELP ON | OFF

FORMAT:

```
SET HELP OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _HELP
30  SET HELP ON
40  PRINT _HELP
50  END

RNH
Name? Tony_____________________________
 0
 1
```

SUMMARY:

SET HELP OFF sets the internal variable **_HELP** to FALSE (0). **SET HELP ON** sets the internal variable _HELP to TRUE (1).

11.1.7 ASK KEYSTROKES

FORMAT:

```
ASK KEYSTROKES num_var
```

EXAMPLE:

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

```
RNH
Please enter your name? Maryanne
Hello Maryanne
Keystrokes: 8
```

SUMMARY:

The **ASK KEYSTROKES** statement asks for the number of user-entered keystrokes.

11.1.8 ASK | SET MARGIN

FORMAT:

```
ASK MARGIN num_var
SET MARGIN num_expr
```

EXAMPLE:

```
10  PRINT REPEAT$('.' ,200)
    PRINT
    ASK MARGIN old_marg
20  INPUT 'What do you want the margin set to': new_marg
30  SET MARGIN new_marg
40  PRINT REPEAT$('.' ,200)
    SET MARGIN old_marg
50  END
```

```
RNH
................................................
................................................
................................................
................................................

What do you want the margin set to? 20
....................
....................
....................
....................
```

SUMMARY:

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

SET MARGIN sets the right margin on the device specified to the number indicated. *num_expr* specifies the column to set the margin to. The margin must be greater than zonewidth.

11.1.9 ASK | SET MESSAGELINE

FORMAT:

```
ASK MESSAGELINE num_var
SET MESSAGELINE num_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,1:;
20  ASK MESSAGELINE orig_mess_line
    MESSAGE 'Current message line is '; orig_mess_line
    DELAY 4
30  new_line = 12
    SET MESSAGELINE new_line
    MESSAGE 'New message line is '; new_line
    DELAY 4
40  SET MESSAGELINE orig_mess_line
    MESSAGE 'Message line has been reset to its original position'
50  END

RNH
                  Current message line is 23

                    New message line is 12

      Message line has been reset to its original position
```

SUMMARY:

ASK MESSAGELINE returns the line number on which the messages are displayed. This numeric value is stored in *num_var*.

SET MESSAGELINE sets the line on which the next message is displayed.

If **SET MESSAGELINE 0** is used, no messages will be displayed.

11.1.10 ASK PAGESIZE

FORMAT:

```
ASK PAGESIZE num_var
```

EXAMPLE:

```
10  ASK PAGESIZE no_lines
    PRINT 'There are'; no_lines; 'lines or rows on this screen'
20  END

RNH
There are 24 lines or rows on this screen
```

SUMMARY:

ASK PAGESIZE returns the number of rows or lines of terminal screen output.

11.1.11 SET PORT ON | OFF

FORMAT:

```
SET PORT ON | OFF
```

EXAMPLE:

```
10  PRINT 'About to print to my printer port'
20  SET PORT ON
    PRINT 'This is on the printer port'
    SET PORT OFF
30  PRINT 'Back to the screen again'
40  END

RNH
About to print to my printer port
Back to the screen again
```

SUMMARY:

If there is an attached printer port on a terminal, this statement can be used to turn the printer port on and off.

SET PORT ON turns ON the attached port. **SET PORT OFF** turns OFF the attached port.

Additional Information on Setting the Port Type

```
TTI_PRINTER_TYPE
```

This is a DCL symbol that you can set to indicate what type of printer is attached to the terminal's printer port.

To set up the printer type, enter the following at the system prompt:

```
$ TTI_PRINTER_TYPE == printer_type
```

Where *printer_type* is one of the following:

```
DEC
HP
OTHER
```

If the symbol is not defined, the type DEC is assumed. Example:

```
$ TTI_PRINTER_TYPE == HP
```

11.1.12 ASK RESPONSES

FORMAT:

```
ASK RESPONSES num_var
```

EXAMPLE:

```
10  INPUT 'Please enter your name': name$
    INPUT 'What day is this': what_day$
    PRINT 'Hello '; name$
    PRINT 'Have a good '; what_day$
20  ASK RESPONSES answers
    PRINT
    PRINT 'Responses:'; answers
30  END

RNH
Please enter your name? Ginger
What day is this? Wednesday
Hello Ginger
Have a good Wednesday

Responses: 2
```

SUMMARY:

The **ASK RESPONSES** statement asks for the number of completed input responses.

11.1.13 SET SCROLL

FORMAT:

```
SET SCROLL int_expr1, int_expr2
```

EXAMPLE:

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

RNH

This
text
will
scroll

This text will not scroll.
```

SUMMARY:

The **SET SCROLL** statement sets up a scrolling region from line *int_expr1* to line *int_expr2*.

11.1.14 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
```

SUMMARY:

This statement allows you to set or reset the pseudo-random number sequence.

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*.

11.1.15 ASK | SET ZONEWIDTH

FORMAT:

```
ASK ZONEWIDTH num_var
SET ZONEWIDTH num_expr
```

EXAMPLE:

```
10  PRINT 1, 2, 3
    ASK ZONEWIDTH curr_width
    PRINT 'The current print zone width is'; curr_width
20  SET ZONEWIDTH 10
    PRINT 1, 2, 3
    ASK ZONEWIDTH new_width
    PRINT 'The new print zone width is'; new_width
30  SET ZONEWIDTH curr_width
40 END

RNH
 1                  2                  3
The current print zone width is 20
 1         2         3
The new print zone width is 10
```

SUMMARY:

ASK ZONEWIDTH finds the print zone width of the device specified and assigns the value to the numeric variable.

SET ZONEWIDTH sets the print zone width of the device specified to the number designated. *num_expr* indicates the width to set the device's print zones.

11.2 ASK | SET SYSTEM

The ASK | SET statements ask about and set various system operation features. ASK SYSTEM and SET SYSTEM have many options. The options are described in the following sections.

11.2.1 ASK | SET SYSTEM: COMMENT

FORMAT:

```
ASK SYSTEM: COMMENT str_var
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
```

SUMMARY:

SET SYSTEM: COMMENT loads the INTOUCH operating system comment area with the specified string.

ASK SYSTEM: COMMENT asks the operating system for the comment for the process. This comment was stored using a SET SYSTEM: COMMENT statement.

SET SYSTEM: COMMENT str_expr 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.

11.2.2 ASK | SET SYSTEM: DIRECTORY

FORMAT:

```
ASK SYSTEM: DIRECTORY str_var
SET SYSTEM: DIRECTORY str_var
```

EXAMPLE:

```
10  ASK SYSTEM: DIRECTORY z0$
    PRINT 'Current directory     '; z0$
    SET SYSTEM: DIRECTORY 'SYS$LOGIN'
    ASK SYSTEM: DIRECTORY z1$
    PRINT 'Directory set to      '; z1$
    DELAY 2
    SET SYSTEM: DIRECTORY z0$
    PRINT 'Directory set back to '; z0$
20  END

RNH
Current directory      USER:[TESTER.INTOUCH]
Directory set to       USER:[TESTER]
Directory set back to  USER:[TESTER.INTOUCH]
```

SUMMARY:

ASK SYSTEM: DIRECTORY asks the operating system for the current device and directory and stores the information in *str_var*.

SET SYSTEM: DIRECTORY sets the default device and directory.

11.2.3 ASK | SET SYSTEM, LOGICAL: VALUE

FORMAT:

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

EXAMPLE:

```
10  SET SYSTEM, LOGICAL 'SYS$SCRATCH': VALUE 'USER:[TESTER]'
    ASK SYSTEM, LOGICAL 'SYS$SCRATCH': VALUE z$
    PRINT 'Logical set to '; z$
20  END

RNH
Logical set to USER:[TESTER]
```

SUMMARY:

ASK SYSTEM, LOGICAL asks the operating system to translate the logical name in *str_expr1* and place the result into the variable specified by *str_var*.

SET SYSTEM, LOGICAL sets the operating system logical name in *str_expr1* to the value in *str_expr2*.

11.2.4 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
```

SUMMARY:

This statement returns the mode of the process which is one of the following:

INTERACTIVE
BATCH
NETWORK
OTHER

11.2.5 ASK SYSTEM: PARAMETER

FORMAT:

```
ASK SYSTEM: PARAMETER str_var
```

EXAMPLE:

```
 1  PROGRAM test_param.int
10  ASK SYSTEM: PARAMETER pdata$
20  PRINT 'Parameter was: '; pdata$
30  END

INTOUCH

SAVE 'test_param'

INTOUCH
                                parameter
                               ------------
                              |            |
$ INTOUCH/SOURCE test_param show parameter
Parameter was: SHOW PARAMETER

INTOUCH
```

SUMMARY:

ASK SYSTEM: PARAMETER returns any parameter given after the program name and places it in *str_var*.

11.2.6 ASK SYSTEM, PASS: SUCCESS

FORMAT:

```
ASK SYSTEM, PASS: SUCCESS int_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
```

SUMMARY:

ASK SYSTEM, PASS: SUCCESS asks the operating system to return TRUE (1) or FALSE (0) in *int_var* indicating whether or not the last PASS command executed successfully.

11.2.7 ASK | SET SYSTEM: PROCESS

FORMAT:

```
ASK SYSTEM: PROCESS str_var
SET SYSTEM: PROCESS str_expr
```

EXAMPLE:

```
10  ASK SYSTEM: PROCESS process$
    curr_process$ = process$
    PRINT 'Current process is: '; curr_process$
20  new_process$ = 'DO_TEST'
    SET SYSTEM: PROCESS new_process$
    ASK SYSTEM: PROCESS process$
    PRINT 'New process is: '; process$
30  SET SYSTEM: PROCESS curr_process$
    ASK SYSTEM: PROCESS process$
    PRINT 'Old process restored: '; process$
40  END
```

```
RNH
Current process is: TESTER_13B_3
New process is: DO_TEST
Old process restored: TESTER_13B_3
```

SUMMARY:

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

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

11.2.8 ASK SYSTEM: RIGHTS

FORMAT:

```
ASK SYSTEM: RIGHTS str_var
```

EXAMPLE:

```
10  ASK SYSTEM: RIGHTS process_rights$
    PRINT 'Your process rights are: '; process_rights$
20  END
```

```
RNH
Your process rights are: FAST_ACCESS,TEST_ACCESS
```

SUMMARY:

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

11.2.9 ASK | SET SYSTEM, SYMBOL: VALUE

FORMAT:

```
ASK SYSTEM, SYMBOL str_expr1: VALUE str_var
SET SYSTEM, SYMBOL str_expr1: VALUE str_expr2
```

EXAMPLES:

```
10  ASK SYSTEM, SYMBOL 'INTOUCH': VALUE i$
    PRINT 'Value of symbol INTOUCH is: '; i$
20  END
```

```
RNH
Value of symbol INTOUCH is: $intouch_image:
```

```
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
```

SUMMARY:

ASK SYSTEM, SYMBOL asks the operating system to translate the symbol name in *str_expr1* and place the result into the variable specified by *str_var*.

SET SYSTEM, SYMBOL sets the operating system symbol name in *str_expr1* to the value in *str_expr2*.

11.2.10 ASK SYSTEM: USER

FORMAT:

```
ASK SYSTEM: USER str_var
```

EXAMPLE:

```
10  ASK SYSTEM: USER uname$
20  PRINT 'User is: '; uname$
30  END
```

```
RNH
User is: TESTER
```

SUMMARY:

ASK SYSTEM: USER returns the operating system name or ID for the user. Under VMS this is the user name.

11.3 ASK | SET WINDOW

The ASK/SET WINDOW statements ask about and reset different screen features. ASK WINDOW and SET WINDOW have many options. The options are described in the following sections.

11.3.1 ASK | SET WINDOW AREA

FORMAT:

```
ASK WINDOW AREA row, col, row, col: DATA str_var
SET WINDOW AREA row, col, row, col: DATA str_expr
```

EXAMPLES:

```
10  CLEAR
20  PRINT AT 10, 4: 'Mary had a';
    PRINT AT 11, 4: 'little lamb';
30  ASK WINDOW AREA 10, 4, 11, 15: DATA x$
40  PRINT
    PRINT x$
50  END

RNH
   Mary had a
   little lamb
Mary had a
          little lamb


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
```

SUMMARY:

ASK WINDOW AREA reads the text displayed on the screen within the area defined by the given upper-left/lower-right coordinates into a string variable. The statement returns a <LF> delimited string. The coordinates are specified by upper-left row, upper-left column, lower-right row and lower-right column. No screen attributes are stored.

SET WINDOW AREA 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.

11.3.2 ASK | SET WINDOW: COLUMN

FORMAT:

```
ASK WINDOW: COLUMN int_var
SET WINDOW: COLUMN int_expr
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 5, 10:;
    ASK WINDOW: COLUMN cur_col
    PRINT 'Cursor is at column'; cur_col
20  SET WINDOW: COLUMN 4
    PRINT 'Hi!'
30  END

RNH
         Cursor is at column 10
   Hi!
```

SUMMARY:

ASK WINDOW: COLUMN returns the current column of the cursor's position.

SET WINDOW: COLUMN positions the cursor at the *int_expr* column within the current row.

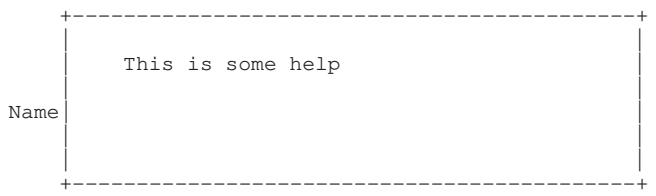
11.3.3 ASK | SET WINDOW: CURRENT

FORMAT:

```
ASK WINDOW: CURRENT str_var
SET WINDOW: CURRENT str_var
```

EXAMPLE:

```
10  CLEAR
    PRINT AT 1,20, UNDERLINE: 'Sample screen'
20  DO
      LINE INPUT 'Name', AT 5,1, LENGTH 30: name$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  _HELP  THEN
        ASK WINDOW: CURRENT old_w$
        CLEAR AREA BOX: 1, 5, 10, 50
        PRINT AT 3, 10, REVERSE: 'This is some help'
        DELAY
        SET WINDOW: CURRENT old_w$
        REPEAT DO
      END IF
30  END DO
40  END

RNH
```



SUMMARY:

ASK WINDOW: CURRENT saves the current screen window image.

SET WINDOW: CURRENT restores a saved screen window image.

11.3.4 ASK | SET WINDOW: DATA

FORMAT:

```
ASK WINDOW: DATA str_var
SET WINDOW: DATA str_expr
```

EXAMPLES:

```
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


10  CLEAR
20  x$ = 'Mary had a' + chr$(10) + 'little lamb'
30  SET WINDOW: DATA x$
40  END

RNH
Mary had a
little lamb
```

SUMMARY:

ASK WINDOW: DATA reads the text displayed on the whole screen into a string variable. The statement returns a <LF> delimited string. No screen attributes are stored.

SET WINDOW: DATA sets the screen to the specified string. This is the mirror image of ASK WINDOW: DATA str_var.

11.3.5 ASK | SET WINDOW: KEYMAP

FORMAT:

```
ASK WINDOW: KEYMAP str_var
SET WINDOW: KEYMAP str_expr
```

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
```

SUMMARY:

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.

11.3.6 SET WINDOW KEYSTROKE

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
```

SUMMARY:

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.

str_expr1 describes the name of the key that you want to change. It can be a single keystroke name, or a comma separated list of names. Keystroke names can be a single letter, or the name of the letter (such as TAB or CTRL/Z).

str_expr2 defines the new meaning of the keystroke. A keystroke meaning consists of one or two components. The keystroke value and the keystroke concept. For example, the CTRL/Z key usually has a value of CHR$(26), and the concept of EXIT. The keystroke value and/or the keystroke concept can be changed. If changing both a value and a concept, separate the two with a comma. You can restore the original meaning of the key by using " ".

The following concepts are supported:

Table 11–1 Supported Keystroke Concepts

Concept name	Description
_EXIT	an EXIT key
_BACK	a BACK key
_HELP	a HELP key
_IGNORE	ignore this keystroke
_INVALID	beep when pressed
_TERMINATOR	keystroke is a line terminator

11.3.7 ASK | SET WINDOW: ROW

FORMAT:

```
ASK WINDOW: ROW int_var
SET WINDOW: ROW int_expr
```

EXAMPLES:

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

RNH
Cursor is at row 5


10  CLEAR
20  SET WINDOW: row 3
30  PRINT 'Hi!'
40  END

RNH

Hi!
```

SUMMARY:

ASK WINDOW: ROW returns the current row of the cursor's position.

SET WINDOW: ROW positions the cursor at the *int_expr* row within the current column.

11.3.8 ASK | SET WINDOW: TYPEAHEAD

FORMAT:

```
ASK WINDOW: TYPEAHEAD str_var
SET WINDOW: TYPEAHEAD str_expr
```

EXAMPLES:

```
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 )
```

```
10  SET WINDOW: TYPEAHEAD 'FRED' + CHR$(13)
    INPUT 'Name': name$
    PRINT name$
20  END
```

```
RNH
Name? FRED
FRED
```

SUMMARY:

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.)

SET WINDOW: TYPEAHEAD puts data into the typeahead buffer as though the user had typed the data in from the terminal.

12 Loops, Conditionals, Chaining, Branching

12.1 Loops

A loop is a section of code that can be repeated. There are two types of loops. **FOR** loops execute a block of code a specific number of times. **DO** loops execute a block of code either: infinitely, until a specified condition is met, or while a specified condition remains true.

12.1.1 FOR . . . NEXT Loop

FORMAT:

```
FOR num_var = num_expr1 TO num_expr2 [STEP num_expr]
        ---
        ---  block of code
        ---
NEXT num_var
```

EXAMPLE:

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

```
RNH
Enter name? Jack
 1  Jack
Enter name? Tom
 2  Tom
Enter name? Sue
 3  Sue
Enter name? Toby
 4  Toby
Finished
Final value: 5
```

SUMMARY:

A **FOR** loop is used to execute a block of code a specified number of times. You might use this statement to count to ten, or to ask a question a certain number of times. INTOUCH counts from *num_expr1* to *num_expr2*, executing the loop each time. When the value of the *num_var* is less than *num_expr2*, INTOUCH stops executing the loop.

FOR loops can be nested.

STEP is used to change the increment of the loop. The default STEP is one.

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

RNH
Enter name? Fred
 1  Fred
Enter name? John
 4  John
Finished
Final value: 7
```

12.1.2 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

RNH
Your name, please? James
Hello, James
Your name, please? Marian
Hello, Marian
Your name, please? exit
```

SUMMARY:

If you want to exit from a FOR loop, use the **EXIT FOR** statement. EXIT FOR returns control to the first statement after the NEXT.

12.1.3 REPEAT FOR

FORMAT:

```
REPEAT FOR
```

EXAMPLE:

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

```
RNH
 1
Your name, please? George
Hello, George
 2
Your name, please?
 2
Your name, please? Sam
Hello, Sam
 3
Your name, please? Tom
Hello, Tom
```

SUMMARY:

REPEAT FOR unconditionally repeats all or part of a FOR loop. You can use REPEAT FOR if you want to repeat part of a FOR NEXT loop and you do not want to increment the index variable. REPEAT FOR transfers control to the first statement after the FOR.

12.1.4 ITERATE FOR

FORMAT:

```
ITERATE FOR
```

EXAMPLE:

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

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

SUMMARY:

You can use **ITERATE FOR** to skip part of a FOR loop. ITERATE FOR transfers control to the NEXT statement.

12.1.5 DO Loop

FORMAT:

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

EXAMPLE:

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

```
RNH
Your name, please? Sam
Hello, Sam
Your name, please? Sue
Hello, Sue
Your name, please? Bart
Hello, Bart
```

SUMMARY:

DO loops repeatedly execute a block of code either infinitely, or until a specified condition is met. **WHILE** and **UNTIL** can be used at either end of the loop to set up conditions.

DO loops can be nested.

WHILE causes the loop to be executed while an expression remains true.

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

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

UNTIL causes the loop to be executed until an expression becomes true.

```
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
```

12.1.6 EXIT DO

FORMAT:

```
EXIT DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please' : name$
      IF  _EXIT  THEN  EXIT DO
      PRINT 'Hello, '; name$
    LOOP
20  PRINT 'Finished'
30  END

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

SUMMARY:

If you want to exit from a DO loop, use the **EXIT DO** statement. EXIT DO transfers control to the first statement after the LOOP statement.

DO...END DO is a single iteration loop. The code between DO and END DO is processed only once unless conditional code specifies exiting or repeating the DO.

12.1.7 REPEAT DO

FORMAT:

```
REPEAT DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please': name$
      IF  _EXIT  THEN  EXIT DO
      IF  name$ = ''  THEN  REPEAT DO
      PRINT 'Hello, '; name$
    LOOP
20  END

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

SUMMARY:

REPEAT DO unconditionally repeats all or part of a DO loop. REPEAT DO transfers control to the first statement after the DO.

12.1.8 ITERATE DO

FORMAT:

```
ITERATE DO
```

EXAMPLE:

```
10  DO
      INPUT 'Your name, please': name$
      IF  _EXIT  THEN  EXIT DO
      IF  name$ = 'SKIP'  THEN  ITERATE DO
      PRINT 'Hello, '; name$
    LOOP
20  END

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

SUMMARY:

You can use **ITERATE DO** to skip part of a DO loop. ITERATE DO transfers control to the LOOP statement.

12.2 Conditionals

Conditionals are constructs which allow you to execute specific blocks of code depending on one or more conditions.

12.2.1 IF/THEN . . . END IF

FORMAT:

```
IF cond_expr THEN statement [ELSE statement]

                      or

IF cond_expr THEN
      ---
      ---  block of code
      ---
[ELSEIF expr THEN
      ---
      ---  block of code
      ---  ]
[ELSE
      ---
      ---  block of code
      ---  ]
END IF
```

EXAMPLE:

```
10  INPUT PROMPT 'Enter your age: ': age
    INPUT PROMPT 'Enter your sex: ': sex$
20  IF  UCASE$(sex$[1:1]) = 'M'  THEN  GOTO  done  ELSE PRINT
30  IF  age < 20  THEN
      PRINT 'Please go to line A.'
    ELSEIF  age > 19  AND  age < 40  THEN
      PRINT 'Please go to line B.'
    ELSE
      PRINT 'Please go to line C.'
    END IF
40  done: END
```

```
RNH
Enter your age: 25
Enter your sex: female

Please go to line B.
```

SUMMARY:

Use the **IF** construct when you want to execute a statement or a block of code, only under specific conditions. If the condition specified is met, the code following the **THEN** is executed. If the condition is not met, either the statement is skipped, or the code following the **ELSE** is executed. **ELSEIF** checks the expression and executes the statement or block of code following the **THEN**.

12.2.2 SELECT CASE . . . END SELECT

FORMAT:

```
SELECT CASE main expr
CASE expr1[, expr2,...]
     ---
     ---  block of code
     ---
[CASE expr3[, expr4,...]
     ---
     ---  block of code
     ---              ...]
[CASE IS {numeric function | boolean function} expr
     ---
     ---  block of code
     ---  ]
[CASE ELSE
     ---
     ---  block of code
     ---  ]
END SELECT
```

EXAMPLE:

```
10  DO
      INPUT 'Procedure (ADD, CHG, DEL)': pro$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      pro$ = UCASE$(pro$)       ! convert to uppercase
20    SELECT CASE pro$
      CASE 'ADD'
        PRINT 'Adding...'
      CASE 'CHA', 'CHANGE'
        PRINT 'Changing...'
      CASE 'DEL', 'DELETE'
        PRINT 'Deleting...'
      CASE ELSE
        MESSAGE ERROR: 'Procedures are: add, del or exit'
        REPEAT DO
      END SELECT
    LOOP
30  END
```

```
RNH
Procedure (ADD, CHG, DEL)? change
Changing...
Procedure (ADD, CHG, DEL)? del
Deleting...
Procedure (ADD, CHG, DEL)? exit
```

SUMMARY:

There are times when you want to execute different blocks of code depending on different conditions. **SELECT CASE** is used to check a set of conditions and execute code depending on the results. SELECT CASE executes the block of code following the **CASE** expression which matches the main expression. **CASE IS** forms a conditional expression. INTOUCH executes the CASE IS code if the expression is TRUE.

12.3 Branching

Branching means jumping from one part of a program to another. Branching is useful when combined with conditionals.

Branching and Code Blocks

Certain constructs (such as GOSUB) include a block of code which is executed within the construct. You can transfer control within these code blocks and out of these code blocks. For example:

```
                    10  DIM name$(4)
                    20  FOR i = 1 TO 4
                          INPUT 'Your name, please': name$(i)
branches within block --    IF  name$(i) <> ''  THEN  GOTO hello
branches out of block --    IF  _EXIT  THEN  EXIT FOR
                          hello:
                          PRINT 'Hello!'
                    30  NEXT i
                    40  PRINT 'Finished'
                    50  END
```

12.3.1 GOTO

FORMAT:

```
GOTO target
```

EXAMPLE:

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

```
RNH
Your Name? Tony
Hello, Tony
Your Name? Betty
Hello, Betty
Your Name? exit
```

SUMMARY:

Use **GOTO** when you want to jump from one part of your program to another. GOTO transfers control to the line number or alphanumeric label specified.

12.3.2 ON . . . GOTO . . . ELSE

FORMAT:

```
ON int_expr GOTO target [,target...] [ELSE statement]
```

EXAMPLE:

```
10  start:
    INPUT 'Procedure (1=add, 2=del, 3=exit)': pro
20  ON pro GOTO add, del, done ELSE PRINT 'Enter 1, 2 or 3'
    GOTO start
    add:
      PRINT 'Adding...'
      GOTO start
    del:
      PRINT 'Deleting...'
      GOTO start
30  done:
    PRINT 'Finished'
    END
```

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

SUMMARY:

ON GOTO lets you jump to one of several locations depending on a condition you set up. ON GOTO transfers control to the target specified by the integer expression (a 1 indicates the first target, 2 is the second target and so on).

12.3.3 GOSUB . . . RETURN

FORMAT:

```
GOSUB target
        .
        .
        .
target
       ---
       ---  block of code
       ---
RETURN
```

EXAMPLE:

```
10  GOSUB get_input
20  PRINT 'Hello, '; name$
    GOTO exit
30  get_input:
    INPUT 'Enter your name': name$
    name$ = UCASE$(name$)
    RETURN
40  exit:
    PRINT 'Finished'
50  END

RNH
Enter your name? Julian
Hello, JULIAN
Finished
```

SUMMARY:

Use **GOSUB** to jump to another location in your program, execute the code at that location and then return to the place you left. The **RETURN** statement sends INTOUCH back to the first statement following the matching GOSUB.

Note

If your label or routine name contains an underscore (_) character (e.g. ask_name, do_add), you do NOT need to use the word "GOSUB". Under these conditions, the "GOSUB" is implied. For example: using ASK_NAME would be the same as using GOSUB ASK_NAME

12.3.4 ON . . . GOSUB

FORMAT:

```
ON int_expr GOSUB target, target... [ELSE statement]
       .
       .
       .
target
      ---
      ---  block of code
      ---
RETURN

[target
      ---
      ---  block of code
      ---
RETURN
      ...]
```

EXAMPLE:

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

RNH
Procedure (1=add, 2=del, 3=exit)? add
Non-numeric input when number expected at START
Procedure (1=add, 2=del, 3=exit)? 5
Enter 1, 2 or 3
Procedure (1=add, 2=del, 3=exit)? 1
Adding...
Procedure (1=add, 2=del, 3=exit)? 3
Finished
```

SUMMARY:

Use **ON...GOSUB** to jump to one of several locations depending on a condition you set up. ON..GOSUB jumps to one of several locations, executes the code at that location and then returns to the first statement after the ON...GOSUB.

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

12.4 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
```

SUMMARY:

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.

12.5 DISPATCH

FORMAT:

```
DISPATCH str_expr
      .
      .
      .
target
     ---
     --- block of code
     ---
RETURN
```

EXAMPLE:

```
10  INPUT 'Routine name', DEFAULT 'add': routine$
    DISPATCH routine$
    STOP
20  add:
      PRINT 'Adding information...'
    RETURN

30  change:
      PRINT 'Changing information...'
    RETURN

40  END

RNH
Routine name? add
Adding information...
```

SUMMARY:

DISPATCH executes a routine that the program determines at run-time.

DISPATCH looks at the contents of the string expression (*str_expr*), searches for a routine with that name and GOSUBS to the routine.

str_expr is the name of the subroutine to execute.

12.6 ROUTINE/END ROUTINE

FORMAT:

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

EXAMPLE:

```
10      get_username
20      END

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

RNH
Username: Tester
```

SUMMARY:

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.

12.6.1 EXIT ROUTINE

FORMAT:

```
EXIT ROUTINE
```

SUMMARY:

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

12.6.2 REPEAT ROUTINE

FORMAT:

```
REPEAT ROUTINE
```

SUMMARY:

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

12.7 Chaining Programs

12.7.1 CHAIN

FORMAT:

```
CHAIN str_expr
```

EXAMPLE:

```
10  LINE INPUT 'Your name (last, first)': name$
    OPEN #1: NAME 'storage.tmp', ACCESS OUTPUT
    PRINT #1: name$
    CLOSE #1
20  INPUT 'Add to CLIENT structure (Y/N)': reply$
    IF  reply$ = 'Y'  THEN  CHAIN 'ADD'
30  END

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

SUMMARY:

Use the **CHAIN** statement to exit the current program and run another program. When INTOUCH executes the CHAIN statement, it ends the current program and runs the chained program.

12.8 Pass Commands to the Operating System

12.8.1 PASS [NOWAIT: | NORETURN:]

FORMAT:

```
PASS [NOWAIT: | NORETURN:] str_expr
```

EXAMPLE:

```
10  INPUT 'What would you like to show': info$
20  PASS 'SHOW ' + info$
30  END

RNH
What would you like to show? user t

      VAX/VMS User Processes at  9-MAY-1995 10:22:29.87
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER       TTI            1
```

SUMMARY:

Use **PASS** to perform system commands without leaving the INTOUCH environment or exiting a program.

PASS passes the specified string expression to the operating system command interpreter. Generally, it passes the string to DCL. The operating system will respond to the string as it would if you entered it at the DCL system 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.

```
10  PASS NOWAIT: 'show user t'
20  END

INTOUCH

RNH

INTOUCH

      VAX/VMS User Processes at  9-MAY-1995 10:23:19.87
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER       TTI            1

end

INTOUCH
```

The **PASS NORETURN** statement passes a command to the operating system but does NOT return to INTOUCH.

```
10  PRINT 'B E F O R E'
20  DELAY 2
30  PASS NORETURN: 'show user t'
40  END

RNH
B E F O R E

      VAX/VMS User Processes at  9-MAY-1995 01:43:14.36
    Total number of users = 1,  number of processes = 1

 Username  Node     Interactive  Subprocess   Batch
 TESTER      TTI            1
$
```

13 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.

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

13.1 OPEN #chnl_num:

FORMAT:

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

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS OUTPUT
    PRINT #1: 'This is the first line of text.'
    PRINT #1: 'This is the second line of text.'
20  CLOSE #1
30  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
    LINE INPUT #1: line_1$, line_2$
    PRINT line_1$
    PRINT line_2$
40  CLOSE #1

RNH
This is the first line of text.
This is the second line of text.
```

SUMMARY:

OPEN opens a file so you can read from it and write to it or to create a new file. **#chnl_num:** is the channel number associated with the file **NAME** and gives the file specification of the file being opened. The file specification can be any string expression.

The default line length of a file is 132 characters. If you want the line to be some other length, use the SET #chnl_num: MARGIN num_expr statement to change the line length.

The OPEN statement has the following options:

- **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.

- **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.

- **OPTIMIZE OFF**

  Opens a file without any INTOUCH I/O optimizations.

- **UNIQUE**

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

```
10  OPEN #12: UNIQUE, ACCESS OUTPUT
20  ASK #12: NAME x$
30  PRINT x$
40  CLOSE #12
50  END

RNH
SYS$SCRATCH:TMP_B90001.TMP
```

13.2 CLOSE #chnl_num

FORMAT:

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

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS OUTPUT
    PRINT #1: 'This is the first line of text.'
    PRINT #1: 'This is the second line of text.'
20  CLOSE #1
30  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
    LINE INPUT #1: line_1$, line_2$
    PRINT line_1$
    PRINT line_2$
40  CLOSE ALL

RNH
This is the first line of text.
This is the second line of text.
```

SUMMARY:

CLOSE #chnl_num closes a file. **CLOSE ALL** closes all files. *#chnl_num* is the channel number associated with the file in the OPEN statement.

13.3 PRINT #chnl_num

FORMAT:

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

EXAMPLE:

```
10  OPEN #1: NAME 'test.tmp', ACCESS OUTPUT
    PRINT #1, USING '{UCASE}?': 'first line'
    PRINT #1: TAB(5); 'second line'
    CLOSE #1
20  OPEN #1: NAME 'test.tmp', ACCESS INPUT
    LINE INPUT #1: record_1$, record_2$
    PRINT record_1$
    PRINT record_2$
    CLOSE #1
30  END

RNH
FIRST LINE
    second line
```

SUMMARY:

PRINT stores data in a file so you can use it even after your program ends. **chnl_num** is the channel number you associate with the file in the OPEN statement. 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. **TAB** causes the next item to be printed at the position specified. The **USING** option is used to format data.

13.4 INPUT #chnl_num

FORMAT:

```
INPUT #chnl_num: var, var...
```

EXAMPLE:

```
10  OPEN #1: NAME 'number.dat', ACCESS OUTPUT
    PRINT #1: 1; 2; 3
    CLOSE #1
20  OPEN #1: NAME 'number.dat', ACCESS INPUT
    INPUT #1: a, b, c
    PRINT a; b; c
30  CLOSE #1
40  END

RNH
 1  2  3
```

SUMMARY:

INPUT gets data from a file so you can use it in your program. *#chnl_num* is the channel number associated with the file in the OPEN statement.

13.5 LINE INPUT #chnl_num

FORMAT:

```
LINE INPUT #chnl_num [, EOF num_var]: str_var, str_var...
```

EXAMPLE:

```
10  OPEN #1: NAME 'test_file.tmp', ACCESS INPUT
20  LINE INPUT #1: line_1$, line_2$
    PRINT line_1$, line_2$
30  CLOSE #1
40  END

RNH
This is the first line of text.         This is the second line of text.
```

SUMMARY:

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 **EOF option** of LINE INPUT causes INTOUCH to return a TRUE/FALSE value indicating when the end of the file has been reached. This eliminates the need for an error handler when reading files.

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

RNH
line was: This is the first line of text.
line was: This is the second line of text.
```

_CHANNEL is the next available channel number.

13.6 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
    ASK #3: MARGIN m
    ASK #3: NAME n$
    ASK #3: CURRENT c$
    PRINT 'The current print zone width is'; x
    PRINT 'The current margin is'; m
    PRINT 'The file specification is: '; n$
    PRINT 'The current record value is: '; c$
30  CLOSE #3
40  END

RNH
The current print zone width is 20
The current margin is 132
The file specification is: USER:[TESTER]STORAGE.DAT
The current record value is:  (none--this is a new output file)
```

SUMMARY:

ASK returns a characteristic of a device and stores the value in a numeric or string variable. The ask option indicates which characteristic is checked. The ask options are:

- ZONEWIDTH num_var

  Assigns the device's zonewidth to the numeric variable specified.

- MARGIN num_var

  Assigns the device's current right margin setting to the numeric variable specified.

- CURRENT str_var

  Use to store a current record value into the str_var.

- NAME str_var

  Asks the INTOUCH operating system for the file specification of the file open on channel #chnl_num and stores the value into the str_var.

13.7 SET #chnl_num:

FORMAT:

```
SET # chnl_num: [ ZONEWIDTH num_expr ] [, MARGIN int_expr]
                [, CURRENT str_expr]
```

EXAMPLE:

```
10  OPEN #3: NAME 'storage.dat', ACCESS OUTPUT
20  ASK #3: ZONEWIDTH x
    PRINT 'The current print zone width is'; x
    SET #3: ZONEWIDTH 35
    ASK #3: ZONEWIDTH x
    PRINT 'The new print zone width is'; x
30  CLOSE #3
40  END

RNH
The current print zone width is 20
The new print zone width is 35
```

SUMMARY:

SET is used to set various characteristics from within your INTOUCH programs. The set option indicates which characteristic is set. The set options are:

- ZONEWIDTH num_expr

  Sets a device's zonewidth to the number specified.

- MARGIN num_expr

  Sets a device's right margin to the number specified.

- CURRENT str_expr

  Sets the current record to that specified by the str_expr.

13.8 Deleting Files

13.8.1 KILL

FORMAT:

```
KILL str_expr
```

EXAMPLE:

```
5   ! DANGER -- Executing this program will DELETE a file

10  INPUT 'What file do you want to delete': file$
20  KILL file$
30  END

RNH
What file do you want to delete?
```

SUMMARY:

Use **KILL** to delete a file you no longer want. KILL deletes the specified file.

14 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.

14.1 CAUSE EXCEPTION

FORMAT:

```
CAUSE EXCEPTION exception_number
```

EXAMPLE:

```
10  DO
      INPUT 'Select a number between 1 and 10': no
      IF  no < 1  OR  no > 10  THEN  CAUSE EXCEPTION 1001
      REPEAT DO
    END DO
20  END
```

```
RNH
Select a number between 1 and 10? 8
Select a number between 1 and 10? 99
Illegal number at 10.2
```

SUMMARY:

CAUSE causes the exception specified to occur. *exception_number* is an integer expression which specifies the number of the exception to generate.

The appendix section of the *INTOUCH - A Guide to the INTOUCH Language* manual contains a list of exception messages.

14.2 WHEN EXCEPTION IN

FORMAT:

```
WHEN EXCEPTION IN
   ---
   ---  protected block
   ---
USE
   ---
   ---  handler
   ---
END WHEN
```

EXAMPLE:

```
10  INPUT 'Your name, please': name$
    WHEN EXCEPTION IN
      INPUT 'How old are you': age
    USE
      PRINT 'Not a valid age'
      RETRY
    END WHEN
    PRINT
    PRINT NAME$; ' is'; age
20  END

RNH
Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

SUMMARY:

WHEN EXCEPTION IN is used to catch run-time exceptions within your program and resolve them. If an exception occurs in the protected block following the WHEN EXCEPTION IN statement, the handler following the **USE** statement is executed. **END WHEN** ends the handler.

14.3 WHEN EXCEPTION USE

FORMAT:

```
WHEN EXCEPTION USE handl_name
   ---
   ---  protected block
   ---
END WHEN
    .
    .
    .
HANDLER handl_name
   ---
   ---  handler
   ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
    WHEN EXCEPTION USE fix_average
      average = tsales/nsales
    END WHEN
    PRINT 'The average is:'; average

20  HANDLER fix_average
      average = 0
      CONTINUE
30  END HANDLER
40  END
```

```
RNH
Enter total sales amount? 25.00
Enter number of sales? 0
The average is: 0
```

SUMMARY:

WHEN EXCEPTION USE is used to catch run-time exceptions within your program and resolve them. WHEN EXCEPTION USE calls a routine to solve the problem. This routine can be used by any number of WHEN EXCEPTION USE statements.

HANDLER begins the handler routine. Everything between the HANDLER and the **END HANDLER** constitutes the handler routine. END HANDLER returns control to the statement following the END WHEN. When the handler is called, this block of code is executed.

14.4 HANDLER Routine Actions

14.4.1 EXIT HANDLER

FORMAT:

```
WHEN EXCEPTION [USE  | IN] handl_name
   .
   .
   .
USE
  ---
  ---  EXIT HANDLER
  ---
END WHEN

         or

HANDLER handl_name
  ---
  ---  EXIT HANDLER
  ---
END HANDLER
```

EXAMPLE:

```
10  WHEN EXCEPTION USE mistake
      INPUT 'Enter your age': age
    END WHEN
    PRINT 'You are'; age; 'years old'

20  HANDLER mistake
      PRINT 'Oops...'
      DELAY 2
      EXIT HANDLER
    END HANDLER
30  END
```

```
RNH
Enter your age? 3x
Oops...
Non-numeric input when number expected at 10.1
Enter your age? 35
You are 35 years old
```

SUMMARY:

EXIT HANDLER allows you to exit the current handler routine. Use EXIT HANDLER to leave the handler when you have solved the exception. Control will be transferred to the outer exception handler or–if there is no outer exception handler–to the INTOUCH system.

14.4.2 RETRY

FORMAT:

```
WHEN EXCEPTION [USE  | IN] handl_name
   .
   .
   .
USE
  ---
  ---  RETRY
  ---
END WHEN

        or

HANDLER handl_name
  ---
  ---  RETRY
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Your name, please': name$
20  WHEN EXCEPTION IN
      INPUT 'How old are you': age
    USE
      PRINT 'Not a valid age'
      RETRY
    END WHEN
30  PRINT
    PRINT name$; ' is'; age
40  END

RNH
Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

SUMMARY:

Use **RETRY** to re-execute the statement that generated the exception. RETRY can only be used in a handler routine.

14.4.3 CONTINUE

FORMAT:

```
WHEN EXCEPTION [USE  | IN] handl_name
   .
   .
   .
USE
  ---
  ---  CONTINUE
  ---
END WHEN

         or

HANDLER handl_name
  ---
  ---  CONTINUE
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
20  WHEN EXCEPTION USE fix_average
      average = tsales / nsales
    END WHEN
30  PRINT 'The average is:'; average

40  HANDLER fix_average
      average = 0
      CONTINUE
    END HANDLER
50  END

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

SUMMARY:

CONTINUE continues normal program execution at the statement following the statement which generated the exception. CONTINUE can only be used within a handler.

14.4.4 RESUME

FORMAT:

```
WHEN EXCEPTION [USE  | IN] handl_name
   .
   .
   .
USE
  ---
  ---  RESUME target
  ---
END WHEN

         or

HANDLER handl_name
  ---
  ---  RESUME target
  ---
END HANDLER
```

EXAMPLE:

```
10  INPUT 'Enter total sales amount': tsales
    INPUT 'Enter number of sales': nsales
20  WHEN EXCEPTION USE fix_average
      average = tsales / nsales
    END WHEN
30  PRINT 'The average is:'; average

40  HANDLER fix_average
      average = 0
      PRINT 'Invalid numbers.  Try again.'
      RESUME 10
    END HANDLER
50  END

RNH
Enter total sales amount? 75.00
Enter number of sales? 0
Invalid numbers.  Try again.
Enter total sales amount? 75.00
Enter number of sales? 3
The average is: 25
```

SUMMARY:

RESUME resumes normal program execution at the line or label you specify. RESUME can only be used within a handler routine.

15 Program Elements and Data in INTOUCH

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

15.1 INTOUCH Program Elements

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

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

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

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

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

Example 15–1: Executing an INTOUCH program

```
INTOUCH

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

INTOUCH

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

INTOUCH
```

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

When you run the above program, you will notice that the day is not asked for. **DAY$** is a reserved identifier that INTOUCH uses to store the day in. There are several identifiers whose use is reserved by the INTOUCH system. They are the following:

```
    DATE    EXLINE   EXTYPE   MAXNUM   PI    TIME
    DATE$   TIME$    ELSE     PRINT    REM   RND

The following will be reserved for future versions of INTOUCH:

    NUL$    CON     IDN    ZER     TRANSFORM
```

You can override INTOUCH supplied identifiers, function names, etc., by redefining their meanings.

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

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

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

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

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

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

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

RNH
 4
 4.5
 5
 5.5
 6
 6.5
 7
```

Note

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

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

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

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

15.1.1 Continuing Program Lines

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

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

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

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

Note

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

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

15.1.2 Labels

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

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

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

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

15.1.3 Storing INTOUCH Programs

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

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

15.2 Comments

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

There are two types of comments allowed in INTOUCH: REM comments and exclamation points (!).

15.2.1 REM

FORMAT:

```
REM comment_text
```

EXAMPLE:

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

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

SUMMARY:

Use **REM** to put remarks on program lines. You can use these remarks to clarify your code. When a program is listed on the screen or printed out, the REM lines are displayed exactly as they were written in the source code.

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

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

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

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

15.2.2 ! comment_text

FORMAT:

```
! comment_text
```

EXAMPLE:

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

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

SUMMARY:

Use the **!** to put comments in a program. You can use comments to clarify parts of your program as shown in the example above. When the program is listed or printed, the "!" line is displayed as it was written in the source code.

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

The Exclamation Point with Line Continuation

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

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

15.3 Data in INTOUCH

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

15.3.1 Data Types

INTOUCH accepts three basic types of data:

1 integer numbers

2 real numbers

3 and string (textual) data

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

15.3.1.1 Integers

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

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

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

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

15.3.1.2 Real Numbers

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

```
5.4             894.0             -369004
```

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

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

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

15.3.1.3 String Data

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

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

About string data:

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

15.3.2 BOOLEAN

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

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

15.3.3 Expressions

Expressions can be:

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

15.3.4 Constants

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

15.3.4.1 Numeric constants

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

```
235             412%
```

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

15.3.4.2 E Notation

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

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

Table 15–1 E Notation

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

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

```
+1E3
```

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

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

15.3.4.3 String Constants

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

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

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

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

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

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

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

The string part of this is:

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

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

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

The string part of this is

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

15.3.5 Variables

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

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

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

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

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

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

Some examples of valid variables are:

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

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

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

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

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

```
INDEX           Z
COUNTER         AMOUNT
```

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

15.3.5.1 Arrays

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

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

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

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

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

About arrays:

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

- You can indicate the low bound explicitly with a TO clause:
  DIM COSTS(5 TO 10)

15.3.5.2 Substrings

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

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

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

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

RNH
IJKLMN
```

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

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

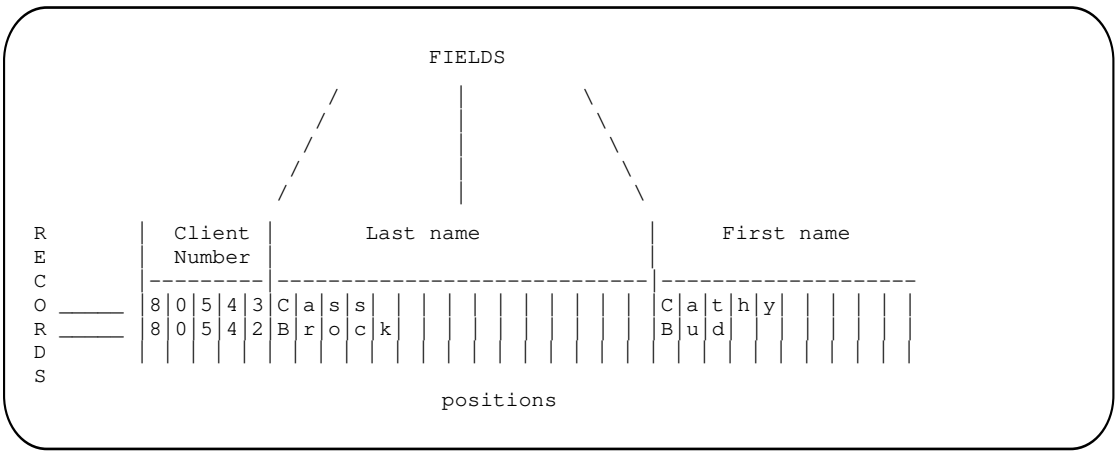
```
10  LET a$ = 'Your tests are in.'
20  PRINT a$
30  LET a$[6:10] = 'results'
40  PRINT a$
50  END

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

15.3.6 Structure References

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

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

Example 15–2: Structures



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

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

For information on creating structures and defining fields, see Chapter 1, Using SETUP with Structures, Fields.

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

```
struc_name(field_expr)
```

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

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

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

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

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

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

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

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

15.3.7 Multiple Occurrence Fields

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

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

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

RNH
10010 Sunset Cliffs Blvd.

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

#2 Bougainvillea Blvd.
```

15.3.8 Compound Expressions

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

Numeric expressions
String expressions
Conditional expressions

15.3.8.1 Numeric Expressions

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

```
                    Constants    Variables

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

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

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

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

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

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

15.3.8.2 String Expressions

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

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

RNH
MOTHER
```

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

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

RNH
This is it.
```

15.3.8.3 Conditional Expressions

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

Conditional Numeric Expressions

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

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

X and Y can be any unconditional or conditional expression.

Performing Relational Operations on Strings

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

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

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

RNH
TEST is less than TEXT
```

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

Logical Operators

The logical operators are:

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

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

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

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

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

Bit Manipulation

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

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

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

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

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

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

RNH
 5
```

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

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

15.3.9 Order of Evaluation

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

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

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

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

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

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

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

2. INTOUCH performs functions second.
3. INTOUCH performs exponentiation.
4. INTOUCH performs multiplication and division.
5. INTOUCH performs addition and subtraction.
6. INTOUCH performs relational operations from left to right. (The relational operators are: =, <, >, <=, >= and <>.) The only exception is the assignment of the result. The result is always assigned last.
7. INTOUCH performs logical operations in the following order:

   NOT
   AND
   OR
   XOR
   IMP
   EQV

16 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.

16.1 LIBRARY

FORMAT:

```
LIBRARY 'libr_name'
```

EXAMPLE:

```
10  LIBRARY 'BASRTL'
20  PRINT 'Waiting 5 seconds...'
30  CALL BAS$SLEEP(5% BY VALUE)
40  END

RNH
Waiting 5 seconds...
```

DESCRIPTION:

You can use the **LIBRARY** statement to specify the libraries you will use in your program.

The LIBRARY statement specifies libraries from which you CALL routines. The routines can be written in any VMS language that supports the standard calling interface (FORTRAN, BASIC, COBOL, etc.).

libr_name is the file specification of a library. The library can be one of the VMS supplied libraries, or a user-defined library. You must name a library with the LIBRARY statement before you call a routine from it. The library name must be a string constant in quotes.

To create libraries for INTOUCH, you can refer to Appendix B, Creating Libraries for Use with INTOUCH.

16.2 CALL

FORMAT:

```
CALL routine_name(arg [BY pass_mech], arg...)
```

EXAMPLE:

```
10  LIBRARY 'BASRTL'
20  PRINT 'Waiting 5 seconds...'
30  CALL BAS$SLEEP(5% BY VALUE)
40  END

RNH
Waiting 5 seconds...
```

DESCRIPTION:

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

The library to which the routine belongs must have been specified in a LIBRARY statement.

routine_name is the name of the routine you are calling. Some routines take arguments. *arg* is an argument you are passing to or getting from the routine. If more than one argument is passed, separate the arguments with commas:

```
CALL routine_name(arg, arg, arg...)
```

16.2.1 Optional Arguments

Some routines have optional arguments–arguments which **can** be used, but do not have to be used. To pass an optional argument, include it in the appropriate place in the argument list. If you do not want to pass an optional argument, specify that no argument is to be used by typing two commas side by side at the appropriate place in the argument list.

For example, the routine LIB$GET_INPUT gets a record of ASCII text. Its argument list contains two optional arguments:

```
LIB$GET_INPUT(get-str.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
                  |                 |                 |
    string variable the      a prompt string       length of the
    input is assigned to                          string variable
```

To input a line without using a prompt string, your CALL statement would have the following format:

```
CALL routine_name(str_var,,num_var)
```

The double commas tell INTOUCH that the optional variable *prompt-str* is not used. For example:

```
10  LIBRARY 'LIBRTL'
    BUF$ = REPEAT$(' ', 200)
    CALL LIB$GET_INPUT(BUF$, , OUT_LEN%)
    PRINT 'You typed: ';BUF$[1:OUT_LEN%]
20  END
```

16.2.2 Passing Mechanisms

pass_mech refers to the mechanism by which values are passed to the routine. The default passing mechanism for reals and integers is by **reference**. The default passing mechanism for strings is by **descriptor**. Here is an explanation of the passing mechanisms available:

BY REF By reference. This is the default passing mechanism for real and integer data. Arguments passed by reference can be changed by the routine they are passed to.

BY VALUE By value. Arguments passed by value cannot be changed by the routine they are passed to.

BY DESC By descriptor. This is the default passing mechanism for strings. By descriptor causes INTOUCH to use the argument's descriptor when passing the argument. Arguments passed by descriptor can be changed by the routine they are passed to.

The system functions _REAL and _INTEGER can be used in conjunction with the library statements. These functions return resulting data associated with the CALL. (See Section A.1, System Functions for more information.)

A Built-in Functions

INTOUCH has numerous built-in functions. This appendix describes the system and other built-in functions.

A.1 System Functions

The following functions return information about the system and user actions:

_BACK

Returns a TRUE or FALSE value. TRUE if the backslash key was pressed at the last prompt.

_CHANNEL

Returns the next available channel number.

```
10  out_ch = _CHANNEL
    OPEN #out_ch: NAME 'sys$scratch:test.lis'
    PRINT #out_ch: 'This is a test'
    CLOSE #out_ch
20  END
```

_DEBUG

Returns a TRUE or FALSE value. TRUE if DEBUG is on. FALSE if DEBUG is off.

_EXIT

Returns a TRUE or FALSE value. TRUE if EXIT was entered at the last prompt.

_EXTRACTED

Tells how many records were extracted in the last extract.

```
10  OPEN STRUCTURE cl: NAME 'tti_run:client'
20  EXTRACT STRUCTURE cl
30    INCLUDE cl(state) = 'CA'
40  END EXTRACT
50  PRINT 'Number of California clients:'; _EXTRACTED
60  END
```

_HELP

Returns a TRUE or FALSE value. TRUE if HELP or a question mark (?) was entered or the Help key is pressed at the last prompt.

_INTEGER

Sometimes when the user does a CALL, an integer value is returned as a result of the CALL. _INTEGER returns any resulting integer data.

```
10  LIBRARY 'LIBRTL'
20  CALL LIB$ICHAR('Q')
30  PRINT 'The integer value of "Q" is: '; _INTEGER
40  END
```

_REAL

Some routines return a real value as the result of the CALL. The _REAL function returns any resulting real data.

```
10  LIBRARY 'MTHRTL'
20  CALL MTH$ALOG(45.3)
30  PRINT 'The log is:'; _REAL
40  END
```

_REPLY

Returns the user's reply to the last prompt. The reply is returned as a string.

_STATUS

Returns the value given to INTOUCH by the operating system, for the last operating system request. This function is useful when debugging system level errors. The _STATUS function is often used with the SYSTEXT$ function.

_STRING

Upon the completion of an INPUT MENU statement, the concept _STRING contains the menu path taken by the user when selecting the menu item. (i.e. "#2;#3" means the 3rd item of the 2nd submenu.)

_TERMINATOR

The _TERMINATOR function returns the name of the key that terminated the last INPUT statement. The values returned are:

UP	Up arrow
DOWN	Down arrow
RETURN	Return
FIND	Find (find line, word)
INS	Insert
SEL	Select
PREV	Prev Screen
NEXT	Next Screen
REM	Remove
HELP	Help
DO	Do

PF1, PF2, PF3, PF4

KPO, KP1, KP2, KP3, KP4, KP5, KP6, KP7, KP8, KP9, MIN, COM, ENT, PER

F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, HELP, DO, F17, F18, F19, F20

A.2 Other Functions

The other non-system functions are:

ABS(num_expr)

ABS returns the absolute value of a specified numeric expression.

ACOS(num_expr)

ACOS(x) returns the angle whose COS is x. The angle is returned in radians.

ANGLE(num_expr, num_expr)

Given X and Y coordinates, returns the angle from 0,0. The angle is returned in radians.

ASCII(str_expr)

The ASCII function returns the decimal ASCII value of a string's first character. It is returned as an integer.

ASIN(num_expr)

ASIN(x) returns the angle whose SIN is x. The angle is returned in radians.

ATN(num_expr)

ATN returns the angle, in radians, of a specified tangent.

CEIL(num_expr)

CEIL(x) returns the ceiling of x. That is, the smallest integer that is not less than x.

CHANGE$(str_expr1, str_expr2, str_expr3)

CHANGE$ changes specified characters in str_expr1. Str_expr1 is the source string. Str_expr2 contains the target characters, and str_expr3 specifies the substitution characters. CHANGE$ returns the changed string.

CHANGE$ searches for the target characters within the source string and replaces these characters with the substitution characters. The substitution characters are mapped onto the target characters.

```
10  LET a$ = 'bdbdbdbd'
20  LET b$ = 'b'
30  LET c$ = 'c'
40  LET changed$ = CHANGE$(a$, b$, c$)
50  PRINT a$
60  PRINT changed$
70  END

RNH
bdbdbdbd
cdcdcdcd
```

CHARSET$[(str_expr)]

Returns the character set specified. The optional string expression can be used to specify the character set to return. The available character sets are:

UCASE	all upper-case letters (A-Z)
LCASE	all lower-case letters (a-z)
CONTROL	all control characters (ASCII 0-31)
ASCII	the ASCII character set, in order (0-255)

ASCII is the default character set for CHARSET$.

```
10  LINE INPUT 'Enter your text': text$
20  ! Change upper-case to lower-case
    ct$ = CHANGE$(text$, &
         CHARSET$('UCASE'), &
         CHARSET$('LCASE'))
    PRINT 'Lower-case version is:'; ct$
30  END
```

CHR$(int_expr1[, int_expr2])

CHR$ returns a string with the specified ASCII value (int_expr1) repeated the specified number of times (int_expr2). If no count is specified, a default count of one is used.

CONVERT(str_expr)

Given a string containing a mapped integer, this function returns the integer value.

```
10  a$ = 'AB'
    b = CONVERT(a$)
20  END

RNH
16961
```

CONVERT$(int_expr1 [[, int_expr2], int_expr3])

Given an integer (int_expr1) and an optional length (int_expr2), which defaults to four, this function returns a string mapping of the integer.

If the optional data type (int_expr3) is 17, the returned string will be a packed floating (PF).

The following data types are supported:

Table A–1 CONVERT$ Function - Supported Data Types

Data Type	Conversion Result
1	Integer (2 or 4 byte)
7	COBOL comp-3 (C3 packed decimal)
17	Packed floating (PF)

```
10  a$ = CONVERT$(16961)
    PRINT a$
20  END

RNH
AB
```

The CONVERT and CONVERT$ functions can be used in situations such as building segmented keys consisting of multiple data types.

COMPARE(str_expr1, str_expr2)

The COMPARE function compares two strings and returns a numeric value ranging from 0 (no match) to 100 (an exact match).

```
10  options$ = 'LEFT,RIGHT,UP,DOWN'
    best = 0
    best$ = ''
    INPUT 'Enter an option': usr_opt$
20  FOR idx = 1 TO ELEMENTS(options$)
      opt$ = ELEMENT$(options$, idx)
      score = COMPARE(opt$, usr_opt$)
      IF  score > best  THEN
        best = score
        best$ = opt$
      END IF
    NEXT idx
30  SELECT CASE best
    CASE 0
      PRINT 'Unknown option: '; usr_opt$
    CASE 100
      PRINT 'Option okay, it was: '; usr_opt$
    CASE ELSE
      PRINT 'Misspelled option: '; usr_opt$
      PRINT USING 'Did you mean ? (## percent)': best$, best
    END SELECT
40  END

RNH
Enter an option? dwn
Misspelled option: dwn
Did you mean DOWN (92 percent)
```

COS(num_expr)

COS returns the cosine of an angle the user specifies in radians.

COSH(num_expr)

COSH returns the hyperbolic cosine of a passed real number.

COT(num_expr)

COT(X) is equivalent to 1/TAN(X).

CPAD$(text_str, size [, pad_str])

CPAD$ returns a new string, padded on the left and on the right with pad characters. Text_str is the string to be centered, size is the size of the new string. The default pad character is a space.

```
10  PRINT CPAD$('123', 9, '0')
20  END

RNH
000123000
```

CSC(num_expr)

CSC(x) is the cosecant of X. It is shorthand for 1/SIN(x).

DATE

Returns today's date in YYDDD format.

DATE$[(int_expr1, [int_expr2])]

The DATE$ function returns the date in image format. Int_expr1 is a given Julian day number, the default is today's date. Int_expr2 indicates the desired output format for the date. The Julian day—the number of days since January 1, 1600.

Table A–2 DATE$ Function - Integer Values

Value (int_expr2)	Output Date Format
0	YYYYMMDD format
1	MMDDYYYY format
2	DDMMYYYY format
3	dd-Mon-yyyy format
4	Month dd, yyyy format

```
PRINT DATE$(123456) gives you

    19380104
    |   | |
   year mo day

PRINT DATE$ gives today's date

    19910417
    |   | |
   year mo day

PRINT DATE$(123456,1) gives you

    01041938
    | |   |
   mo day year

PRINT DATE$(123456,2) gives you

    04011938
    | |   |
  day mo  year

PRINT DATE$(123456,3) gives you

    04-Jan-1938
```

```
PRINT DATE$(123456,4) gives you
    January 4, 1938
```

To print today's date in a suitable format for a letter heading, you can use:

```
PRINT DATE$(DAYS(DATE$),4) gives you
    April 17, 1995
```

DAYS(str_expr [, int_num])

Given a date in CCYYMMDD or YYMMDD format, returns the number of days since January 1, 1600 (this date is day 1). This number is called the Julian day.

```
10  PRINT DAYS('19930403')
20  PRINT DAYS('930403')
30  END

RNH
 143634
 143634
```

Int_num indicates the desired input format for the date. The default input format is zero. If the century is not included, it assumes 1900 as the century.

Table A–3 DAYS Function - Integer Values

Value (int_num)	Input Date Format
0	CCYYMMDD or YYMMDD
1	MMDDCCYY or MMDDYY
2	DDMMCCYY or DDMMYY
3	DD-Mon-CCYY or DD-Mon-YY
4	Month DD, CCYY

```
10  PRINT DAYS('19930403',0)
    PRINT DAYS('04031993',1)
    PRINT DAYS('03041993',2)
    PRINT DAYS('03-Apr-1993',3)
    PRINT DAYS('April 3, 1993',4)
20  END

RNH
 143634
 143634
 143634
 143634
 143634
```

DAY$[(int_expr)]

Given an integer expression specifying the number of days since January 1, 1600, DAY$ returns the day of the week. If no integer expression is given, DAY$ returns the day of the week for today's date. The day is returned as a string expression (Friday, Saturday, etc.).

DECODE(str_expr, int_expr)

Given the string representation of a number and the base that the value is in (int_expr), INTOUCH returns the value in base 10. The number is returned as a real number.

DEG(num_expr)

Given an angle in radians, returns the number of degrees.

DIV0(num_expr1, num_expr2)

The DIV0 function divides num_expr1 by num_expr2. If num_expr2 (divisor) is 0, 0 is returned.

```
10  PRINT DIV0(0.8, 0.000004)
    PRINT DIV0(0.8, 0.0)
    PRINT DIV0(6, 3)
    PRINT DIV0(6, 0)
20  END

RNH
 200000
 0
 2
 0
```

DTYPE(expr)

The DTYPE function returns as an integer value, the data type of an expression or dynamic variable: 1 = string, 2 = integer, 3 = real.

```
10  DECLARE DYNAMIC x
20  x = 45.6
30  PRINT DTYPE(x)
40  END

RNH
3
```

EDIT$(str_expr,int_expr)

EDIT$ performs one or more editing operations, depending on the value of the integer expression, on the supplied string argument. The integer expression is one of the integers below, or a sum of integers below for the desired edit functions:

Table A–4 EDIT$ Function - Operation Values

Value	Edit Operation
1	Trim parity bits.
2	Discard all spaces and tabs.
4	Discard characters: CR, LF, FF, ESC, RUBOUT and NULL.
8	Discard leading spaces and tabs.
16	Reduce spaces and tabs to One space.
32	Convert lower case to upper case.
64	Convert "[" to "(" and "]" to ")".
128	Discard trailing spaces and tabs.
256	Do not alter characters inside quotes.

ELEMENTS(str_expr1 [, str_expr2])

Returns the number of elements in a string expression which contains a list of elements. Str_expr1 is the string containing the list of elements. Str_expr2 is the separator. A comma is the default separator.

- Leading and trailing spaces are removed from the text.
- Quoted data is treated as one element.

ELEMENT$(str_expr1, num_expr [, str_expr2])

ELEMENT$ returns the element from str_expr1 which is specified by the num_expr. Str_expr1 contains a set of elements with separators between them. The default separator is a comma:

```
10  LET a$ = ELEMENT$('ADD,DEL,EXIT',2)
20  PRINT a$
30  END

RNH
DEL
```

You can specify a separator other than the comma with str_expr2.

```
10  LET sentence$ = 'This is a test.'
20  LET a$ = ELEMENT$(sentence$,2,' ')
30  PRINT a$

RNH
is
```

More than one separator in a row returns a null for the corresponding element.

```
10  LET sentence$ = 'This,, is, a, test'
    PRINT ELEMENT$(sentence$, 2)
20  END

RNH
```

ENCODE$(num_expr, num_int)

Returns a string containing a number converted to the base specified. Num_expr is the value to convert. Num_int is the base to convert. For instance, '2' indicates binary, etc.

EPS(num_expr)

INTOUCH carries 15 digits of precision. The EPS(number) returns the smallest value, which when added to "number" yields the next number.

EVAL(str_expr)

This function evaluates the expression described in str_expr, and returns the result. If the variable being assigned the result is dynamic, the function puts the result in whatever data type the expression result was in. If the variable is NOT dynamic, and the result is the wrong data type, a "Data type mismatch" error is returned.

```
10  LINE INPUT 'Enter an expression': a$
    PRINT 'The result is '; EVAL(a$)
20  END

RNH
Enter an expression?  5 + 4
The result is  9
```

EXLABEL$

EXLABEL$ returns the label and line number executing when the last exception occurred, e.g., DO_INPUT.4

EXLINE

EXLINE returns the line number executing when the last exception occurred.

EXP(num_expr)

EXP function returns the value of the mathematical constant, "e", raised to a specified power.

EXTEXT$(int_expr)

EXTEXT$ returns explanatory text associated with a specified exception number.

EXTYPE

EXTYPE returns the number of the last exception that occurred. It is returned as an integer.

FALSE

Returns the constant 0. It is returned as an integer.

FILESPEC$(str_expr1 [, str_expr2 [, str_expr3]])

FILESPEC$ parses a file specification and returns either a full file specification or specific file specification fields.

Str_expr1 is the file specification to be parsed. If no file specification is given, the device and directory you are currently running from are returned.

Str_expr2 is a list of field names, separated by commas, which are to be returned. The field names are:

DEVICE	device name
DIRECTORY	directory name
NAME	file name
TYPE	type or extension name
VERSION	file version number
LOCATION	device and directory names
ALL or ""	full file specification

Str_expr3 is the default file specification. This parameter is optional.

FILESPEC$ can be used in various formats.

```
10  PRINT FILESPEC$('x.y', 'ALL')
    PRINT FILESPEC$('', 'ALL')
20  END
```

```
RNH
USER:[FRED]X.Y;
USER:[FRED].;
```

```
10  x$ = 'TTI_RUN:CLIENT'
    PRINT FILESPEC$(x$, 'ALL', '.dat')
20  END
```

```
RNH
STORAGE:[INTOUCH]CLIENT.DAT;
```

```
10  PRINT FILESPEC$('tti_run:client', 'ALL', '.dat')
    PRINT FILESPEC$('tti_run:client', 'LOCATION')
    PRINT FILESPEC$('tti_run:client', 'LOCATION, NAME')
    PRINT FILESPEC$('tti_run:client.dat;1')
20  END
```

```
RNH
STORAGE:[INTOUCH]CLIENT.DAT;
STORAGE:[INTOUCH]
STORAGE:[INTOUCH]CLIENT
STORAGE:[INTOUCH]CLIENT.DAT;1
```

FINDFILE$(str_expr [, int_expr])

Given a file name to find, returns the complete file specification of the first file found that matches the name given. If no file is found, the function returns a null string.

FINDFILE$ calls can be nested if the inner call has only one argument (i.e., the file specification, but no index number).

str_expr	The name of the file to search for. This can be just part of the full file specification.
int_expr	Which file specification to return if multiple files are found. This parameter is optional. The default is to return the first file found.
Result	The complete file specification of the file found.

```
10  PRINT FINDFILE$('tti_run:*.int')
```

```
RNH
DUA0:[INTOUCH]ALIENS.INT;5
```

```
10  DO
      LINE INPUT 'File.Spec': spec$
      IF  _EXIT  THEN  EXIT DO
      FOR i = 1 TO 9999
        file$ = FINDFILE$(spec$, i)
        IF  file$ = ''  THEN  EXIT FOR
        PRINT file$
      NEXT i
    LOOP
20  END

RNH
File specification? TTI_RUN:CLIENT.*
DUA0:[INTOUCH]CLIENT.DAT;7
DUA0:[INTOUCH]CLIENT.DEF;3
DUA0:[INTOUCH]CLIENT.INT;1
DUA0:[INTOUCH]CLIENT.SAV;1
DUA0:[INTOUCH]CLIENT.STR;16
```

FORMAT$(expr, str_expr)

Given an expression and a format, returns the result of the expression in the format indicated.

The '@' format character causes the character not to be translated by the formatter. The '<' and '>'are treated like an '@' character. You can justify a character string, but avoid zero suppression and zero insertion.

The FORMAT$ function takes an expression of any data type for the data to format (the first argument), including string expressions.

```
10  z$ = FORMAT$('1234567', '###~-####')
    PRINT 'Phone number: '; z$
20  END

RNH
Phone number: 123-4567
```

The FORMAT$ function returns all asterisks "*" in the case of overflow.

```
10  z$ = FORMAT$(12.23,'#.##')
    PRINT z$
20  END

RNH
****
```

FORMAT$( ) returns the same string data as given by the following:

```
PRINT USING str_expr: expr
```

The FORMAT$ function supports the DATE format and date arguments. Given a date in YYMMDD or CCYYMMDD format, FORMAT$ returns the date in the date format requested.

```
FORMAT$(z$, '{DATE [argument]}?')
```

The **?** can be replaced with a mask. If no date argument is provided, the default is MDCY.

```
10  z1$ = FORMAT$('950401', '{DATE MDCY}?')
    z2$ = FORMAT$('950401', '{DATE MDCY}##/##/####')
    z3$ = FORMAT$('19950401', '{DATE MDCY}?')
    z4$ = FORMAT$('19950401', '{DATE MDCY}##/##/####')
20  PRINT z1$, z2$
    PRINT z3$, z4$
30  END

RNH
04011995            04/01/1995
04011995            04/01/1995
```

Table A–5 FORMAT$ Function - Date Arguments

DATE Argument	YYMMDD Input	Result	CCYYMMDD Input	Result
none	950207	02071995	19950207	02071995
YMD	950207	950207	19950207	950207
CYMD	950207	19950207	19950207	19950207
MDY	950207	022195	19950207	022195
MDCY	950207	02071995	19950207	02071995
DMY	950207	070295	19950207	070295
DMCY	950207	07021995	19950207	07021995
DMONY	950207	07-Feb-95	19950207	07-Feb-95
DMONCY	950207	07-Feb-1995	19950207	07-Feb-1995
MONTHDY	950207	February 7, 95	19950207	February 7, 95
MONTHDCY	950207	February 7, 1995	19950207	February 7, 1995

The FORMAT$ function supports character rotation. The ROTATE option rotates the last nn characters of a string to the first position in the string.

```
FORMAT$(z$, '{ROTATE n}?')
```

The **?** can be replaced with a mask.

```
10  z1$ = FORMAT$('5552527800', '{ROTATE 3}?')
    z2$ = FORMAT$('5552527800', '{ROTATE 3}###~ ###~-####')
    z3$ = FORMAT$('TuneTommy', '{ROTATE 5}?')
    z4$ = FORMAT$('TuneTommy', '{ROTATE 5}#####~ ####')
20  PRINT z1$, z2$
    PRINT z3$, z4$
30  END

RNH
8005552527            800 555-2527
TommyTune             Tommy Tune
```

The FORMAT$ function supports the TIME format. Given a military standard time in HHMM, HH:MM, HHMMSS or HH:MM:SS format, FORMAT$ returns the time as HH:MM AM/PM or HH:MM:SS AM/PM.

```
FORMAT$(z$, '{TIME}?')
```

```
10  t1$ = FORMAT$('1022', '{TIME}?')
    t2$ = FORMAT$('19:45:36', '{TIME}?')
20  PRINT t1$
    PRINT t2$
30  END
```

```
RNH
10:22 AM
07:45:36 PM
```

The FORMAT$ function supports the ZIPCODE format. Given a 5, 6 or 9 digit zipcode, FORMAT$ returns a formatted zipcode.

```
FORMAT$(z$, '{ZIPCODE}?')
```

```
10  z1$ = FORMAT$('92126', '{ZIPCODE}?')
    z2$ = FORMAT$('K8A3P9', '{ZIPCODE}?')
    z3$ = FORMAT$('931327845', '{ZIPCODE}?')
20  PRINT '5 character zipcode : '; z1$
    PRINT '6 character zipcode : '; z2$
    PRINT '9 character zipcode : '; z3$
30  END
```

```
RNH
5 character zipcode : 92126
6 character zipcode : K8A 3P9
9 character zipcode : 93132-7845
```

FP(num_expr)

Given a number, returns the fractional part of the number.

FULLTIME$[(float_expr, [int_var])]

Given the number of seconds since the INTOUCH base date, this function returns the date and time in one of the formats given below.

Where float_expr is the number of seconds since the INTOUCH base date. The default is the current date and time. January 1, 1600 00:00:00 is considered the second 0.

Table A–6 FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
0	CCYYMDD HHMMSS
1	MMDDCCYY HHMMSS
2	DDMMCCYY HHMMSS

Table A–6 (Cont.) FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
3	DD-Mon-CCYY HH:MM:SS
4	Month DD, CCYY HH:MM:SS

```
10  PRINT FULLTIME$
20  sec = SECONDS('19910621 115042')
30  PRINT FULLTIME$(sec, 0)
40  PRINT FULLTIME$(sec, 1)
50  PRINT FULLTIME$(sec, 2)
60  PRINT FULLTIME$(sec, 3)
70  PRINT FULLTIME$(sec, 4)
80  END

RNH
19910621 123756
19910621 115042
06211991 115042
21061991 115042
21-Jun-1991 11:50:42
June 21, 1991 11:50:42
```

HASH$(str_expr1 [, str_expr2 [, int_expr]])

This function changes the plain text in str_expr1 into a hashed eight-byte string value. It can be used to develop one-way hashed passwords. The optional text in str_expr2 and optional int_expr can be used to further make the hashed value unique.

```
10  password$ = HASH$('TRUTH')
    INPUT 'Password': pwd$
    IF  HASH$(pwd$) = password$  THEN
      PRINT 'That was the correct password.'
    ELSE
      PRINT 'That was not the correct password.'
    END IF
20  END

RNH
Password?  MONEY
That was not the correct password.
```

INT(num_expr)

INT returns the whole portion of a real number as a real number.

INTEGER(num_expr)

INTEGER changes any numeric expression into an integer value and assigns the integer value to the variable specified.

IP(num_expr)

IP truncates the value of a real number at the decimal point and returns the integer portion.

ITEM(str_expr1, str_expr2)

The ITEM function returns the number of the first item that matches the whole or partial item name given. A negative number is returned if more than one item matches.

```
10  z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'MOD')
    PRINT z
20  END

RNH
4


10  z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'EX')
    PRINT z
20  END

RNH
-2
```

LBOUND(array_name [,int_expr])

Given an array and a dimension number, returns the low bound for that dimension. The default dimension is 1.

LCASE$(str_expr)

LCASE returns a string expression with all letters in lower case.

LEFT[$](str_expr, int_expr)

LEFT$ returns the leftmost **nn** characters from a string. Int_expr is the last character position to be included in the resulting string.

LEN(str_expr)

LEN returns the length of a string. It returns an integer.

LOG(num_expr)

LOG returns the natural logarithm of a specified number.

LOG2(num_expr)

LOG2 returns a number's base 2 logarithm.

LOG10(num_expr)

LOG10 returns a number's common logarithm.

LPAD$(text_str, size [, pad_str])

LPAD$ pads a string on the left with pad characters. The default pad character is a space.

```
10  PRINT LPAD$('123', 6, '0')
20  END
RNH

000123
```

LTRIM$(str_expr)

Removes all leading spaces (those on the left side of the string).

MATCH(str_expr1, str_expr2)

Str_expr1 contains a list of elements separated by commas. Str_expr2 contains a string. MATCH compares str_expr2 with each of the elements in str_expr1 and gives the number of the element that matches.

Quoted data is treated as one element; the quotes are not removed.

```
10  INPUT 'Which procedure (ADD,DEL,QUIT)': pro$
20  LET x = MATCH('ADD,DEL,QUIT', pro$)
30  ON x GOSUB 50, 70, 90
40  GOTO 10
50  PRINT 'Adding...'
60  RETURN
70  PRINT 'Deleting...'
80  RETURN
90  END
```

MAX(num_expr, num_expr)

MAX(x,y) returns the larger of the two values x and y.

MAXLEN(str_var)

Returns the maximum number of characters that a string variable can contain. Since all string variables are variable length, with a maximum of 65535, this function always returns 65535.

MAXNUM

Returns the largest number available in this implementation of INTOUCH.

MAXSIZE(array_name)

Returns the total number of elements that can be contained in an array.

MID[$](str_expr, int_expr1 [,int_expr2])

MID$ or MID returns a substring from the middle characters of a specified string, leaving the string unchanged. Int_expr1 is the starting position of the substring, and int_expr2 is the length of the substring. MID$(str_expr,int_expr1) will return the rest of the string.

```
10  a$ = 'beginmiddleend'
    middle$ = MID$(a$, 6, 6)
    end$ = MID$(a$, 6)
    PRINT middle$, end$
20  END

RNH
middle     middleend
```

MIN(num_expr1, num_expr2)

MIN(x,y) returns the lesser of the values x and y.

MOD(num_expr1, num_expr2)

Gives the remainder of one number divided by another.

ORD(str_expr)

Given the ASCII name of a character, returns the location of that character in the ASCII character table. It returns an integer number.

ORDNAME$(int_expr)

Given the location of a character in the ASCII character table, returns the name of that character.

PARSE$(str_expr)

PARSE$ splits a string into tokens and returns each token separated by a space. Letters are upper-cased, except within quotes. Tail comments are ignored. Embedded "$" characters are allowed. The maximum line length is 1024 characters.

```
a$ = 'company$ = abc$ +"and sons" !rnn'
PRINT PARSE$(a$)

RNH
COMPANY$ = ABC$ + "and sons"
```

PATTERN(str_expr1, str_expr2)

Matches any characters in text (str_expr1) with the pattern (str_expr2). Str_expr1 is the text to search and str_expr2 is the pattern being searched for. Returns the location of the first character in the text that contains the pattern. If the characters cannot be found, returns zero.

PATTERN Options and Examples

Pattern options can be mixed and matched with unlimited complexity.

?

Matches any character in the text.

```
10  IF  PATTERN('The quick brown fox', &
       'f?x') > 0  THEN
       PRINT 'Found'
    END IF
20  END

RNH
Found
```

*****

Matches zero or more of a character that precedes the asterisk.

```
10  IF  PATTERN('aaa   03/26/92', 'a* *03/26/92') > 0  THEN
      PRINT 'The date is found'
    END IF
20  END

RNH
The date is found
```

{}

Used to define a group of characters. The characters in the enclosed group can be ranges such as {a-z} or individual characters such as {AQX}.

```
10  text$ = 'A1N5V7N0'
    rule_str$ = '{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}'
    IF  PATTERN(text$, rule_str$) > 0  THEN
      PRINT 'Driver's licence is correct'
    END IF
20  END

RNH
Driver's licence is correct
```

{^}

Looks for characters that are NOT {^A-Z}. The result of line 30 below shows the difference between using '?' and {^}.

```
10  PRINT PATTERN('Mary J. Smith','{^Mar}')
20  PRINT PATTERN('Mary J. Smith','J. {^S}')
30  PRINT PATTERN('Mary J. Smith','J. S?')
40  END

RNH
4
0
6
```

~

The '~' (quote) character looks for a pattern of text that IS an * (stands for itself).

```
10  text$ = '$4,670.00'
    IF  PATTERN(text$, '$~4, 670.00') > 0  THEN
      PRINT 'Your text is correct'
    END IF
20  END

RNH
Your text is correct
```

{<cc|ccc|c>}

Looks for text in str_expr1 that matches the enclosed groups.

```
10  text$ = 'The area code is 619'
    IF  PATTERN(text$, 'is {<619|714|916>}') > 0  THEN
      PRINT 'Your area code is on the list'
    END IF
20  END

RNH
Your area code is on the list
```

PI

Returns the value 3.141592653589793.

PIECES(str_expr1 [,str_expr2])

and

PIECE$(str_expr1,num_expr[,str_expr2])

PIECE$ returns an element from str_expr1 specified by num_expr. Str_expr1 contains a list of elements. The separator can be indicated by str_expr2. The default separator is a carriage-return line-feed pair.

These two functions are similar to the ELEMENTS( ) and ELEMENT$( ) functions except that:

- no leading or trailing spaces are removed from the text.
- quoted data is not skipped over.
- the default separator is the two-character sequence. CHR$(13)+CHR$(10), the CR/LF sequence used by WRAP$( ).

```
10 CLEAR
    MESSAGE 'Press GOLD/F when done'
    LINE INPUT AREA 5, 10, 8, 60: text$
    PRINT AT 10, 1: 'Rewrapped text'
    wt$ = WRAP$(text$, 1, 30)
    PRINT 'Number of lines: '; PIECES(wt$)
    PRINT wt$
    PRINT
    PRINT 'First line was: '; PIECE$(wt$, 1)
20  END
```

POS(str_expr1, str_expr2[, int_expr])

POS searches for a substring within a string. It returns the substring's starting character position. Str-exp1 is the string to be searched, str-exp2 is the substring to locate, and int-exp is the OPTIONAL character position at which to begin the search. (The default is the start of the string.)

PRETTY$(str_expr)

PRETTY$ converts text so that the text displays on any terminal. Named control characters show up with their names. Other control characters show up as {X} where "X" is the letter to press or as {XX} where "XX" is the hexadecimal value of the character.

```
10  a$ = 'Hello' + CHR$(5) + CHR$(161) + CHR$(7)
20  PRINT PRETTY$(a$)
30  END

RNH
Hello{^E}{A1}{bel}
```

QUOTE$(str_expr)

The QUOTE$ function encloses a string expression in double quotes. If the string expression is already enclosed in double quotes, QUOTE$ leaves it alone. If the string expression is already wrapped in single quotes, QUOTE$ replaces them with double quotes. Elements double quoted within the string expression are given another pair of double quotes (see following example). Elements single quoted within the string expression are ignored.

```
10  DO
      CLEAR
      PRINT AT 1,1:
      MESSAGE 'Enter a line of text to be quoted'
      PRINT 'Text:'
      INPUT '', LENGTH 30: line$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  line$ = ''  THEN REPEAT DO
20    PRINT
      PRINT 'Quoted text using the QUOTE$ function...'
      PRINT quote$(line$)
      DELAY
    LOOP
30  END

RNH
Text:
? The little boy cried "wolf!"

Quoted text using the QUOTE$ function...
"The little boy cried ""wolf!"""
```

RAD(num_expr)

Given a measurement in degrees, returns the number of radians.

REAL(num_expr)

REAL changes any numeric expression into a real or floating-point value and assigns the real value to the variable specified.

```
10  INPUT 'Your age': age%
    LET decimal_age = REAL(age%)
    PRINT 'Your number is'; decimal_age
20  END
```

REMAINDER(num_expr1, num_expr2)

REMAINDER(x,y) returns the remainder when X is divided by Y. It differs subtly from MOD. MOD(-4,3) = 2 while REMAINDER(-4,3) = -1.

REPEAT$(str_expr, int_expr)

Creates a string composed of the specified string repeated the specified number of times.

REPLACE$(str_expr1, str_expr2 [,str_sep1][,str_sep2])

Searches for a list of patterns in the str_expr1 and replaces them with the output string from str_expr2. Returns the replaced string expression.

Str_expr1 is a list of patterns to search for.

Str_expr2 is the replacement list.

Str_sep1 is the optional separator for replacement items. The default is a comma.

Str_sep2 is the optional separator between the input and output text in items. Default is =.

```
10  text$ = '01-Mar-1981'
    PRINT REPLACE$(text$, 'r=y 8=9' , ' ')
20  END

RNH
01-May-1991
```

RIGHT[$](str_expr, int_expr)

RIGHT$ returns the rightmost characters from a string. The int_exp is the character position of the last character to be included in the substring COUNTING FROM THE RIGHT.

```
10  ans$ = RIGHT$('Daniel', 2)
    PRINT 'rightmost char.= ', ans$
20  END

RNH
rightmost char.= el
```

RND

or

RND(num_expr)

RND returns a random number greater than or equal to zero and less than one. If a numeric expression (num_expr) is given, RND returns a whole number between one and the numeric expression.

ROUND(num_expr [, int_expr])

ROUND rounds a num_expr to the specified number of decimal places (int_expr). The default int_expr is 0.

RPAD$(text_str, size[,pad_str])

RPAD$ pads a string on the right with pad characters. The default pad character is a space.

```
10  PRINT RPAD$('123', 6, '0')
20  END

RNH
123000
```

RTRIM$(str_expr)

Returns a string without any trailing spaces (those on the right side).

SCAN(str_expr1, str_expr2 [, int_expr])

Scans str_expr1 for the characters in str_expr2 and returns the position at which str_expr2 begins. Int_expr specifies a character position at which the search is to begin.

The characters in str_expr2 need not appear contiguously in str_expr1.

```
10  LET a$ = 'Cameron Whitehorn'
    LET b$ = 'amr Wtor'
    LET position = SCAN(a$, b$)
    PRINT position
20  END

RNH
2
```

SEC(num_expr)

Returns a secant of a given angle (1/COS(num_expr)). Num_expr is a passed angle.

SECONDS(str_expr)

Given a full-time string in CCYYMMDD HHMMSS, YYMMDD HHMMSS, HHMMSS or HHMM format, returns the number of seconds since the INTOUCH base date (January 1, 1600 00:00:00).

The number of seconds is returned as a floating point number.

```
10  z  = SECONDS('19931201 103050')
    z1 = SECONDS('931201 103050')
    z2 = SECONDS('103050')
    z3 = SECONDS('1030')
20  PRINT 'Seconds CYMDHMS ='; z
    PRINT 'Seconds  YMDHMS ='; z1
    PRINT 'Seconds     HMS ='; z2
    PRINT 'Seconds     HM  ='; z3
30  END

RNH
Seconds CYMDHMS = 12430837850
Seconds  YMDHMS = 12430837850
Seconds     HMS = 37850
Seconds     HM  = 37800
```

SEG$(str_expr, int_expr1, int_expr2)

Extracts the substring given a first and last character position.

SGN(num_expr)

Returns the sign of a number. It returns a +1 if the expression is positive, a -1 if the expression is negative, and 0 if the expression is zero.

SIN(num_expr)

SIN returns the sine of an angle specified in radians.

SINH(num_expr)

SINH(X) returns the hyperbolic sine X.

SIZE(array_name [,int_expr])

Returns the number of elements in one dimension of an array.

array-name	Array to examine
int-expr	Dimension to get the size of. The default dimension is 1.

SKIP(str_expr1 [, str_expr2] [, int_expr])

SKIP returns the position of the character following the last skipped character.

Str_expr1 is the text string to be searched.

Str_expr2 contains the list of characters which are to be skipped. If only one argument is given, SKIP will skip over spaces, tabs and nulls.

Int_expr contains the search start position. This parameter is optional.

```
10  a$ = '31415 hello'
    z = SKIP(a$, '1234567890 ')
20  PRINT mid(a$, z)
30  END

RNH
Hello
```

SORT$(str_expr1 [,str_expr2])

Sorts the elements from a str_expr1 in ASCII value order. Returns a list of the sorted elements.

Str_expr1 contains the list of elements to be sorted.

Str_expr2 is an optional separator. Default is a comma.

```
10  a$ = 'code area is'
    a_sort$ = SORT$(a$, ' ')
    PRINT a_sort$
20  END

RNH
area code is
```

SPACE$(num_expr)

Returns the number of spaces indicated by num_expr.

SQR(num_expr)

SQR returns the square root of a number.

STR$(num_expr)

STR$ changes a number to a numeric string. The string that is created does not have any extra leading or trailing spaces.

SYSTEXT$

or

SYSTEXT$(int_expr)

SYSTEXT$ returns the text associated with the operating system status specified. If no int_expr is supplied, INTOUCH returns the text for the last system status.

TAB(int_expr)

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to the right to a specified column.

TAN(num_expr)

TAN returns the tangent of an angle that is specified in radians.

TANH(num_expr)

Returns the hyperbolic tangent of the numeric expression given.

TIME(int_expr)

The value returned by the TIME function depends on the value of int_expr.

If int_expr = 0, TIME returns the number of seconds since midnight.

If int_expr = 1, TIME returns the CPU time of the process in tenths of a second.

If int_expr = 2, TIME returns connect time of the current process in minutes.

TIME$

or

TIME$(num_expr)

If num_expr is NOT specified, TIME$ returns the current time of day in HH:MM:SS format.

Num_expr is the number of seconds since midnight. The result is returned in HH:MM format.

```
10  PRINT TIME$(1800)
    PRINT TIME$(54178)
20 END

RNH
00:30
15:02
```

TIME(5)

Returns the number of seconds since INTOUCH was invoked. This function can be used to time events to the nearest 100th/sec.

TRIM$(str_expr)

Returns the string specified stripped of any leading or trailing spaces and tabs.

```
10  LET a$ = '    HELLO    '
20  PRINT '*'; a$; '*'
30  LET stripped$ = TRIM$(a$)
40  PRINT '*'; stripped$; '*'

RNH
*    HELLO    *
*HELLO*
```

TRUE

Returns the constant 1. It is returned as an integer.

TRUNCATE(num_expr, int_expr)

Truncates a real number to a given number of decimal places.

UBOUND(array_name [, int_expr])

Given an array and a dimension number, returns the upper bound for that dimension. It returns an integer value. The default dimension is 1.

UCASE$(str_expr)

UCASE returns a string expression with all letters in upper case.

UNQUOTE$(str_expr)

The UNQUOTE$ function removes one set of quotes from a quoted string expression. If the string expression is not quoted, UNQUOTE$ leaves the string alone. UNQUOTE$ does not affect internally quoted elements.

```
10  DO
      CLEAR
      PRINT AT 1,1:
      MESSAGE 'Enter a line of text to be unquoted'
      PRINT 'Text:'
      INPUT '', LENGTH 50: line$
      IF  _BACK  OR  _EXIT  THEN  EXIT DO
      IF  line$ = ''  THEN  REPEAT DO
20    PRINT
      PRINT 'Quotes removed using the UNQUOTE$ function...'
      PRINT UNQUOTE$(line$)
      DELAY
    LOOP
100 END

RNH
Text:
? "I will not take these 'things' for granted"
```

```
Quotes removed using the UNQUOTE$ function...
I will not take these 'things' for granted
```

VAL(num_str)

VAL returns the floating-point value of a numeric string.

VALID(text_str, rule_str)

VALID is used to validate user responses.

Text_str is the text to be validated.

Rule_str is the list of validation rules.

Multiple validation rules are separated by a semi-colon. If given characters are NOT between quotes, they are upper-cased.

VALID returns an error if there is an invalid validation rule.

```
'Illegal validation rule' (-4021)
```

VALID returns TRUE or FALSE according to the following validation rules:

- **ALLOW text1, text2, text3 [to text4]**

  ALLOW compares the text_str with each element on the rule_str list (defaults to CHARACTER) and returns TRUE if found.

  For numeric comparison, "NUMBER" or "INTEGER" *must* be used, otherwise, VALID( ) will do a string-compare validation...left justified.

```
10  IF  VALID('ann', 'ALLOW ann, dan, tom')  THEN  PRINT 'true'
20  IF  NOT VALID('10',  'NUMBER; ALLOW 1 to 6; MAXLENGTH 2') &
          THEN PRINT 'false'
30  IF  VALID('12', 'NUMBER; ALLOW 1 TO 24,99') THEN  PRINT 'true'
40  END

RNH
true
false
true
```

- **DISALLOW text1, text2, text3 [to text4]**

  Returns TRUE if text_str cannot be found on the rule_str list.

```
10  IF VALID('10', 'DISALLOW 01,03,05; MINLENGTH 2') &
      THEN PRINT 'true'
20  END

RNH
true
```

- **MINLENGTH nn**

  Minimum of nn characters long.

- **MAXLENGTH nn**

  Maximum of nn characters long.

- **LENGTH nn**

  Exactly nn characters long.

- **CHARACTERS ccc**

  Returns TRUE if the given characters (letters or numbers) are on the rule_str list.

- **NOCHARACTERS ccc**

  Returns TRUE if the given characters are not on the rule_str list.

- **LETTERS**

  Returns TRUE if given text_str consists only of A - Z, a - z or space.

- **LCASE**

  Returns TRUE if given text_str consists of lower-case letters.

- **UCASE**

  Returns TRUE if given text_str consists of upper-case characters.

- **DIGITS**

  Returns TRUE if text_str consists of the numbers 0 - 9.

- **DECIMALS nn**

  Specifies the maximum number of decimal places.

- **NUMBER**

  Indicates that text_str is numeric (validates that number).

- **INTEGER**

  Returns TRUE if given integer number is up to 32 bits long (i.e. 2147483647).

- **INTEGER WORD**

  Returns TRUE if given integer number is up to 16 bits long (i.e. 32767).

- **DATE**

  Validates a given date in: YYMMDD or CCYYMMDD format.

DATE	YMD	YYMMDD format
DATE	DMY	DDMMYY format
DATE	MDY	MMDDYY format
DATE	MDCY	MMDDCCYY format

Note

For all dates, use 'LENGTH nn' to control whether a 6- or 8-character date is required.

- **DATE DMONY**

  Checks if text_str is a legal date in DD-Mon-YY format and returns TRUE or FALSE.

```
10  PRINT VALID('01-May-93', 'DATE DMONY')
20  END

RNH
1
```

- **DATE DMONCY**

  Checks if text_str is a legal date in DD-Mon-CCYY format and returns TRUE or FALSE.

```
10  PRINT VALID('01-May-1993', 'DATE DMONCY')
20  END

RNH
1
```

- **FULLTIME**

  Checks if text_str is a legal date and time in CCYYMMDD HHMMSS or YYMMDD HHMMSS format and returns TRUE or FALSE.

```
10  PRINT VALID('19931231 010101', 'FULLTIME')
    PRINT VALID('931231 010101', 'FULLTIME')
20  END

RNH
1
1
```

- **CODE**

  Checks for VALID INTOUCH code syntax.

```
10  a$ = 'PRINT 'Hello, ' name$'
    IF  NOT VALID(a$, 'CODE')  THEN  PRINT 'false'
30  END
```

- **PATTERN**

  Matches text_str with PATTERN's rule_str. For details see the PATTERN$ function.

- **REQUIRED**

  Returns TRUE if text_str is not null or space(s).

- **YES/NO**

  Returns TRUE if text_str consists only of the following: YES, NO, Y or N.

- **VRULES**

  Checks if a text_str is a legal set of validation rules, and returns TRUE or FALSE.

```
10  PRINT VALID('integer', 'vrules')
20  END

RNH
1
```

- **PRINTMASK**

  Checks the format of text_str and returns TRUE if it is legal or FALSE otherwise.

```
10  text_str$ = '##.##'
    PRINT VALID(text_str$, 'PRINTMASK')
20  END

RNH
1
```

- **EXPRESSION**

  Validates a text_str$ and returns TRUE if it is a legal INTOUCH expression or FALSE otherwise.

```
10  text_str$ = 'total = a% + 30'
    PRINT VALID(text_str$, 'EXPRESSION')
    PRINT VALID('##~-###', 'EXPRESSION')
20  END

RNH
1
0
```

- **MENU**

Validates a text_str$ and returns TRUE if it is a legal menu description for an INPUT MENU statement, or FALSE otherwise.

```
10  text_str$ = '%multi,a,b,c'
    PRINT VALID(text_str$, 'MENU')
20  END

RNH
1
```

- **FILTER**

  Filters the text_str before the validation. The following filters are allowed:

  — **LTRIM$** - Remove leading spaces.

  — **RTRIM$** - Remove trailing spaces.

  — **TRIM$** - Remove both leading and trailing spaces.

  — **UCASE$** - Convert to upper-case characters.

  — **LCASE$** - Convert to lower-case characters.

  — **REPLACE** - If text_str contains a few 'old_t1' characters, all of them will be replaced by 'new_t1'.

  ```
  old_t1=new_t1, old_t2=new_t2, ....
  ```

  — **REMOVE** - Removes any characters from the text_str.

  — **CHANGE** - Changes specified characters. For details see the CHANGE$ function.

  ```
  10  IF  VALID('abcd10', &
            'FILTER REMOVE "10"; LETTERS'  THEN  PRINT 'true'
      IF  VALID("ab1cd1", "FILTER REPLACE '1'='e';LETTERS") &
            THEN  PRINT 'true'
      IF  VALID(" 1234  ","FILTER TRIM; NUMBER") &
            THEN  PRINT 'true'
  30  END

  RNH
  true
  true
  true
  ```

  — **RESTORE** - Restores data to pre-filtered state for further validations. Can be used for multiple validation rules.

```
10  IF  VALID("123", &
            "FILTER REPLACE '1'='a';RESTORE;NUMBER") &
    THEN  PRINT "true"
20  END

RNH
true
```

WRAP$(str_expr, int_expr1, int_expr2)

Returns a word-wrapped text string, given left and right margins. Each line of the string is separated with a CR/LF.

Where string_expr = text string to wrap, int_expr1 = left margin, int_expr2 = right margin.

XLATE$(str_expr1, str_expr2)

The XLATE$ function translates one string to another by referencing a table you supply. For example, the XLATE$ function can translate from EBCDIC to ASCII. The first str_expr is the string to be translated. The second str_expr is the translation table.

B Creating Libraries for Use with INTOUCH

To create libraries for use with INTOUCH, you will need to follow the steps described below:

1 Write the source code using an editor of your choice. The source code can be in FORTRAN or any other VMS provided language that supports the common calling interface.

2 Compile the source code.

3 Write an options file. Use an editor to create an options file. Give the options file a file extension of ".OPT".

For VAX systems:

Enter the option UNIVERSAL to specify the modules (subprograms and functions) contained in your routine.

```
UNIVERSAL = module [, module...]
```

If your routine contains more than one module, separate multiple module names with commas. The routine described above has only one module:

```
UNIVERSAL = MULT
```

For Alpha systems:

Use the following format to specify the modules (subprograms and functions) contained in your routine.

```
SYMBOL_VECTOR = (module1=PROCEDURE)
SYMBOL_VECTOR = (module2=PROCEDURE)
  . . .
```

If your routine contains more than one module, use a separate line for each module in the .OPT file. The routine described above has only one module:

```
SYMBOL_VECTOR = (MULT=PROCEDURE)
```

4 Link the library.

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

For the routine above, if your options file is called ROUTINE.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]ROUTINE.EXE"
    CALL mult(8%, 9%, answer%)
    PRINT 'The answer is '; answer
90  END
```

Index

A

ABORT command • 5–4
ABS function • A–3
Access rules • 1–15
ACOS function • A–3
ADABAS • 1–2
ADABAS data files • 1–4
ADD STRUCTURE • 3–5
Alpha AXP systems • 5–9
Alternate key • 1–24
Ambiguous constant • 15–8
Ampersand in line continuation • 15–3
ANGLE function • A–3
AREA option in INPUT • 8–6
Arithmetic operators • 15–16
Array • 3–3
  description • 15–12
  elements • 15–12
  subscript • 15–12
Array elements • 1–13
Array low bound • 10–3
Arrays • 10–1
ASCII function • A–4
ASIN function • A–4
ASK #chnl_num options
  CURRENT • 13–5
  MARGIN • 13–5
  NAME • 13–5
  ZONEWIDTH • 13–5
ASK #chnl_num statement • 13–5
ASK ERRORS statement • 11–3
ASK KEYSTROKES statement • 11–5
ASK MARGIN statement • 11–5
ASK MESSAGELINE statement • 11–6
ASK PAGESIZE statement • 11–7
ASK RESPONSES statement • 11–8
ASK SEED statement • 11–9
ASK statements • 11–1
ASK STRUCTURE #string_expr • 3–26
ASK STRUCTURE: ACCESS • 3–26
ASK STRUCTURE: CAPABILITY • 3–21
  INDEXED • 3–22
  INPUT • 3–22
  null string • 3–22
  OUTPUT • 3–22
ASK STRUCTURE: CAPABILITY (cont'd)
  RELATIVE • 3–22
ASK STRUCTURE: CURRENT • 3–16
ASK STRUCTURE: ENGINE • 3–22
ASK STRUCTURE: EXTRACTED • 3–23
ASK STRUCTURE: FIELDS • 3–17
ASK STRUCTURE: ID • 3–23
ASK STRUCTURE: KEYS • 3–18
ASK STRUCTURE: POINTER • 3–24
ASK STRUCTURE: RECORD • 3–25
ASK STRUCTURE: RECORDSIZE • 3–25
ASK STRUCTURE: TIMEOUT • 3–26
ASK STRUCTURE FIELD: item • 3–18
  ACCESS • 3–19
  APPLICATION • 3–19
  ATTRIBUTES • 3–19
  CHANGEABLE • 3–19
  CLASSIFICATION • 3–20
  DATATYPE • 3–20
  DESCRIPTION • 3–20
  HEADING • 3–20
  HELP • 3–20
  KEYED • 3–20
  LENGTH • 3–20
  NAME • 3–20
  NULL • 3–20
  NUMBER • 3–20
  OPTIMIZED • 3–21
  POSITION • 3–21
  PRINTMASK • 3–21
  PROMPT • 3–21
  SCREENMASK • 3–21
  VRULES • 3–21
ASK SYSTEM, LOGICAL: VALUE statement • 11–12
ASK SYSTEM, PASS: SUCCESS statement • 11–14
ASK SYSTEM, SYMBOL statement • 11–15
ASK SYSTEM: COMMENT statement • 11–11
ASK SYSTEM: DIRECTORY statement • 11–11
ASK SYSTEM: MODE statement • 11–13
ASK SYSTEM: PARAMETER statement • 11–13
ASK SYSTEM: PROCESS statement • 11–14
ASK SYSTEM: RIGHTS statement • 11–15
ASK SYSTEM: USER statement • 11–16
ASK SYSTEM options • 11–11
ASK WINDOW: COLUMN statement • 11–18
ASK WINDOW: CURRENT statement • 11–18

ASK WINDOW: DATA statement • 11–19
ASK WINDOW: KEYMAP statement • 11–20
ASK WINDOW: ROW statement • 11–22
ASK WINDOW: TYPEAHEAD statement • 11–23
ASK WINDOW AREA statement • 11–17
ASK WINDOW options • 11–16
ASK ZONEWIDTH statement • 11–10
Assigning variables • 7–1
ATN function • A–4
AT row, column option in INPUT • 8–7
ATTRIBUTES option in INPUT • 8–8

B

_BACK in SET BACK ON/OFF • 11–2
_BACK internal variable • 8–3
Backslash • 5–2
_BACK system function • A–1
Basic elements of a program • 15–1
Bit manipulation • 15–19
BOOLEAN variables • 15–9
Branching • 12–9
Branching and code blocks • 12–9
BREAK • 4–6
Brief field display • 1–17
Built-in functions • A–1

C

Callable routines in libraries • 16–1
Called routine • 16–2
Call routines written in other languages • 16–1
CALL SETUP • 1–26
CALL SETUP gets into SETUP • 1–1
CALL statement • 16–1, 16–2, B–2
CANCEL ADD • 3–5
CANCEL EXTRACT • 3–11
CAUSE EXCEPTION statement • 14–1
CEIL function • A–4
CHAIN statement • 12–16
CHANGE$ function • A–4
_CHANNEL • 13–5
Channel number • 13–1
_CHANNEL system function • A–1
CHARSET$ function • A–4
CHR$ function • A–5
CLASSIFICATION file • 1–8
Classification in SETUP • 1–8
CLEAR AREA attributes • 9–9
CLEAR AREA BOX attributes • 9–10
CLEAR AREA BOX statement • 9–9
CLEAR AREA statement • 9–8
CLEAR statement • 9–7
CLOSE #chnl_num statement • 13–2
CLOSE ALL • 3–4
CLOSE STRUCTURE • 3–4
COBOL
  checking define results • 1–26
  define example • 1–25
  defining record fields • 1–25
COBOL_DEF.INT • 1–25
Commands
  ABORT • 5–4
  COMPILE • 5–8
  EDIT • 5–6
  EXIT • 5–1
  FRAME ON/OFF • 5–7
  HELP • 5–1
  IDE • 5–8
  INTOUCH/COMPILE • 5–9
  INTOUCH/DEVELOP • 5–10
  INTOUCH/IMAGE • 5–9
  LIST • 5–5
  LIST ERROR • 5–6
  LNH • 5–5
  LNH ERROR • 5–6
  NEW • 5–2
  OLD • 5–4
  REFRESH • 5–7
  RENUMBER • 5–7
  REPLACE • 5–3
  RNH • 5–2
  RUN • 5–2
  SAVE • 5–3
  UNSAVE • 5–5
! comment • 15–6
Comments in programs • 15–5
COMPARE function • A–6
COMPILE command • 5–8
Compile directives • 6–1
Compiled program image • 5–9
Compile-time errors • 5–6
Compound expressions • 15–16
  concatenated string expressions • 15–17
  conditional expressions • 15–17
  numeric expressions • 15–16

Concatenated string expressions • 15–17
Conditional expressions • 15–17
Conditional numeric expressions • 15–17
Conditionals - IF/END IF, CASE • 12–7
Constants
  description • 15–9
  E notation • 15–10
  numeric - integer • 15–9
  numeric - real • 15–9
  string • 15–10
CONTINUE in exception handler • 14–5
Continuing program lines • 15–3
Conventions
  other • xviii
  program examples • xvii
  user • xvi
Conventions used in manual • xvi
CONVERT$ function • A–5
CONVERT function • A–5
COS function • A–6
COSH function • A–6
COT function • A–6
CPAD$ function • A–6
Create data file procedure in SETUP • 1–22
CSC function • A–7
CSET statement • 7–3

D

Database engine • 1–2, 1–22
Database engines - supported • 1–2
Data dictionary • 1–4, 1–25
Dataset • 1–3
DATA statement • 7–4
Data type in SETUP • 1–11
Data types
  integer number • 15–7
  real number • 15–7, 15–8
  string • 15–7, 15–8
DATE$ function • A–7
DATE directive used with PRINT USING • 9–5
DATE format with FORMAT$ • A–14
DATE function • A–7
DAY$ function • A–9
DAYS function • A–8
dBASE III • 1–2
DBMS • 1–2
DBMS data files • 1–3
DBMS structure handling • 3–15, 3–16, 3–32, 3–33
DCL symbol for running INTOUCH images • 5–10
DEBUG OFF • 4–2
DEBUG ON • 4–1
%DEBUG program directive • 6–2
Debug statements
  BREAK • 4–6
  DEBUG OFF • 4–2
  DEBUG ON • 4–1
  GO • 4–8
  LIST STATS • 4–5
  STATS OFF • 4–5
  STATS ON • 4–4
  STEP • 4–7
  TRACE OFF • 4–3
  TRACE ON • 4–2
DEBUG system description • 4–1
_DEBUG system function • A–1
Decimal point • 1–13
DECLARE PREFIX statement • 7–7
DECLARE statement • 7–5
DECLARE STRING/INTEGER/REAL/DYNAMIC
    statement • 7–5
DECLARE STRUCTURE statement • 7–6
DECODE function • A–9
Default file extension - .INT • 15–5
Default menu item • 8–8
DEFAULT option in INPUT • 8–8
Define fields procedure in SETUP • 1–6
Defining arrays • 10–1
Defining COBOL record fields • 1–25
Definition file • 1–4
DEG function • A–9
DEGREE • A–9
DELAY statement • 6–6
Delete access level • 1–21
DELETE STRUCTURE • 3–7
Deleting a file • 13–7
Description in SETUP • 1–11
Developing INTOUCH programs • 5–10
Dimension arrays • 10–2
DIM statement • 10–1, 15–12
Directives in programs
  %DEBUG • 6–2
  %INCLUDE • 6–1
  %INCLUDE CONDITIONAL • 6–1
  %MESSAGE • 6–2
  %MESSAGE ERROR • 6–2

Directives used with PRINT USING • 9–4
DISPATCH statement • 12–14
DIV0 function • A–9
DO...END DO loop • 12–6
DO loop • 12–4
DO loop description • 12–1
DTYPE function • 7–6, A–9

E

EDIT$ function • A–10
EDIT command • 5–6
Editing an input response • 8–4
Edit string • A–10
ELAPSED option in INPUT • 8–9
ELEMENT$ function • A–10
ELEMENTS function • A–10
ENCODE$ function • A–11
END ADD used with ADD STRUCTURE • 3–5
END DO statement • 12–6
END statement • 6–4
END WHEN with WHEN EXCEPTION • 14–2
EOF option with LINE INPUT • 13–4
EPS function • A–11
ERASE option in INPUT • 8–9
Error counter • 11–3
_ERROR in SET ERROR ON/OFF • 11–3
EVAL function • A–11
Evaluation order • 15–20
Exception
  line • A–12
Exception handling • 14–1
Exclamation point with comments • 15–6
EXCLUDE in EXTRACT STRUCTURE • 3–9
Execute library routines • 16–2
EXECUTE statement • 12–13
EXIT ADD • 3–6
EXIT command • 5–1
EXIT DO statement • 12–5
EXIT EXTRACT • 3–12
EXIT FOR in FOR EACH struc loop • 3–11
EXIT FOR statement • 12–2
EXIT HANDLER statement • 14–3
_EXIT in SET EXIT ON/OFF • 11–4
_EXIT internal variable • 8–3
EXIT ROUTINE statement • 12–15
_EXIT system function • A–1
EXLABEL$ function • A–11
EXLINE function • A–12
EXP function • A–12
Expressions - description • 15–9
Expressions evaluated • 3–2
EXTEXT$ function • A–12
Extract a range of keys • 3–13
_EXTRACTED system function • A–1
EXTRACT STRUCTURE • 3–8
EXTRACT STRUCTURE, SET, FIELD: KEY • 3–16
EXTRACT STRUCTURE, SET, USING • 3–15
EXTRACT STRUCTURE: APPEND • 3–14
EXTRACT STRUCTURE: KEY • 3–12
EXTRACT STRUCTURE: PARTIAL KEY • 3–13
EXTYPE function • A–12

F

FALSE function • A–12
FASTFILE • 1–2, 1–3
Field access in structure • 3–1
Field attributes • 1–13
Field class • 1–8
Field displays • 1–17
Field expressions • 3–2
Field Name in SETUP • 1–7
Field names and numbers • 3–2
Fields
  multiple occurrences • 3–3
  variable length • 1–12
Fields in structures • 15–14
Fields list display • 1–18
Fields with multiple occurrences • 15–15
File extension - .INT • 15–5
File organization • 1–23
Files • 13–1
FILESPEC$ function • A–12
File specification • 2–1
FINDFILE$ function • A–13
FOR...NEXT loop • 12–1
FOR EACH struc/NEXT struc loop • 3–10
FOR loop description • 12–1
FORMAT$ function • A–14
  DATE format options • A–14
  ROTATE option • A–15
  TIME format • A–16
  ZIPCODE format • A–16

FP function • A–16
FRAME ON/OFF command • 5–7
Full field display • 1–18
FULLTIME$ function • A–16
Functions - Non-System
ABS • A–3
ACOS • A–3
ANGLE • A–3
ASCII • A–4
ASIN • A–4
ATN • A–4
CEIL • A–4
CHANGE$ • A–4
CHARSET$ • A–4
CHR$ • A–5
COMPARE • A–6
CONVERT • A–5
CONVERT$ • A–5
COS • A–6
COSH • A–6
COT • A–6
CPAD$ • A–6
CSC • A–7
DATE • A–7
DATE$ • A–7
DAY$ • A–9
DAYS • A–8
DECODE • A–9
DEG • A–9
DIV0 • A–9
DTYPE • A–9
EDIT$ • A–10
ELEMENTS • A–10
ELEMENTS$ • A–10
ENCODE$ • A–11
EPS • A–11
EVAL • A–11
EXLABEL$ • A–11
EXLINE • A–12
EXP • A–12
EXTEXT$ • A–12
EXTYPE • A–12
FALSE • A–12
FILESPEC$ • A–12
FINDFILE$ • A–13
FORMAT$ • A–14
FP • A–16
FULLTIME$ • A–16
HASH$ • A–17
INT • A–17
INTEGER • A–17
Functions - Non-System (cont'd)
IP • A–18
ITEM • A–18
LBOUND • A–18
LCASE$ • A–18
LEFT[$] • A–18
LEN • A–18
LOG • A–18
LOG10 • A–19
LOG2 • A–19
LPAD$ • A–19
LTRIM$ • A–19
MATCH • A–19
MAX • A–19
MAXLEN • A–20
MAXNUM • A–20
MAXSIZE • A–20
MID[$] • A–20
MIN • A–20
MOD • A–20
ORD • A–20
ORDNAME$ • A–21
PARSE$ • A–21
PATTERN • A–21
PI • A–23
PIECE$ • A–23
PIECES • A–23
POS • A–23
PRETTY$ • A–23
QUOTE$ • A–24
RAD • A–24
REAL • A–24
REMAINDER • A–25
REPEAT$ • A–25
REPLACE$ • A–25
RIGHT[$] • A–25
RND • A–25
ROUND • A–26
RPAD$ • A–26
RTRIM$ • A–26
SCAN • A–26
SEC • A–26
SECONDS • A–26
SEG$ • A–27
SGN • A–27
SIN • A–27
SINH • A–27
SIZE • A–27
SKIP • A–27
SORT$ • A–28
SPACE$ • A–28

Functions - Non-System (cont'd)
SQR • A–28
STR$ • A–28
SYSTEXT$ • A–28
TAB • A–29
TAN • A–29
TANH • A–29
TIME • A–29
TIME$ • A–29
TIME(5) • A–29
trim$ • A–30
TRUE • A–30
TRUNCATE • A–30
UBOUND • A–30
UCASE$ • A–30
UNQUOTE$ • A–30
VAL • A–31
VALID • A–31
WRAP$ • A–36
XLATE$ • A–36
Functions - System
_BACK • A–1
_CHANNEL • A–1
_DEBUG • A–1
_EXIT • A–1
_EXTRACTED • A–1
_HELP • A–2
_INTEGER • A–2
_REAL • A–2
_REPLY • A–2
_STATUS • A–2
_STRING • A–2
_TERMINATOR • A–3

G

4GL - fourth-generation database language • xv
GO • 4–8
GOSUB...RETURN statement • 12–11
GOSUB Construct • 12–11
GOTO statement • 12–9
Guide to the INTOUCH Language • xvi

H

HALT statement • 6–5
HANDLER...END HANDLER with WHEN EXCEPTION • 14–3
HASH$ function • A–17
HELP command • 5–1
Help facility • 5–1
_HELP in SET HELP ON/OFF • 11–4
Help in SETUP • 1–2
_HELP internal variable • 8–3
_HELP system function • A–2
Help text • 1–15
HYPHEN directive used with PRINT USING • 9–4

I

IDE command • 5–8
IF/THEN...END IF statement • 12–7
IF construct • 12–8
IMPLICIT prefix for variables • 7–7
%INCLUDE CONDITIONAL program directive • 6–1
INCLUDE in EXTRACT STRUCTURE • 3–9
%INCLUDE program directive • 6–1
Indexed file • 1–23
INGRES • 1–2
INGRES data files • 1–3
INPUT #chnl_num statement • 13–3
Input from a terminal • 8–1
INPUT options
AREA • 8–6
AT row, column • 8–7
ATTRIBUTES • 8–8
DEFAULT • 8–8
ELAPSED • 8–9
ERASE • 8–9
LENGTH • 8–9
MENU • 8–10
PROMPT • 8–13
SCREEN • 8–13
TIMEOUT • 8–16
VALID • 8–16
INPUT options list • 8–6
Input response options
backslash • 8–3
Ctrl/Z • 8–3
data entry • 8–3
EXIT • 8–3
HELP or Help key • 8–3
UP arrow • 8–3

INPUT statement • 8–1, 8–2
INPUT statement types • 8–2
INPUT string variable • 8–4
INPUT styles • 8–2
  formatted data entry • 8–2
  free format multiline • 8–2
  pop-up menus • 8–3
  simple input • 8–2
Inputting multiple variables • 8–5
INTEGER function • A–17
Integer number • 15–7
_INTEGER system function • A–2
INT function • A–17
INTOUCH/COMPILE command • 5–9
INTOUCH/DEVELOP command • 5–10
INTOUCH/IMAGE command • 5–9
INTOUCH and VMS • 2–1
INTOUCH commands • 5–2
INTOUCH language • xv
INTOUCH program description • 15–1
INTOUCH program elements • 15–1
INTOUCH reserved words • 15–2
.INT program extension • 15–5
.INT_IMG compiled image extension • 5–9
.INT_IMG_AXP compiled image extension • 5–9
IP function • A–18
ISHOW.COM • 11–11
ITEM function • A–18
ITERATE DO statement • 12–6
ITERATE FOR statement • 12–3
ITERATE in FOR EACH struc loop • 3–11

K

KEY/TO option in EXTRACT STRUCTURE • 3–9
KEY INPUT statement • 8–1, 8–2, 8–16
KEY option in EXTRACT STRUCTURE • 3–9, 3–12
Key range extraction • 3–13
Keystroke concepts • 11–22
KILL statement • 13–7

L

Labels in programs • 15–4
LBOUND function • A–18
LCASE$ function • A–18
LCASE directive used with PRINT USING • 9–4
LEFT[$] function • A–18
LEN function • A–18
LENGTH option in INPUT • 8–9
LET statement • 7–1
LET used with ADD STRUCTURE • 3–5
Libraries • 16–1, B–1
  Creating for INTOUCH • 16–1
  User-defined library • 16–1
  VMS supplied library • 16–1
Library routine • 16–2
  optional arguments • 16–2
  passing mechanisms • 16–3
  using _INTEGER function with CALL • 16–3
  using _REAL function with CALL • 16–3
LIBRARY statement • 16–1, B–2
Line in a program • 15–2
LINE INPUT #chnl_num statement • 13–4
LINE INPUT statement • 8–1, 8–2, 8–5
Line length of file • 13–1
Line number range • 15–3
Line terminators • 15–2
  Enter • 15–2
  Return • 15–2
LIST command • 5–5
LIST ERROR command • 5–6
LIST STATS • 4–5
LNH command • 5–5
LNH ERROR command • 5–6
LOCK STRUCTURE • 3–7
LOG10 function • A–19
LOG2 function • A–19
LOG function • A–18
Logical operators • 15–18
Loops • 12–1
LPAD$ function • A–19
LSET statement • 7–2, 7–3
LTRIM$ function • A–19

M

MATCH function • A–19
MAX function • A–19
MAXLEN function • A–20
MAXNUM function • A–20
MAXSIZE function • A–20
Memory image of program • 5–9

Menu directives • 8–11
%AT row, column • 8–11
%AUTODISPLAY • 8–11
%AUTOVBAR • 8–11
%BAR • 8–11
%COLUMNS • 8–11
%HEADING • 8–11
%ITEMS • 8–11
%LOCKSTEP • 8–11
%MAXWIDTH number • 8–11
%MENUBAR • 8–12
%MULTI • 8–12
%REPLACE • 8–12
%SIZE number • 8–12
%SPLIT • 8–12
%TITLE • 8–12
%VBAR • 8–12
%WIDTH number • 8–12
Menu elements • 8–10
MENU option in INPUT • 8–10
Menu response options • 8–12
MESSAGE DELAY: statement • 9–6
MESSAGE ERROR: statement • 9–6
MESSAGE ERROR incrementing error counter • 11–3
%MESSAGE ERROR program directive • 6–2
%MESSAGE program directive • 6–2
MESSAGE statement • 9–6
Message trapping • 9–10
MID[$] function • A–20
MIN function • A–20
MOD function • A–20
Modify structure general information • 1–19
Modify structure procedure in SETUP • 1–19
Modify structure security information • 1–20
Multiple occurrence fields • 3–3, 15–15
Multiple statements on a single line • 15–3

N

NEW command • 5–2
Null string • 15–10
Numeric expressions • 15–16

O

OLD command • 5–4
ON...GOSUB...RETURN statement • 12–12
ON...GOTO...ELSE statement • 12–10
ON GOSUB Construct • 12–12
OPEN #chnl_num options
ACCESS INPUT • 13–1
ACCESS OUTIN • 13–2
ACCESS OUTPUT • 13–1
OPTIMIZE OFF • 13–2
UNFORMATTED • 13–2
UNIQUE • 13–2
OPEN #chnl_num statement • 13–1
OPEN STRUCTURE • 3–3
OPTION BASE statement • 7–9, 10–3
OPTION COMMUNICATE ON/OFF • 9–10
OPTION REQUIRE DECLARE statement • 7–9
ORACLE • 1–2
ORACLE data files • 1–4
Order of evaluation • 15–20
ORD function • A–20
ORDNAME$ function • A–21

P

Page size of screen • 11–7
PARSE$ function • A–21
Pass command to system • 12–17
PASS NOWAIT:|NORETURN: statement • 12–17
PATTERN function • A–21
PIECE$ function • A–23
PIECES function • A–23
PI function • A–23
POISE • 1–2
POS function • A–23
PREFIXED variable • 7–7
PRETTY$ function • A–23
Primary key • 1–24
PRINT #chnl_num statement • 13–3
PRINT format characters • 9–3
Printing data to a terminal • 9–1
Printing structure fields • 3–2
Print mask • 1–14
PRINT options • 9–2
AT row, column • 9–2
BLINK, UNDERLINE, BOLD, REVERSE • 9–2
ERASE • 9–3
PRINT USING • 9–3
TAB • 9–2
WIDE • 9–3

PRINT statement • 1–14, 9–1
PRINT USING directives • 9–4
DATE • 9–5
HYPHEN • 9–4
LCASE • 9–4
ROTATE • 9–5
TIME • 9–5
UCASE • 9–4
ZIPCODE • 9–6
Program description • 15–1
Program directives • 6–1
Program elements • 15–1
Program examples • xvii
Program labels • 15–4
Program line • 15–2
Program line numbers • 15–3
PROGRAM statement • 6–3
PROMPT option in INPUT • 8–13
Prompt text • 1–14

Q

QUOTE$ function • A–24

R

RAD function • A–24
RANDOMIZE statement • 6–5
Rdb • 1–2
Rdb data files • 1–3
Read access level • 1–21
READ statement • 7–4
REAL function • A–24
Real number • 15–8
_REAL system function • A–2
Record management system • 1–2
Records in structures • 15–14
Record size • 1–23
REDIM statement • 10–2, 15–12
REEXTRACT STRUCTURE • 3–9
REFRESH command • 5–7
Relational operators • 15–17
Relative record access • 3–31
REMAINDER function • A–25
REM statement • 15–5
RENUMBER command • 5–7
REPEAT$ function • A–25
REPEAT DO statement • 12–6
REPEAT FOR statement • 12–2
REPEAT in FOR EACH struc loop • 3–11
REPEAT ROUTINE statement • 12–16
REPLACE$ function • A–25
REPLACE command • 5–3
_REPLY system function • A–2
Report heading • 1–14
Reserved words • 15–2
RESTORE statement • 7–4
RESUME in exception handler • 14–6
RETRY in exception handler • 14–4
RETURN with GOSUB • 12–11
RIGHT[$] function • A–25
RMS • 1–2
RMS file • 1–22
RND function • A–25
RNH command • 5–2
ROTATE directive used with PRINT USING • 9–5
ROTATE option with FORMAT$ • A–15
ROUND function • A–26
ROUTINE...END ROUTINE statement • 12–15
RPAD$ function • A–26
RSET statement • 7–2, 7–3
RTRIM$ function • A–26
RUN command • 5–2
.RUN extension • 5–8
Running compiled images • 5–9

S

S1032 • 1–2
SAVE command • 5–3
Scale factor • 1–13
SCAN function • A–26
Screen - control appearance • 9–6
SCREEN input format characters • 8–14
SCREEN input format options • 8–15
AJ - auto jump • 8–15
AT row, column • 8–15
BOLD, BLINK, UNDERLINE, REVERSE • 8–15
DIGITS • 8–15
ELAPSED time • 8–15
LCASE • 8–15
NOECHO • 8–15
REQ - required • 8–15
UCASE • 8–15
VALID • 8–15

Index

Screen mask • 1–14
Screen message trapping • 9–10
SCREEN option in INPUT • 8–13
SEC function • A–26
SECONDS function • A–26
SEG$ function • A–27
SELECT CASE...END SELECT statement • 12–8
Sequential file • 1–23
SET #chnl_num options
  CURRENT • 13–6
  MARGIN • 13–6
  ZONEWIDTH • 13–6
SET #chnl_num statement • 13–6
SET AUTOEXIT statement • 11–1
SET BACK ON|OFF statement • 11–2
SET ERROR ON|OFF statement • 11–2
SET EXIT ON|OFF statement • 11–4
SET HELP ON|OFF statement • 11–4
SET MARGIN statement • 11–5
SET MESSAGELINE statement • 11–6
SET PORT ON|OFF statement • 11–7
SET SCROLL statement • 11–9
SET SEED statement • 11–9
SET statements • 11–1
SET STRUCTURE #string_expr • 3–26
SET STRUCTURE, SET, USING • 3–32
SET STRUCTURE, SET, USING, FIELD: KEY • 3–33
SET STRUCTURE, SET, USING: EXPRESSION 'owner' • 3–32
SET STRUCTURE, SET: EXPRESSION 'owner' • 3–33
SET STRUCTURE: CURRENT • 3–28
SET STRUCTURE: EXTRACTED 0 • 3–27
SET STRUCTURE: ID • 3–30
SET STRUCTURE:POINTER • 3–31
SET STRUCTURE: RECORD • 3–31
SET STRUCTURE: TIMEOUT • 3–32
SET STRUCTURE FIELD:KEY • 3–29
SET STRUCTURE FIELD: PARTIAL KEY • 3–29
SET SYSTEM, LOGICAL: VALUE statement • 11–12
SET SYSTEM, SYMBOL statement • 11–15
SET SYSTEM: COMMENT statement • 11–11
SET SYSTEM: DIRECTORY statement • 11–11
SET SYSTEM: PROCESS statement • 11–14
SET SYSTEM options • 11–11
SETUP
  Access rules • 1–15
  array elements • 1–13
  Brief Field Display • 1–17
  Classification • 1–8
  Create • 1–1
SETUP (cont'd)
  Create data file procedure • 1–22
  Database engine • 1–2
  Data dictionary • 1–4
  Dataset • 1–3
  Data Type • 1–11
  Define • 1–1
  Define fields procedure • 1–6
  Description • 1–11
  Ending position • 1–12
  entry • 1–1
  Field length • 1–12
  Field Name • 1–7
  File organization • 1–23
  First position • 1–12
  Full Field Display • 1–18
  Help • 1–2
  Help text • 1–15
  List Fields Display • 1–18
  Modify structure • 1–1
  Modify structure general information • 1–19
  Modify structure procedure • 1–19
  Modify structure security information • 1–20
  occurrences • 1–13
  Primary key • 1–24
  Print mask • 1–14
  Procedures • 1–6
  Prompt text • 1–14
  Record size • 1–23
  Report heading • 1–14
  Screen mask • 1–14
  Semantics • 1–13
  Show • 1–1
  Show procedure • 1–16
  Structure Display • 1–18
  Structure name • 1–2
  Validation rules • 1–15
SETUP DEFINE screen • 1–7, 1–10
SETUP procedures • 1–1
SETUP procedures menu • 1–5
SETUP prompts • 1–2
SETUP routine • 1–26
SETUP Routine • 1–1
SET WINDOW: COLUMN statement • 11–18
SET WINDOW: CURRENT statement • 11–18
SET WINDOW: DATA statement • 11–19
SET WINDOW: KEYMAP statement • 11–20
SET WINDOW: ROW statement • 11–22
SET WINDOW: TYPEAHEAD statement • 11–23
SET WINDOW AREA statement • 11–17

SET WINDOW KEYSTROKE statement • 11–21
SET WINDOW options • 11–16
SET ZONEWIDTH statement • 11–10
SGN function • A–27
Show procedure in SETUP • 1–16
SIN function • A–27
SINH function • A–27
SIZE function • A–27
SKIP function • A–27
SORT$ function • A–28
SORT ASCENDING/DESCENDING • 3–9
SORT in EXTRACT STRUCTURE • 3–9
SPACE$ function • A–28
SQR function • A–28
Statements
ASK #chnl_num • 13–5
ASK ERRORS • 11–3
ASK KEYSTROKES • 11–5
ASK MARGIN • 11–5
ASK MESSAGELINE • 11–6
ASK PAGESIZE • 11–7
ASK RESPONSES • 11–8
ASK SEED • 11–9
ASK SYSTEM, LOGICAL: VALUE • 11–12
ASK SYSTEM, PASS: SUCCESS • 11–14
ASK SYSTEM, SYMBOL: VALUE • 11–15
ASK SYSTEM: COMMENT • 11–11
ASK SYSTEM: DIRECTORY • 11–11
ASK SYSTEM: MODE • 11–13
ASK SYSTEM: PARAMETER • 11–13
ASK SYSTEM: PROCESS • 11–14
ASK SYSTEM: RIGHTS • 11–15
ASK SYSTEM: USER • 11–16
ASK WINDOW: COLUMN • 11–18
ASK WINDOW: CURRENT • 11–18
ASK WINDOW: DATA • 11–19
ASK WINDOW: KEYMAP • 11–20
ASK WINDOW: ROW • 11–22
ASK WINDOW: TYPEAHEAD • 11–23
ASK WINDOW AREA • 11–17
ASK ZONEWIDTH • 11–10
CALL • 16–2
CAUSE EXCEPTION • 14–1
CHAIN • 12–16
CLEAR • 9–7
CLEAR AREA • 9–8
CLEAR AREA BOX • 9–9
CLOSE #chnl_num • 13–2
CONTINUE in exception handler • 14–5
CSET • 7–3
CSET FILL • 7–3
Statements (cont'd)
DATA • 7–4
DECLARE PREFIX • 7–7
DECLARE STRING/INTEGER/REAL/DYNAMIC • 7–5
DECLARE STRUCTURE • 7–6
DELAY • 6–6
DIM • 10–1
DISPATCH • 12–14
END • 6–4
END DO • 12–6
EXECUTE • 12–13
EXIT DO • 12–5
EXIT FOR • 12–2
EXIT HANDLER • 14–3
EXIT ROUTINE • 12–15
FOR...NEXT • 12–1
GOSUB...RETURN • 12–11
GOTO • 12–9
HALT • 6–5
IF/THEN...END IF • 12–7
INPUT • 8–1
INPUT #chnl_num • 13–3
ITERATE DO • 12–6
ITERATE FOR • 12–3
KEY INPUT • 8–1
KILL • 13–7
LET • 7–1
LIBRARY • 16–1
LINE INPUT • 8–1
LINE INPUT #chnl_num • 13–4
LSET • 7–2
LSET FILL • 7–3
MESSAGE • 9–6
MESSAGE DELAY: • 9–6
MESSAGE ERROR: • 9–6
ON...GOSUB...RETURN • 12–12
ON...GOTO...ELSE • 12–10
OPEN #chnl_num • 13–1
OPTION BASE • 7–9, 10–3
OPTION COMMUNICATE ON/OFF • 9–10
OPTION REQUIRE DECLARE • 7–9
PASS NOWAIT:|NORETURN: • 12–17
PRINT • 9–1
PRINT #chnl_num • 13–3
PROGRAM • 6–3
RANDOMIZE • 6–5
READ • 7–4
REDIM • 10–2
REM • 15–5
REPEAT DO • 12–6
REPEAT FOR • 12–2

Statements (cont'd)
REPEAT ROUTINE • 12–16
RESTORE • 7–4
RESUME in exception handler • 14–6
RETRY in exception handler • 14–4
ROUTINE...END ROUTINE • 12–15
RSET • 7–2
RSET FILL • 7–3
SELECT CASE...END SELECT • 12–8
SET #chnl_num • 13–6
SET AUTOEXIT • 11–1
SET BACK ON|OFF • 11–2
SET ERROR ON|OFF • 11–2
SET EXIT ON|OFF • 11–4
SET HELP ON|OFF • 11–4
SET MARGIN • 11–5
SET MESSAGELINE • 11–6
SET PORT ON|OFF • 11–7
SET SCROLL • 11–9
SET SEED • 11–9
SET SYSTEM, LOGICAL: VALUE • 11–12
SET SYSTEM, SYMBOL: VALUE • 11–15
SET SYSTEM: COMMENT • 11–11
SET SYSTEM: DIRECTORY • 11–11
SET SYSTEM: PROCESS • 11–14
SET WINDOW: COLUMN • 11–18
SET WINDOW: CURRENT • 11–18
SET WINDOW: DATA • 11–19
SET WINDOW: KEYMAP • 11–20
SET WINDOW: ROW • 11–22
SET WINDOW: TYPEAHEAD • 11–23
SET WINDOW AREA • 11–17
SET WINDOW KEYSTROKE • 11–21
SET ZONEWIDTH • 11–10
STEP in FOR/NEXT • 12–1
STOP • 6–4
WHEN EXCEPTION IN • 14–1
WHEN EXCEPTION USE • 14–2
STATS OFF • 4–5
STATS ON • 4–4
_STATUS system function • A–2
STEP • 4–7
STEP in FOR/NEXT loop • 12–1
STOP statement • 6–4
STR$ function • A–28
STR file change • 1–20, 1–22
String data • 15–8
String delimiters • 15–10
String expressions - concatenated • 15–17
_STRING system function • 8–8, A–2
Structure • 15–13
contents • 3–1
description • 3–1
field access • 3–1
field expression • 3–2
field names • 15–14
fields • 3–1, 15–14
record manipulation • 3–1
records • 3–1, 15–14
Structure display • 1–18
Structure name • 1–2
Structure references • 15–13
Structure security level • 1–21
Structure statements
ADD STRUCTURE • 3–5
ASK STRUCTURE #string_expr • 3–26
ASK STRUCTURE: ACCESS • 3–26
ASK STRUCTURE: CAPABILITY • 3–21
ASK STRUCTURE: CURRENT • 3–16
ASK STRUCTURE: ENGINE • 3–22
ASK STRUCTURE: EXTRACTED • 3–23
ASK STRUCTURE: FIELDS • 3–17
ASK STRUCTURE: ID • 3–23
ASK STRUCTURE: KEYS • 3–18
ASK STRUCTURE: POINTER • 3–24
ASK STRUCTURE: RECORD • 3–25
ASK STRUCTURE: RECORDSIZE • 3–25
ASK STRUCTURE: TIMEOUT • 3–26
ASK STRUCTURE FIELD: item • 3–18
CANCEL ADD • 3–5
CANCEL EXTRACT • 3–11
CLOSE ALL • 3–4
CLOSE STRUCTURE • 3–4
DELETE STRUCTURE • 3–7
EXIT ADD • 3–6
EXIT EXTRACT • 3–12
EXTRACT STRUCTURE • 3–8, 3–13
EXTRACT STRUCTURE, SET, FIELD: KEY • 3–16
EXTRACT STRUCTURE, SET, USING • 3–15
EXTRACT STRUCTURE: APPEND • 3–14
FOR EACH/NEXT • 3–10
LOCK STRUCTURE • 3–7
OPEN STRUCTURE • 3–3
REEXTRACT STRUCTURE • 3–9
SET STRUCTURE, SET, USING • 3–32
SET STRUCTURE, SET, USING, FIELD: KEY • 3–33
SET STRUCTURE, SET, USING: EXPRESSION 'owner' • 3–32

Structure statements (cont'd)
SET STRUCTURE, SET: EXPRESSION 'owner' • 3–33
SET STRUCTURE: CURRENT • 3–28
SET STRUCTURE: EXTRACTED 0 • 3–27
SET STRUCTURE: ID • 3–30
SET STRUCTURE:POINTER • 3–31
SET STRUCTURE: RECORD • 3–31
SET STRUCTURE: TIMEOUT • 3–32
SET STRUCTURE FIELD:KEY • 3–29
SET STRUCTURE FIELD: PARTIAL KEY • 3–29
UNLOCK STRUCTURE • 3–7
Substrings • 15–13
System functions • A–1
SYSTEXT$ function • A–28

T

TAB function • A–29
TAB in PRINT #chnl_num • 13–3
TAN function • A–29
TANH function • A–29
Terminal message trapping • 9–10
_TERMINATOR function • 8–17
_TERMINATOR system function • A–3
TIME$ function • A–29
TIME(5) function • A–29
TIME directive used with PRINT USING • 9–5
TIME format with FORMAT$ • A–16
TIME function • A–29
TIMEOUT option in INPUT • 8–16
TRACE feature example • 4–3
TRACE OFF • 4–3
TRACE ON • 4–2
Translate • A–36
Trapping messages • 9–10
TRIM$ function • A–30
Trim leading spaces • A–19
TRUE function • A–30
TRUNCATE function • A–30
TTI_PRINTER_TYPE • 11–7
TTI_RUN:COBOL_DEF.INT • 1–25
TTI_RUN:ISHOW.COM • 11–11

U

UBOUND function • A–30
UCASE$ function • A–30
UCASE directive used with PRINT USING • 9–4
UNLOCK STRUCTURE • 3–7
UNQUOTE$ function • A–30
UNSAVE command • 5–5
Update access level • 1–21
USERBASE • 1–2
User conventions • xvi
USE with WHEN EXCEPTION • 14–2
USING in PRINT #chnl_num • 13–3

V

VAL function • A–31
Validation rules • 1–15
ALLOW • A–31
CHARACTERS • A–32
CODE • A–33
DATE • A–33
DATE DMONCY • A–33
DATE DMONY • A–33
DECIMALS • A–32
DIGITS • A–32
DISALLOW • A–31
EXPRESSION • A–34
FILTER • A–35
CHANGE • A–35
LCASE$ • A–35
LTRIM$ • A–35
REMOVE • A–35
REPLACE • A–35
RESTORE • A–35
RTRIM$ • A–35
TRIM$ • A–35
UCASE$ • A–35
FULLTIME • A–33
INTEGER • A–32
INTEGER WORD • A–32
LCASE • A–32
LENGTH • A–32
LETTERS • A–32
MAXLENGTH • A–32
MENU • A–34
MINLENGTH • A–31
NOCHARACTERS • A–32
NUMBER • A–32
PATTERN • A–34
PRINTMASK • A–34

Validation rules (cont'd)
REQUIRED • A–34
UCASE • A–32
VRULES • A–34
YES/NO • A–34
VALID function • 1–15, A–31
VALID option in INPUT • 8–16
Variable length fields • 1–12
Variables - description • 15–11
VMS and INTOUCH • 2–1
VMS languages • 16–1
VMS library • 16–1

W

WHEN EXCEPTION IN statement • 14–1
WHEN EXCEPTION USE statement • 14–2
WRAP$ function • A–36
Wrapped text • 8–6
Write access level • 1–21

X

XLATE$ function • A–36

Z

ZIPCODE directive used with PRINT USING • 9–6
ZIPCODE format with FORMAT$ • A–16
Zonewidth in SET MARGIN • 11–6