VAX® INTOUCH®

TECHNICAL 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 REMOTE TAPE FACILITY

DYNAMIC LOAD BALANCER PLUS FRAGMENTATION CONTROLLER REMOTE DEVICE FACILITY

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

RMS Expert

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

DBMS DCL DECNET RDB RMS VAX VMS

Last Revision date: 14-Feb-1994

Copyright ©1992, 1994 Touch Technologies, Inc.

Contents

PREFACE xvii

CHAPTER 1 SETUP ROUTINE 1–1

1.1 SETUP–OVERVIEW 1–1
1.1.1 Entering SETUP 1–2

1.2 SETUP PROMPTS 1–2

1.3 DEFINE PROCEDURE 1–5

1.4 SECURITY 1–10

1.5 CREATE PROCEDURE 1–11

1.6 SHOW PROCEDURE 1–12

1.7 DEFINITIONS 1–13

CHAPTER 2 INTOUCH RUNNING UNDER VMS 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 EXTRACT STRUCTURE 3–5
3.2.4 REEXTRACT STRUCTURE 3–6
3.2.5 FOR EACH 3–7
3.2.6 CANCEL EXTRACT 3–7
3.2.7 EXIT EXTRACT 3–8
3.2.8 EXTRACT STRUCTURE: PARTIAL KEY 3–8
3.2.9 EXTRACT STRUCTURE: APPEND 3–9
3.2.10 EXTRACT STRUCTURE, SET, USING 3–10
3.2.11 EXTRACT STRUCTURE, SET, FIELD: KEY 3–11
3.2.12 DELETE STRUCTURE 3–11
3.2.13 ADD STRUCTURE 3–12
3.2.14 CANCEL ADD 3–12
3.2.15 EXIT ADD 3–13
3.2.16 LOCK | UNLOCK STRUCTURE 3–14

3.3 SET STRUCTURE 3–15
3.3.1 SET STRUCTURE : EXTRACTED 0 3–15
3.3.2 SET STRUCTURE, SET, USING 3–15
3.3.3 SET STRUCTURE, SET, USING: EXPRESSION 'owner' 3–16
3.3.4 SET STRUCTURE, SET: EXPRESSION 'owner' 3–16
3.3.5 SET STRUCTURE, SET, USING, FIELD: KEY 3–17
3.3.6 SET STRUCTURE: CURRENT 3–17
3.3.7 SET STRUCTURE FIELD:KEY 3–18
3.3.8 SET STRUCTURE FIELD: PARTIAL KEY 3–18
3.3.9 SET STRUCTURE: ID 3–19
3.3.10 SET STRUCTURE:POINTER 3–20
3.3.11 SET STRUCTURE: RECORD 3–20
3.3.12 SET STRUCTURE: TIMEOUT 3–21

3.4 ASK STRUCTURE 3–21
3.4.1 ASK STRUCTURE: CURRENT 3–21
3.4.2 ASK STRUCTURE: FIELDS 3–22
3.4.3 ASK STRUCTURE: KEYS 3–22
3.4.4 ASK STRUCTURE FIELD: item 3–22
3.4.5 ASK STRUCTURE: CAPABILITY 3–25
3.4.6 ASK STRUCTURE: ENGINE 3–26
3.4.7 ASK STRUCTURE: EXTRACTED 3–27
3.4.8 ASK STRUCTURE: ID 3–27
3.4.9 ASK STRUCTURE: POINTER 3–28
3.4.10 ASK STRUCTURE: RECORD 3–28

3.4.11 ASK STRUCTURE: RECORDSIZE 3–29
3.4.12 ASK STRUCTURE: ACCESS 3–30
3.4.13 ASK STRUCTURE: TIMEOUT 3–30

3.5 ASK | SET STRUCTURE #STRING_EXPR 3–30

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

4.3 STATISTICS FEATURE 4–4
4.3.1 STATS ON 4–4
4.3.2 STATS OFF 4–4
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 4–6
4.4.3 GO 4–7

CHAPTER 5 INTOUCH COMMANDS 5–1

5.1 THE EXIT AND HELP COMMANDS 5–1

5.2 RUN 5–2

5.3 NEW 5–2

5.4 SAVE 5–3

Contents

5.5 REPLACE 5–3

5.6 ABORT 5–4

5.7 OLD 5–4

5.8 UNSAVE 5–5

5.9 LIST 5–5

5.10 LIST ERRORS 5–6

5.11 EDIT 5–6

5.12 RENUMBER 5–7

5.13 FRAME ON |OFF 5–7

5.14 REFRESH 5–8

5.15 IDE 5–8

5.16 COMPILE 5–9
5.16.1 Running Compiled Images 5–10

CHAPTER 6 DIRECTIVES 6–1

6.1 %INCLUDE 6–1
6.1.1 %INCLUDE CONDITIONAL 6–1

6.2 %DEBUG 6–2

6.3 %MESSAGE 6–2

6.3.1 %MESSAGE ERROR 6–3

CHAPTER 7 ASSIGNING VARIABLES 7–1

7.1 LET 7–1

7.2 LSET 7–2

7.3 RSET 7–2

7.4 CSET 7–3

7.5 LSET, RSET, CSET FILL 7–3

7.6 DATA/READ 7–4

7.7 RESTORE 7–4

7.8 DECLARE STRING | INTEGER | REAL 7–5

7.9 DECLARE DYNAMIC 7–5

7.10 DECLARE STRUCTURE 7–6

7.11 DECLARE PREFIX 7–7

7.12 OPTION REQUIRE DECLARE 7–8

CHAPTER 8 INPUTTING DATA FROM A TERMINAL 8–1

8.1 INPUT 8–1
8.1.1 Types of INPUT Statements 8–2
8.1.2 INPUT Styles 8–2

8.2 INPUT OPTIONS 8–3

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

8.4 INPUTTING STRINGS 8–5

8.5 INPUTTING MULTIPLE VARIABLES 8–5
8.5.1 Simple Input Statement 8–5
8.5.2 LINE INPUT Statement 8–6

8.6 "PROMPT TEXT" 8–7

8.7 PROMPT OPTION 8–7

8.8 AT OPTION 8–7

8.9 LENGTH OPTION 8–7

8.10 DEFAULT OPTION 8–7

8.11 VALID OPTION 8–8

8.12 ERASE OPTION 8–8

8.13 TIMEOUT OPTION 8–8

8.14 ELAPSED OPTION 8–8

8.15 ATTRIBUTES OPTION 8–8

8.16 AREA OPTION 8–9

8.17 SCREEN OPTION 8–10

8.17.1 Screen Format Characters 8–10
8.17.2 Screen Format Options 8–11

8.18 MENU OPTION 8–11
8.18.1 Menu DIRECTIVES: 8–12

8.19 KEY INPUT 8–14

CHAPTER 9 PRINTING DATA TO A TERMINAL 9–1

9.1 PRINT 9–1
9.1.1 PRINT Options 9–1
9.1.2 ? as a Substitute for PRINT Statement 9–4
9.1.3 Print Format Characters 9–4

9.2 USING THE INTOUCH SCREEN 9–5
9.2.1 MESSAGE [ERROR: | DELAY:] 9–5

9.3 CLEAR 9–6
9.3.1 AREA Option 9–6
9.3.2 BOX Option 9–8
9.3.3 OPTION COMMUNICATE ON|OFF 9–8

CHAPTER 10 DEFINING ARRAYS 10–1

10.1 DIM 10–1

10.2 REDIM 10–1

10.3 OPTION BASE 10–2

CHAPTER 11 SET AND ASK STATEMENTS 11–1

11.1 SET AUTOEXIT 11–1

Contents

11.2 SET BACK ON | OFF 11–2

11.3 SET ERROR ON | OFF 11–2

11.4 ASK ERRORS 11–3

11.5 SET EXIT ON | OFF 11–3

11.6 SET HELP ON | OFF 11–4

11.7 ASK KEYSTROKES 11–4

11.8 ASK | SET MARGIN 11–5

11.9 ASK | SET MESSAGELINE 11–5

11.10 SET PORT ON | OFF 11–6
11.10.1 Additional Information on Setting the Port Type 11–6

11.11 ASK RESPONSES 11–7

11.12 SET SCROLL 11–7

11.13 ASK | SET SEED 11–8

11.14 ASK | SET SYSTEM 11–8
11.14.1 ASK | SET SYSTEM: COMMENT 11–9
11.14.2 ASK | SET SYSTEM: DIRECTORY 11–9
11.14.3 ASK | SET SYSTEM, LOGICAL: VALUE 11–10
11.14.4 ASK SYSTEM: MODE 11–11
11.14.5 ASK SYSTEM: PARAMETER 11–11
11.14.6 ASK SYSTEM, PASS: SUCCESS 11–12
11.14.7 ASK | SET SYSTEM: PROCESS 11–12
11.14.8 ASK SYSTEM: RIGHTS 11–13
11.14.9 ASK | SET SYSTEM, SYMBOL: VALUE 11–13
11.14.10 ASK SYSTEM: USER 11–14

11.15 ASK | SET WINDOW 11–14
11.15.1 ASK | SET WINDOW AREA 11–14
11.15.2 ASK WINDOW: COLUMN 11–15
11.15.3 ASK | SET WINDOW: CURRENT 11–16
11.15.4 ASK | SET WINDOW: DATA 11–17
11.15.5 ASK | SET WINDOW: KEYMAP 11–18
11.15.6 SET WINDOW KEYSTROKE 11–19
11.15.7 ASK | SET WINDOW: ROW 11–20
11.15.8 ASK | SET WINDOW: TYPEAHEAD 11–21

11.16 ASK | SET ZONEWIDTH 11–22

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

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

12.2 CONDITIONALS 12–5
12.2.1 IF 12–5
12.2.2 SELECT CASE 12–6

12.3 BRANCHING 12–7
12.3.1 GOTO 12–7
12.3.2 ON GOTO 12–8
12.3.3 GOSUB 12–8
12.3.4 ON..GOSUB 12–9

12.4 EXECUTE 12–10

12.5 DISPATCH 12–11

12.6 ROUTINE/END ROUTINE 12–12
12.6.1 EXIT ROUTINE 12–12
12.6.2 REPEAT ROUTINE 12–13
12.6.3 Branching and Code Blocks 12–13

12.7 CHAINING PROGRAMS 12–13
12.7.1 CHAIN 12–13

12.8 PASS COMMANDS TO THE OPERATING SYSTEM 12–14
12.8.1 PASS [NOWAIT: | NORETURN:] 12–14

CHAPTER 13 FILES 13–1

13.1 OPEN #CHNL_NUM: 13–1

13.2 INPUT 13–2

13.3 LINE INPUT 13–3

13.4 ASK #CHNL_NUM: 13–3

13.5 SET #CHNL_NUM: 13–4

13.6 CLOSE 13–5

13.7 PRINT 13–6

13.8 DELETING FILES 13–6
13.8.1 KILL 13–6

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 THE EXCEPTION HANDLING STATEMENTS 14–3
14.4.1 EXIT HANDLER 14–3
14.4.2 RETRY 14–4
14.4.3 CONTINUE 14–4
14.4.4 RESUME 14–5

14.5 CALLING ROUTINES WRITTEN IN OTHER LANGUAGES 14–6
14.5.1 LIBRARY 14–6
14.5.2 CALL 14–6

CHAPTER 15 DATA IN INTOUCH 15–1

15.1 THE PROGRAM, END, STOP AND HALT STATEMENTS 15–1
15.1.1 PROGRAM 15–1
15.1.2 END 15–1
15.1.3 STOP 15–2
15.1.4 HALT 15–2

15.2 LABELS 15–3

15.3 MULTIPLE STATEMENTS 15–3

15.4 CONSTANTS 15–4

15.5 IDENTIFIERS 15–4
15.5.1 Integer variables 15–5
15.5.2 Real variables 15–5
15.5.3 String variables 15–5
15.5.4 Boolean Variables 15–5
15.5.5 Scalars 15–5
15.5.6 Arrays 15–5
15.5.7 Structure 15–6
15.5.8 Reserved Identifiers 15–6

15.6 EXPRESSIONS 15–6

15.7 MISCELLANEOUS STATEMENTS 15–7
15.7.1 RANDOMIZE 15–7
15.7.2 DELAY 15–8

APPENDIX A BUILT-IN FUNCTIONS A–1

A.1 SYSTEM FUNCTIONS A–1

A.2 OTHER FUNCTIONS A–3

APPENDIX B CONVENTIONS USED IN THIS MANUAL B–1

APPENDIX C MAKING LIBRARIES FOR USE WITH INTOUCH C–1

INDEX

EXAMPLES

1–1 To Use Setup 1–2
1–2 Setup Prompts 1–3
1–3 DEFINE Procedure 1–5
1–4 Field Contents Prompts 1–6
1–5 CREATE 1–11
1–6 Record Size Prompt 1–11
1–7 SHOW 1–12
1–8 SHOW Definitions 1–12
1–9 Field Information Displayed 1–13
1–10 EXIT 1–13
4–1 TRACE 4–3
4–2 LIST STATS 4–5
4–3 GO 4–7

FIGURES

1–1 Character Positions 1–8

TABLES

8–1 PARAMETERS 8–9
11–1 Concepts—Supported 11–20
A–1 DATE$ Function - Integer Values A–7
A–2 DAYS Function - Integer Values A–8
A–3 EDIT$ Function - Operation Values A–10
A–4 FULLTIME$ Function - Integer Values A–15

Preface

Introduction

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

The INTOUCH language:

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

INTOUCH includes a transparent interface to the RMS file management system. The interface includes:

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

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

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

Preface

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

INTOUCH's Specifications 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.

1 SETUP Routine

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 following procedures are available in SETUP:

- DEFINE

  Define fields in a structure. Use DEFINE when you add new fields, modify existing ones, or delete fields.

- CREATE

  Create a DATA File.

- SHOW

  Show field definitions. Use SHOW to see a display of the fields defined for this structure.

- EXIT

  Exit INTOUCH's SETUP routine. EXIT will save the work you have done on the structure and exit you back to the INTOUCH prompt.

Each of the SETUP procedures is defined on the following pages.

1.1.1 Entering SETUP

To enter SETUP, type CALL SETUP at the INTOUCH prompt:

Example 1–1 To Use Setup

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

 INTOUCH

 CALL SETUP









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

SETUP will ask you a number of questions including the structure name, data file name, record system being used, etc. You can create a new structure by entering new file names or modify existing structures by selecting them at the prompts.

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

1.2 SETUP Prompts

The following pages describe the SETUP prompts and your options for answering them.

Example 1–2 Setup Prompts

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status :
          S e t   u p   D a t a   S t r u c t u r e s

 Structure name? CUSTOMER
 Data file? CUSTOMER.DAT
 Record system? FASTFILE
 Definition file? CUSTOMER.DEF


 Procedure? define

            Procedures:  define, show, create, exit

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

- Structure name?

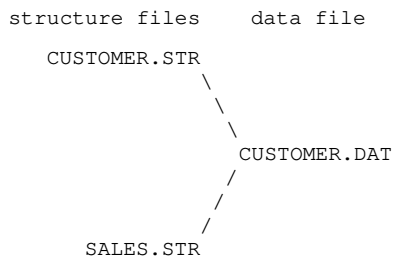
Enter a structure name. If you are creating a new structure, this will be the name of the structure file. The default extension for the file is .STR.

Your structure name can be any VMS file specification. See Chapter "INTOUCH RUNNING UNDER VMS" for further descriptions of the VMS file specifications.

- Data file? CUSTOMER

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

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



Your data file name can be any VMS file specification. See Chapter 8 for further descriptions of the VMS file specifications.

- Record system? RMS

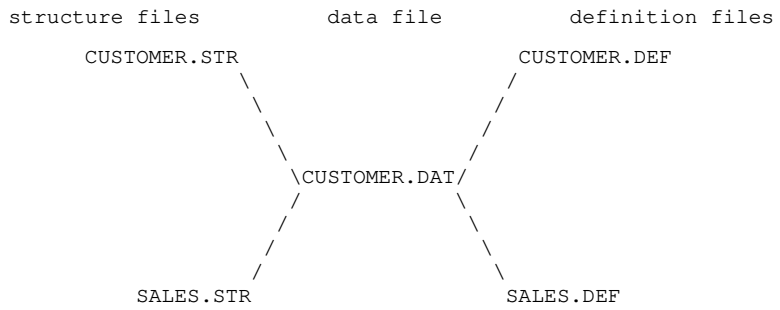
Enter the record management system you wish to use. INTOUCH supports several record management systems. They are displayed at the bottom of the screen when you get the record system prompt. (Please refer to the manual INTOUCH Interfaces to VAX Database Engines for an explanation of DBMS, ADABAS, Rdb, dBASE and CDD.)

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

- Definition file? CUSTOMER

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

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



Your definition file name can be any VMS file specifications. See chapter 8 for further descriptions on the VMS file specifications.

- Definition file? NONE

The keyword NONE is valid when the datafile is RDB or dBASE. When you enter NONE, INTOUCH uses the definitions embedded in the database engines with the fields.

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

- Definition file?

```
CDD>DEVDISK:[DEVINT.CDDTEST]CUSTOMER
    ^                       ^^      ^
    |                       ||      |
    +-----------------------++------+
       Directory name        record
                             name
```

CDD Plus access is read only at this time.

- Procedure? DEFINE

Select one of the procedure options.

1.3 DEFINE Procedure

You can use the DEFINE procedure to define the fields in a structure.

Choose the DEFINE procedure by typing DEFINE at the Procedure prompt. You will be asked first for the name of the field you wish to add or modify. You will then be prompted for information about the field.

Example 1–3 DEFINE Procedure

```
INTOUCH 4.1                                Program: NONAME
The Next Generation Language               Status :
                      Definitions for CUSTOMER

 Structure name? CUSTOMER
 Data file? CUSTOMER.DAT
 Record system? RMS
 Definition file? CUSTOMER.DEF



 Procedure? DEFINE

             Procedures:  define, show, create, exit

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

Next, you will be prompted to define the contents of the fields:

Example 1–4 Field Contents Prompts

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                     Definitions for CUSTOMER

 Field name? ID
 Description? Customer ID Number
 Data type? CH
 First position? 1
 Field length? 5
 Attributes? none
 Prompt text? Customer ID Number
 Report heading? Customer ID Number
 Print mask?
 Screen mask?
 Help text? Enter a 5-digit ID number
 Access rules? normal


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

- Field name?

Enter the name of the field you wish to add or modify. The field name:

must begin with a letter.

may only consist of numbers, letters and underscores "_".

may be up to 32 characters long.

If you are adding a new field, the field name will be used to identify this field. (See the SHOW procedure for a list of field names.)

You can delete a field by typing a minus sign "-", and then the field name. For example:

```
Field name? -ID
```

If you are modifying a field, the defaults for the rest of the DEFINE prompts will be your original responses. You can keep these defaults by pressing the RETURN key at the prompt. Change the defaults by retyping the response.

To return to the previous prompt, type a backslash "\" and then press the RETURN key. SETUP will return you to the previous prompt where you may change your response.

- Description? Customer ID number

Enter a description of the field. Your description:

can consist of any characters on the keyboard.

can be up to 30 characters long.

If you are adding this field, the field name is the default description.

- Data type? CH

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

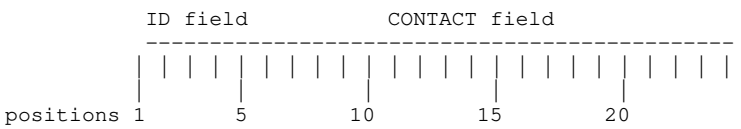
```
AC      ASCII counted

AP      Pointer to ASCII counted

C3      COBOL comp-3, also known as packed decimal; the
        C3 data type also supports the VAX DECIMAL data type

CH      Character

DS      Date stamp    (length=8)
DS      IBIS date     (length=6)

EB      EBCDIC

FL      Floating point

GF      G floating point

IN      Integer signed

IU      Integer unsigned

PF      Packed floating point

PZ      Packed zipcode

QS      Quadword signed

RO      Right overpunch

RS      Right sign separate

UN      Undefined

ZE      Zoned EBCDIC

ZN      Zoned punch; used by DIBOL
```

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

- First position? 1

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

Figure 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.

- Field length?

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

You can enter the ending-position for the field length. You enter "P ending-position".

Example: Length? P46

Variable Length Fields

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

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

- Attributes? none

Enter the attributes for this field. The field attributes available are:

```
DATE:YMD      This field contains a date.  The field must
     MDY      be six or eight digits long.

DBFLD:fieldname
              This is a special semantic for augmented
              definition files only.  The DBFLD semantic
              is used to associate an augmented field
              definition with an actual database field.

EB            Information to be stored as EBCDIC coded
              characters.
```

FULLTIME	Indicates that the field contains a date-time stamp and the time value is to be preserved. By default, INTOUCH stores/retrieves only the date portion of a date stamp field.
NUM	Allows numbers only.
NUM:scale	SCALE specifies the scale factor -- how many decimal places to use. Scale factors may be positive or negative (from -99 to 99).
RJ	Causes data to be right-justified.
UC	Enters all letters in upper-case.
ZF	Data entered is zero filled.
ZS	Causes all leading zeros to be suppressed and converted to spaces.

If you do not want any of these field attributes, enter NONE at the prompt. If you are adding a new field, NONE is the default. If you are modifying an existing field, the default will be the attribute you have previously specified.

- Prompt text? Customer ID Number

Enter the prompt text for this field. The prompt text will be used when data is entered for this field. The prompt text:

can use any characters on the keyboard.

can be up to 30 characters long.

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

- Report heading? Main Phone Number

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

- Print mask?

Enter a print mask for this field. The print mask is used to display information entered into this field. The print mask will be used whenever you use INTOUCH's PRINT statement to print field information, or when you use one of the INTOUCH routines. Print masks are created with the # and @ characters just as formats are created in PRINT USING. (See "Printing Data to a Terminal" for more information.)

- Screen mask?

The screen mask is used to enter data into this field. This prompt should be left blank for the time being.

- Help text?

Enter the help text for this field. The help text can be displayed during data entry if the user types HELP at this field prompt. Your help text:

can be up to 70 characters long.

can consist of any characters on the keyboard.

- Access rules? normal

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

When you have answered all the field prompts, the screen will clear and you may enter definitions for the next field:

```
Field name? PHONE
Description? Main phone number
Data type? CH
First position? 10
Field length? 137
Attributes? num, rj
Prompt text? Main phone number
Report heading? Main phone number
Print mask? (###)~ ###~-####
Screen mask?
Help text? Enter a phone number
Access rules? normal
```

When you have finished defining fields, type EXIT at the field prompt, or while holding down the CTRL key, press the letter "Z". You will return to the Procedure prompt:

- Field name? EXIT

1.4 Security

The SECURITY procedure allows structure security to be assigned.

The security levels assigned within CALL SETUP can be compared to security levels assigned to an individual user. The levels are used to control access.

```
          STRUCTURE SECURITY AND ACCESS LEVELS

Structure security level? N____
Read access level? N____
Update access level? N____
Write access level? N____
Delete access level? N____
```

1.5 CREATE Procedure

Choose the CREATE procedure by typing CREATE at the Procedure prompt.

You will be asked for the record size of your structure records. The legal record sizes depend on the file management system you use. For RMS, sizes range from 1 to approximately 32000. After you have entered the record size, your data file will be created using the appropriate record management system.

Note

Some record management systems may ask additional questions.

Example 1–5 CREATE

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




 Procedure? CREATE

             Procedures:  define, show, create, exit

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

Example 1–6 Record Size Prompt

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




 Record size? 200

            CUSTOMER.dat  created with record size 200

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

1.6 SHOW Procedure

The SHOW procedure displays information about the structure.

Choose the SHOW procedure by typing SHOW at the Procedure prompt:

You will be asked if you want the definitions for the defined fields. You may currently SHOW field definitions.

Example 1–7 SHOW

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




 Procedure? SHOW

             Procedures:  define, show, create, exit

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

Example 1–8 SHOW Definitions

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




 Show? definitions

                  Show: definitions, structure

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

1.7 Definitions

SHOW DEFINITIONS displays the following information for all defined fields: field name, description, beginning position, field length, data type, numeric attribute, scale factor, right-justification, zero fill and upper-casing. Field information is displayed in order by starting position.

Example 1–9 Field Information Displayed

```
INTOUCH 4.1                                  Program: NONAME
The Next Generation Language                 Status :
                     Definitions for CUSTOMER

Field name   Description        Pos - len  DT   N   SC   R  Z  U
----------   -----------------  ---------  --   -   --   -  -  -
ID           Customer ID number   1 -   5  CH   N        N  N  N
CONTACT      Primary contact      6 -  35  CH   N        N  N  N
COMPANY      Company name        41 -  30  CH   N        N  N  N
ADDR         Address line        71 -  25  CH   N        N  N  N
CITY         City                96 -  30  CH   N        N  N  N
STATE        State              126 -   2  CH   N        N  N  N
ZIP          Zip code           128 -   9  CH   Y        N  N  N
PHONE        Main phone number  137 -  10  CH   Y        N  N  N

                      Press RETURN to continue

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

After you have looked at the field definitions, type EXIT at the prompt. INTOUCH will return to the INTOUCH prompt:

Example 1–10 EXIT

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




 Procedure? EXIT

                 Show: definitions, structure

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

2 INTOUCH RUNNING UNDER VMS

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

```
filename.ext
```

"Filename" is the filename, and "ext" is the extension. A file name and extension:

- may each be up to, but not longer than 39 characters.
- may 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 data file, and the data dictionary.

- The structure file contains the name of the data file and record management system used, and the name of the data dictionary.
- 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 must be opened before any reference to it may occur.
- 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 may appear anywhere a normal variable may. 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 may 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 may use a string constant to specify the field name. If you give the field name as a string constant, you need not enclose it in quotes. INTOUCH will use the string constant as the field name:

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

If you specify the field as 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 my California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(STATE),CL(PHONE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END
```

```
RNH
List of my California Clients

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

SUMMARY:

OPEN 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 may 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 my California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END
```

SUMMARY:

CLOSE STRUCTURE closes a structure from further access. You should close your structure before your program ends.

3.2.3 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 my California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(STATE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END
```

SUMMARY:

EXTRACT 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.

3.2.4 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
30 REEXTRACT STRUCTURE CL
     EXCLUDE CL(PHONE)[1:3] = '619'
     SORT ASCENDING BY CL(LAST)
   END EXTRACT
40 PRINT 'List of my California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
50 CLOSE STRUCTURE CL
60 END
```

SUMMARY:

REEXTRACT 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.

3.2.5 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
30 PRINT 'List of my California Clients'
   PRINT
   FOR EACH CL
     PRINT CL(FIRST);' ';CL(LAST),CL(PHONE)
   NEXT CL
40 CLOSE STRUCTURE CL
50 END
```

SUMMARY:

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

3.2.6 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  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.

3.2.7 EXIT EXTRACT

FORMAT:

```
EXIT EXTRACT
```

EXAMPLE:

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
    EXTRACT STRUCTURE CL
      PRINT 'Client: '; CL(LAST)
      LINE INPUT 'Press return to continue': Z$
      IF  _EXIT  THEN  EXIT EXTRACT
    END EXTRACT
    PRINT 'Records extracted:'; _EXTRACTED
20  END

RNH
Client: Smith
Press return to continue? {RETURN}
Client: Jones
Press return to continue? EXIT
Records extracted: 1
```

SUMMARY:

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

3.2.8 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"
    INPUT 'Name': NAME$
    EXTRACT STRUCTURE CL, FIELD LAST: PARTIAL KEY NAME$
      PRINT CL(LAST)
    END EXTRACT
20  END

RNH
Name? S
Smith
```

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.

3.2.9 EXTRACT STRUCTURE: APPEND

FORMAT:

```
EXTRACT STRUCTURE struc_name : APPEND
```

EXAMPLE:

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

RNH
```

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

SUMMARY:

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

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

3.2.11 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:

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

3.2.12 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
35 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.

3.2.13 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 client id number': ID%
   INPUT 'Enter client last name': LAST$
   INPUT 'Enter client first name':FIRST$
   INPUT 'Enter client state':STATE$
   INPUT 'Enter client phone':PHONE$
30 ADD STRUCTURE CL
     PRINT 'Adding '; LAST$; ', '; FIRST$
     LET CL(ID) = ID%
     LET CL(LAST)= LAST$
     LET CL(FIRST)= FIRST$
     LET CL(STATE) = STATE$
     LET CL(PHONE) = PHONE$
   END ADD
40 CLOSE STRUCTURE CL
50 END
```

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.

3.2.14 CANCEL ADD

FORMAT:

```
CANCEL ADD
```

EXAMPLE:

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT", ACCESS OUTIN
    ADD STRUCTURE CL
      INPUT 'Client ID': CL(ID)
      IF  _EXIT  THEN  CANCEL ADD
      INPUT 'Last name': CL(LAST)
      INPUT 'First name': CL(FIRST)
      PRINT 'Adding client'
    END ADD
20  END

RNH
Client ID? EXIT
```

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.

3.2.15 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.

3.2.16 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 CL(FIRST); ' '; CL(LAST)
     LOCK STRUCTURE CL   ! Gives us exclusive access
                         ! to this record
     INPUT 'Enter the new phone number':PHONE$
     LET CL(PHONE) = PHONE$
     UNLOCK STRUCTURE CL ! Puts the record out to disk
                         ! and releases it
   NEXT CL
40 CLOSE STRUCTURE CL
50 END


RNH
Earl Errant
Enter the new phone number: 2135557676
Kathy Cass
Enter the new phone number: 5554995663
Dale Derringer
Enter the new phone number: 8185559014
INTOUCH
```

SUMMARY:

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

3.3 SET STRUCTURE

SET STRUCTURE allows structure manipulation. You can use the SET structure statement to change various device and structure characteristics of INTOUCH structures. Listed below are the format and description of the SET statement with its STRUCTURE options:

3.3.1 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
30  END
```

SUMMARY:

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

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

3.3.3 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:

The **SET STRUCTURE struc_name1, SET 'set_name', USING struc_name2: EXPRESSION 'owner'** statement used for DBMS handling fetches the owner of the structure specified by struc_name1. The structure specified by struc_name2 is the owner of struc_name1 with the set given in 'set_name'.

3.3.4 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:

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

3.3.5 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:

The **SET STRUCTURE struc_name1, SET 'set_name', USING 'struc_name2', FIELD field_expr: KEY str_expr** statement fetches the first 'struc_name1' record using the key given in str_expr within the set, 'set_name', where 'struc_name2' is the owner.

3.3.6 SET STRUCTURE: CURRENT

FORMAT:

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

EXAMPLE:

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

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 ""
```

3.3.7 SET STRUCTURE FIELD:KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE TEST : NAME 'TTI_RUN:CUSTOMER'
20  SET STRUCTURE TEST, FIELD CUSTNBR : KEY '13727'
30  PRINT TEST(NAME)
40  CLOSE ALL
50  END

RNH
Tumbleweeds Real Estate
```

SUMMARY:

The FIELD option lets you specify a field in a structure. This option is currently used in conjunction with the KEY option. 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. The KEY must be a string expression.

3.3.8 SET STRUCTURE FIELD: PARTIAL KEY

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
    INPUT 'Name': NAME$
    SET STRUCTURE CL, FIELD LAST: PARTIAL KEY NAME$
    PRINT CL(ID); ' '; CL(LAST)
20  END

    RNH
    NAME? S
    32001 Smith
```

SUMMARY:

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

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

3.3.10 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 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.

3.3.11 SET STRUCTURE: RECORD

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE ML : NAME "TTI_RUN:MAINT_LOG"
    SET STRUCTURE ML : RECORD 3
    PRINT ML(COST)

RNH
$370.00
```

SUMMARY:

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

3.3.12 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.

3.4 ASK STRUCTURE

ASK STRUCTURE allows structure manipulation. You can use the ASK statement to ask about various device and structure characteristics from within your programs. Listed below are the formats and descriptions of the ASK statement with the STRUCTURE options:

3.4.1 ASK STRUCTURE: CURRENT

FORMAT:

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

EXAMPLE:

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

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.

3.4.2 ASK STRUCTURE: FIELDS

FORMAT:

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

SUMMARY:

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

3.4.3 ASK STRUCTURE: KEYS

FORMAT:

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

SUMMARY:

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

3.4.4 ASK STRUCTURE FIELD: item

FORMAT:

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

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.

3.4.5 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
INDEX,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.

3.4.6 ASK STRUCTURE: ENGINE

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE c1 : name 'tti_run:vendor'
30  ASK STRUCTURE  c1 : ENGINE ename$
40  PRINT 'DATABASE ENGINE is:'; ename$
50  END

RNH

DATABASE ENGINE is: RMS
```

SUMMARY:

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

3.4.7 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
 11
```

SUMMARY:

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

3.4.8 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 a structure and stores it into the string variable given.

3.4.9 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 STRUCTURE_NAME block, ASK STRUCTURE: POINTER asks the structure for the number of the current pointer.

3.4.10 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.

3.4.11 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.

3.4.12 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.

3.4.13 ASK STRUCTURE: TIMEOUT

FORMAT:

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

SUMMARY:

This statement asks for the current TIMEOUT setting.

3.5 ASK | SET STRUCTURE #string_expr

FORMAT:

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

EXAMPLE:

```
10  OPEN STRUCTURE CL: NAME 'TTI_RUN:CLIENT'
    STR$ = 'CL'
    FLD$ = 'ID'
    GOSUB DO_WORK
    STOP

20  ROUTINE DO_WORK
    ASK STRUCTURE #STR$, FIELD #FLD$: DESCRIPTION DSC$
    PRINT 'Description is: '; DSC$
    END ROUTINE

90  END

RNH
Description is: Client ID number
```

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.

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
1050 DEBUG ON
```

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.

Example 4–1 TRACE

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

 INTOUCH

 TRACE ON

 INTOUCH

 RUN

 INTOUCH

 TRACE ON                        10     10.1     10.2      10.3


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

4.2.2 TRACE OFF

FORMAT:

```
TRACE OFF
```

EXAMPLE:

```
TRACE OFF
1050 TRACE OFF
```

SUMMARY:

Use TRACE OFF to turn the trace feature off.

4.3 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 was 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 speeding up program execution.

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 LIST STATS

```
 INTOUCH 4.1                          Program: NONAME  STATS DEBUG
The Next Generation Language         Status : RUN

 LIST STATS
  NONAME              6-DEC-1984           18:51
       1         0.00          10 I=1
     100         0.25             DO WHILE I<100
      99         1.07               ROOT = SQR(I)
      99         5.77               PRINT I, ROOT
      99         0.28               I=I+1
      99         0.04             LOOP
       1         0.02          20 END

 INTOUCH


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

4.4 BREAK and STEP Features

4.4.1 BREAK

FORMAT:

```
BREAK
```

EXAMPLE:

```
1050 BREAK
```

SUMMARY:

Use BREAK to interrupt program execution. The BREAK statement will only be executed when DEBUG is on. For instance, you might use BREAK to stop program execution if a variable is assigned an illegal value.

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.

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 4–3 GO

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

10 FOR I=1 TO 6
20   IF  I=4  THEN BREAK
30 NEXT I
40 END
RNH
Break at 20

INTOUCH

25  PRINT I;    ! add a new line of code
GO
4 5 6


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

5 INTOUCH COMMANDS

5.1 The EXIT and HELP Commands

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

EXIT

The user may exit an INTOUCH procedure, routine or the INTOUCH environment by typing EXIT at an available prompt.

Typing EXIT at the INTOUCH prompt will cause the user to exit INTOUCH and return to the VAX prompt ($). Typing EXIT in an INTOUCH routine will cause the user to do 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

The user can get help by typing HELP. Typing HELP at the INTOUCH prompt or in an INTOUCH routine will provide the user with helpful information.

INTOUCH has an extensive library of HELP text on many topics. Typing HELP by itself will cause INTOUCH to display a list of topics and the prompt "Topic?". The user chooses a topic. INTOUCH provides information about that topic. INTOUCH may display a list of sub-topics and the prompt "Subtopic?". This allows users to get more detailed information.

The user can ask for specific information. For example:

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

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

5.2 RUN

FORMAT:

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

EXAMPLE:

```
RUN "HELLO"
    or
RNH "HELLO"
```

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.

5.3 NEW

FORMAT:

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

EXAMPLE:

```
NEW "HELLO"
```

SUMMARY:

To write a new program, type 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 length of the workspace name may now be up to, but not longer than, 19 characters.

5.4 SAVE

FORMAT:

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

EXAMPLE:

```
SAVE "HELLO"
```

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".

5.5 REPLACE

FORMAT:

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

EXAMPLE:

```
REPLACE "HELLO"
```

SUMMARY:

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

5.6 ABORT

FORMAT:

```
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 to pass to the operating system:

```
ABORT int_expr
```

5.7 OLD

FORMAT:

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

EXAMPLE:

```
OLD "HELLO"
```

SUMMARY:

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

5.8 UNSAVE

FORMAT:

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

EXAMPLE:

```
UNSAVE "HELLO"
```

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".

5.9 LIST

FORMAT:

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

EXAMPLE:

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

SUMMARY:

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

5.10 LIST ERRORS

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

5.11 EDIT

FORMAT:

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

EXAMPLE:

```
EDIT 510-570, FRED
```

SUMMARY:

EDIT transfers control to the editor. The 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 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 and labels. Only the sections you reference will be moved into the editor.

5.12 RENUMBER

FORMAT:

```
RENUMBER
```

EXAMPLE:

```
RENUMBER
```

SUMMARY:

RENUMBER operates on the program in current memory. RENUMBER 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.

5.13 FRAME ON | OFF

FORMAT:

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

EXAMPLE:

```
FRAME ON
FRAME OFF
```

SUMMARY:

Turns the INTOUCH window frame–the bars across the top and bottom of your screen–off or on.

5.14 REFRESH

FORMAT:

```
REFRESH
```

EXAMPLE:

```
REFRESH
```

SUMMARY:

Use REFRESH when your screen gets garbled. REFRESH repaints the screen, the frame and the INTOUCH prompt.

5.15 IDE

FORMAT:

```
IDE
```

EXAMPLE:

```
IDE
```

SUMMARY:

Use IDE if you need to know the version number for your copy of INTOUCH. Use the version number when contacting Touch Technologies, Inc. with suggestions or errors.

5.16 COMPILE

FORMAT:

```
COMPILE ["file_spec"]
```

EXAMPLE:

```
COMPILE "HELLO"
```

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 VAX prompt by typing INTOUCH followed by the filename:

```
$ INTOUCH HELLO
```

INTOUCH will begin and the program will be olded and run. When the program has finished running, you will return to the VAX prompt.

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

COMPILED IMAGES

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

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

```
$ INTOUCH/COMPILE filename
```

Example:

```
$INTOUCH/COMPILE myprogram
```

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

Example:

```
Compiling:
    $INTOUCH/COMPILE CUSTOMER

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

5.16.1 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 myprogram
```

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"
```

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

```
$INTOUCH/DEVELOP
```

6 DIRECTIVES

6.1 %INCLUDE

FORMAT:

```
%INCLUDE "File_spec"
```

EXAMPLE:

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

SUMMARY:

Use 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.

6.1.1 %INCLUDE CONDITIONAL

FORMAT:

```
%INCLUDE CONDITIONAL: "include_file"
```

SUMMARY:

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

6.2 %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 doesn't show up in production systems.

6.3 %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.

6.3.1 %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.

7 ASSIGNING VARIABLES

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

7.1 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
```

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.

7.2 LSET

FORMAT:

```
LSET str_var = str_expr
```

EXAMPLE:

```
10 HEADING$ = REPEAT$('.', 20)  ! HEADING$=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.

7.3 RSET

FORMAT:

```
RSET str_var = str_expr
```

EXAMPLE:

```
10 HEADING$ = REPEAT$('.',20) ! HEADING$=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.

7.4 CSET

FORMAT:

```
CSET str_var = str_expr
```

EXAMPLE:

```
10 HEADING$ = REPEAT$('.', 20)  ! HEADING$=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.

7.5 LSET, RSET, CSET FILL

FORMAT:

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

EXAMPLE:

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

RNH
$1111
```

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.

7.6 DATA/READ

FORMAT:

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

EXAMPLE:

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

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.

7.7 RESTORE

FORMAT:

```
RESTORE
```

EXAMPLE:

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

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 may be used again by the next READ statement.

7.8 DECLARE STRING | INTEGER | REAL

FORMAT:

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

EXAMPLE:

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

SUMMARY:

Use the DECLARE statement to declare the data type of a variable. That way you don't 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.

7.9 DECLARE DYNAMIC

FORMAT:

```
DECLARE DYNAMIC var1 [, var2 ...]
```

EXAMPLE:

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

RNH
 9
Bob
```

SUMMARY:

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.

7.10 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.

7.11 DECLARE PREFIX

FORMAT:

```
DECLARE PREFIX str_expr
DECLARE PREFIX DOIT
```

EXAMPLE:

```
10  A = 100
20  EXECUTE "declare prefix excode \ a = 45 \ print a"
30  PRINT A
40  EXECUTE "print excode$a"
50  END

RNH
45
100
45
```

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

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

```
STRUCT_NAME(#"field_name")
```

For example:

```
VEND(city)  --> VEND(#"city")
```

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

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

EXAMPLE:

```
 10  OPEN STRUCTURE AR$CUST : NAME "TTI_RUN::CUSTOMER", &
       ACCESS INPUT
     GOSUB FIND_CUSTOMER
     STOP
100  ROUTINE FIND_CUSTOMER
       DECLARE PREFIX FIND
       C$ = "13727"
       SET STRUCTURE AR$CUST, FIELD CUSTNBR : KEY C$
       PRINT AR$CUST(#"name")
     END ROUTINE
999  END
```

SUMMARY:

This statement causes all non-prefixed variable references, structure name references, and field name references to inherit a prefix of DOIT or a prefix that is explicitly given.

```
PRINT TOTAL    <-->  PRINT DOIT$TOTAL
PRINT NAME$    <-->  PRINT DOIT$NAME$
PRINT XYZ$SUM  <-->  PRINT XYZ$SUM    (prefix is
                                  explicitly given)
```

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

```
ROUTINE xxxx
END ROUTINE
DECLARE PREFIX NONE
```

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

7.12 OPTION REQUIRE DECLARE

FORMAT:

```
OPTION REQUIRE DECLARE
```

EXAMPLE:

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

SUMMARY:

OPTION REQUIRE DECLARE causes INTOUCH to require that all variables in the 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.

8 INPUTTING DATA FROM A TERMINAL

8.1 INPUT

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

For information on INPUT from a text file, see the chapter, "File Handling."

The user types data at the terminal in response to the INPUT statement. This response is assigned to the variable(s) listed. The user's data must be the same data type as the variable, or INTOUCH generates an exception. If in response to the INPUT statement, the user presses the RETURN key and:

the variable is numeric, INTOUCH assigns it a 0.

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

FORMAT:

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

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

SUMMARY:

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

8.1.1 Types of INPUT Statements

There are three types of INPUT statements:

- INPUT

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

- LINE INPUT

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

- KEY INPUT

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

8.1.2 INPUT Styles

There are four input styles:

- simple input from the terminal

  Example:

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

 INTOUCH

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

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

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

  Example:

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

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

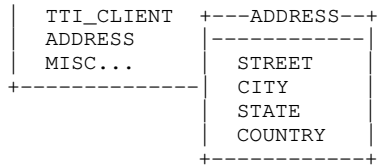
  Example:

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

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

  Example:

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

     RNH
```



8.2 INPUT Options

The INPUT statement has the following options. (Please refer to individual options, described in detail later in this chapter):

PROMPT	Displays the prompt text.
AT row, col	Positions the cursor on the desired row and column.
ERASE	Clears the input line prior to accepting input and after the input has been completed.
LENGTH nn	Limits the number of characters that a user can type.
DEFAULT	Lets you provide defaults for INPUT statements.
VALID	Validates user responses.
TIMEOUT	Limits the time the user has to respond to the INPUT prompt.
AREA	Does free format multiline text input from an area on the screen.
SCREEN	Creates formatted data entry screens.
MENU	Displays and receives input from "pop-up" menus.
ELAPSED	Keeps track of the time it takes the user to respond to an INPUT.

8.3 INPUT COMMANDS and Keystrokes Available to the User

At an input, there are several commands and keystrokes that the user can enter instead of an answer for an input.

EXIT or CTRL/Z

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

\ or Up-arrow

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

HELP

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

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

8.3.1 Editing Input

If users want to edit their input, they can use the following keys:

Left-arrow	Move left.
Right-arrow	Move right.
Delete key	Delete character to the left of the cursor.
TAB	Move to next input field or terminate input.
CTRL/A	Toggle between insert and overstrike mode.
CTRL/E	Move to end of line.
CTRL/H	Move to beginning of line.
CTRL/J	Delete the word to the left of the cursor.
CTRL/U	Delete all the characters to the left of the cursor.
Return	Terminate the input. You can press Return anywhere in the line; you do not have to be at the end of the line.

8.4 Inputting Strings

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

```
10 INPUT NAME$
   PRINT NAME$
20 END

RNH
? JOHN
JOHN

INTOUCH
```

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

```
RNH
? '  John'
  John

INTOUCH
```

8.5 Inputting Multiple Variables

8.5.1 Simple Input Statement

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

```
10 INPUT NAME$, AGE%, SEX$
20 PRINT NAME$, AGE%, SEX$
30 END

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

INTOUCH
```

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

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

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

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

INTOUCH
```

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

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

INTOUCH
```

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

8.5.2 LINE INPUT Statement

The LINE INPUT statement gets a full line of data from the terminal. It is used in those cases where an entire line from the terminal must be returned to the program. LINE INPUT accepts everything the user types up until a line terminator is pressed, as the input line. (Line terminators are: RETURN key, "\", CTRL/Z and TAB.) It includes commas, semicolons, etc. All 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 commas.

```
10 LINE INPUT COMMENT_1$, COMMENT_2$
   PRINT COMMENT_1$
   PRINT COMMENT_2$
20 END
```

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

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

INTOUCH
```

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

8.6 "PROMPT TEXT"

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

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

8.7 PROMPT Option

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

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

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

8.8 AT Option

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

Row and col may be any integer numeric constants.

8.9 LENGTH Option

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

8.10 DEFAULT Option

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

Pre-positioning to a Default Menu Item

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

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

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

Upon the completion of a *INPUT MENU* statement, the function *_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.)

8.11 VALID Option

The VALID Option validates user responses according to validation rules.

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

8.12 ERASE option

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

8.13 TIMEOUT Option

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

A 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 name. Fractions of a second can be indicated by including decimal digits.

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

8.14 ELAPSED Option

The ELAPSED Option keeps track of the time it takes the user to respond to an INPUT. A '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.

8.15 ATTRIBUTES option

FORMAT:

```
[LINE] INPUT ATTRIBUTES attr_list : str_var
LINE INPUT AREA row1, col1, row2, col2, ATTRIBUTES attr_list: str_var
INPUT SCREEN str_expr1, ATTRIBUTES attr_list: str_var
```

EXAMPLE:

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

SUMMARY:

The ATTRIBUTES option does input with attributes. Available attributes are the following: BOLD, BLINK, REVERSE, UNDERLINE. Multiple attributes used in one INPUT statement are separated by commas.

8.16 AREA Option

FORMAT:

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

Table 8–1 PARAMETERS

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

SUMMARY:

The AREA Option does input from an area on the screen. During a line area input, the following features are available:

- Automatic word-wrap while text is being entered
- Insert and delete capability
- Use of left, right, up, and down arrow keys

While you are entering text in an area, you can use the following 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/T	Moves cursor to the top of the area
GOLD/B	Moves cursor to the bottom of the area
GOLD/F	Complete (finish) and save the input
F10	Exit (abort) the input
CTRL/Z	Exit (abort) the input

HELP	Exits from the input and sets the variable _HELP to true
\	EXIT

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

8.17 SCREEN Option

FORMAT:

```
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 may be used within the format to control how the fill-in area works. These fall into two categories: format Options and format characters. Format Options come first. They are followed by a colon and then by format characters. If no format Options are used, place a colon after the first angle bracket.

8.17.1 Screen Format Characters

#

The # is used to specify digits. Wherever a # sign appears, only these types of characters are allowed as input. For instance, the # might be used to get the user's phone number:

@

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 ~ is used to specify that any character that follows the ~ will not be replaced.

8.17.2 Screen Format Options

If you use format Options, they must be placed before the format characters and followed by a colon. Please refer to *INTOUCH A Guide to the Language* for a list of format Options.

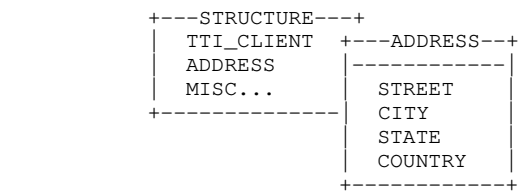
8.18 MENU Option

FORMAT:

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

EXAMPLE:

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

RNH
```



SUMMARY:

Use the MENU Option 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 **%multi** is used and 'SEL' key is pressed, "Str_var" contains the names of the items, separated by a linefeed character.

Elements can be either items or directives.

An item consists of either a description or a description and = sign followed by either a data value or a sub-menu indicator. If no = sign is given, the data value is the same as the description. If the description contains spaces or special characters, it must be enclosed within quotes.

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

The sub-menu consists of a "{" followed by menu elements, followed by "}".

8.18.1 Menu DIRECTIVES:

%TITLE "....."

Assigns a title to this menu.

%AT row, column

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

Example:

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

%REPLACE

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

%MULTI

Allows multiple items to be selected.

%BAR ["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.

%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. vertically scrolled using the UP, DOWN, NEXT and PREV keys.

%WIDTH number

Controls the minimum width of the current column.

%MAXWIDTH number

Controls the maximum width of the entire 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.

%COLUMNS number

Sets the number of columns for the menu.

%ITEMS number

Creates a multi-column menu.

%SPLIT

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

%MENUBAR

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

AUTOVBAR [ON|OFF]

Turns vertical bars ON and OFF in menus.

%VBAR

Inserts a vertical bar.

%AUTODISPLAY [ON | OFF]

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

User Interface

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

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

8.19 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 KEY INPUT. KEY INPUT returns the following:

```
_BACK
_EXIT
_HELP
_TERMINATOR
```

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

9 Printing Data to a Terminal

9.1 PRINT

FORMAT:

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

EXAMPLE:

```
10 INPUT NAME$
   PRINT 'Hello, ';NAME$
   PRINT BOLD, UNDERLINE: 'Here is a number';1.93
20 END
```

SUMMARY:

The PRINT statement is used to print text on the screen. The comma causes the next item to be printed in a new print zone. The semicolon causes the next item to be printed immediately following the first. If a message line is being altered, and a message requiring a delay is present, the print will delay.

For more information, please refer to the chapter on file handling.

9.1.1 PRINT Options

The PRINT statement has the following options:

AT Option

The AT Option prints the first variable at the coordinates specified.

BLINK | UNDERLINE | BOLD | REVERSE Options

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

WIDE Option

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

FORMAT:

```
PRINT WIDE:expr
```

EXAMPLE:

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

ERASE Option

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

FORMAT:

```
PRINT ERASE: expr
```

EXAMPLE:

```
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
```

PRINT USING Option

Use USING Option to specially format text. USING Option lets you print text to the screen using an edit mask. This allows you to format output uniformly–even when you don't know what that output is going to be.

```
10 LET A$ = '#.## ##.##'
20 PRINT USING '#.##': 1.9
30 PRINT USING A$: 1.93, -1.93
RNH
1.90
1.93 -1.93

INTOUCH
```

USING Option has the following directives:

- *{UCASE}*

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

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

  RNH
  MARCH

  INTOUCH
  ```

- *{LCASE}*

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

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

  RNH
  march
  ```

- *{HYPHEN}*

  The **hyphen** directive causes INTOUCH to suppress the hyphen character if it is the last non-blank character after the format is applied.

  ```
  10  print using '<#####~-####' : '92123'
      print using '{hyphen}<#####~-####' : '92123'
      end
  RNH

  92123-
  92123
  ```

9.1.2 ? as a Substitute for PRINT Statement

FORMAT:

```
?
```

EXAMPLE:

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

SUMMARY:

The "?" can be used as a synonym for PRINT at the command prompt.

9.1.3 Print Format Characters

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

#	The # is used to indicate a character position. The # can also be used to specify numeric characters.
@	The @ 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.
```

Semicolons

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

Commas

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

9.2 Using the INTOUCH Screen

9.2.1 MESSAGE [ERROR: | DELAY:]

FORMAT:

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

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$
20 WHEN EXCEPTION IN
     INPUT 'How old are you': AGE
   USE
     MESSAGE ERROR: 'Not a valid age'
     RETRY
   END WHEN
30 PRINT
   PRINT NAME$; ' is'; AGE
40 END
```

SUMMARY:

INTOUCH uses the message line (defaults to line 23) to display messages in bold print and errors. 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. Multiple 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 your 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 lengthier messages.

9.3 CLEAR

FORMAT:

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

EXAMPLE:

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

PURPOSE:

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

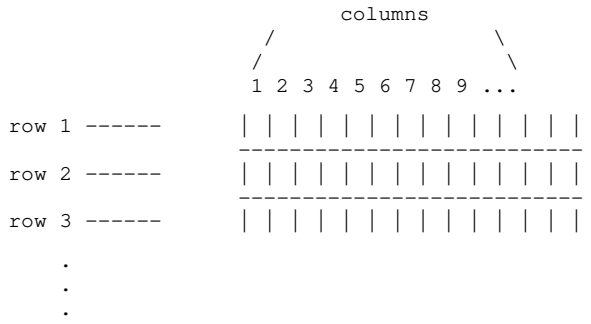
SUMMARY:

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

9.3.1 AREA Option

The AREA option clears a specific section of the screen. The area is rectangular in shape. Row specifies a vertical position on the screen. Rows are numbered sequentially from the top of the screen to the bottom. With the INTOUCH frame on, there are typically 19 rows. With the frame off, there are usually 24 rows.

Col specifies a column–a horizontal position on the screen. Columns are numbered sequentially from the first character position on the left of the screen to the last character position on the right side. Usually there are either 80 or 132 columns (depending on the terminal setting).



Two coordinates must be specified. These coordinates mark opposite corners of a rectangular area. INTOUCH clears the rectangular area specified by these two coordinates. For instance, the statement CLEAR AREA 2,3,8,20 would clear a rectangular area:

The first coordinate (2,3) indicates the upper-left corner. The second coordinate (8,20) indicates the lower-right corner.

FORMAT:

```
CLEAR AREA attr_list: row1,col1,row2,col2
```

EXAMPLE:

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

SUMMARY:

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

9.3.2 BOX Option

The BOX option creates an empty box with a frame.

FORMAT:

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

EXAMPLE:

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

SUMMARY:

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

9.3.3 OPTION COMMUNICATE ON | OFF

FORMAT:

```
OPTION COMMUNICATE ON | OFF
```

EXAMPLE:

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

SUMMARY:

Use to turn INTOUCH message trapping on or off.

10 DEFINING ARRAYS

Arrays are a type of variable. They are used to store and manipulate tables of variable information. An array must be defined before it is used in a program.

10.1 DIM

FORMAT:

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

EXAMPLE:

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

SUMMARY:

DIM 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.

10.2 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:

```
5  DIM NAME$(4)
10 INPUT 'How many names': NUM
   REDIM NAME$(NUM)
20 FOR I=1 TO NUM
     INPUT NAME$(I)
   NEXT I
30 PRINT
40 FOR I=1 TO NUM
     PRINT NAME$(I)
   NEXT I
```

SUMMARY:

If you have already dimensioned an array and now want to change the size of its dimensions, use the REDIM statement. REDIM works in the same manner and has the same options as the DIM statement.

You can dynamically expand arrays as needed while keeping current array values. If you REDIM an array to a larger size, the old values are kept. If your applications depend on REDIM initializing all array values, change your code as follows:

Old Code

```
REDIM X(100)
```

New Code

```
REDIM X(1)
REDIM X(100)
```

The REDIM X(1) forces all array values to be initialized by the second REDIM statement.

10.3 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
```

SUMMARY:

Normally, when no low bound is specified for a dimension, INTOUCH gives the dimension a low bound of 1. The OPTION BASE statement lets you specify a default low bound of 0 or 1.

11 Set and Ask Statements

SET and ASK find and change characteristics within an INTOUCH program. SET sets various characteristics, and ASK returns the value of various characteristics. SET and ASK have several different options.

Note

For information on SET #chnl and ASK #chnl statements, please refer to the chapter, "File Handling" in the A Guide to the Language manual.

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

11.1 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
```

SUMMARY:

Use to slowly back a user out of a program if the terminal is left idle. The num_expr is expressed in minutes.

11.2 SET BACK ON | OFF

FORMAT:

```
SET BACK OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: REPLY$
20  PRINT _BACK
30  SET BACK OFF
40  PRINT _BACK
50  END

RNH
Name? ______________________________
 0
 0
```

SUMMARY:

SET BACK OFF sets the internal variable _BACK to FALSE (0). SET BACK ON sets the internal variable _BACK to TRUE (1).

11.3 SET ERROR ON | OFF

FORMAT:

```
SET ERROR ON | OFF
```

EXAMPLE:

```
10  DO
      SET ERROR OFF
      INPUT 'Enter the age': age
      IF  AGE < 1  THEN
        PRINT 'Too young: '; age
        SET ERROR ON
      ELSE
        SET ERROR OFF
      END IF
    LOOP WHILE _ERROR
20  END
```

SUMMARY:

The following methods can be used to SET the _ERROR flag:

- SET ERROR ON
- MESSAGE ERROR: statement
- Execution of the END WHEN statement

The following methods can be used to CLEAR the _ERROR flag:

- SET ERROR OFF
- GOSUB or DISPATCH statement
- WHEN EXCEPTION IN statement

11.4 ASK ERRORS

FORMAT:

```
ASK ERRORS num_var
```

SUMMARY:

Asks for the number of user errors (MESSAGE ERROR increments this).

11.5 SET EXIT ON | OFF

FORMAT:

```
SET EXIT OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: reply$
20  PRINT _EXIT
30  SET EXIT OFF
40  PRINT _EXIT
50  END

RNH
Name? EXIT_____________________________
 0
 0
```

SUMMARY:

SET EXIT OFF sets the internal variable _EXIT to FALSE (0). SET EXIT ON sets the internal variable _EXIT to TRUE (1).

11.6 SET HELP ON | OFF

FORMAT:

```
SET HELP OFF | ON
```

EXAMPLE:

```
10  LINE INPUT 'Name', LENGTH 30: REPLY$
20  PRINT _HELP
30  SET HELP OFF
40  PRINT _HELP
50  END
RNH

Name? HELP_____________________________
 0
 0
```

SUMMARY:

SET HELP OFF sets the internal variable _HELP to FALSE (0). SET HELP ON sets the internal variable _HELP to TRUE (1).

11.7 ASK KEYSTROKES

FORMAT:

```
ASK KEYSTROKES num_var
```

SUMMARY:

Asks for the number of user-entered keystrokes.

11.8 ASK | SET MARGIN

FORMAT:

```
ASK MARGIN num_var
SET MARGIN num_expr
```

EXAMPLE:

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

RNH
................................................
................................................
................................................
................................................

What do you want the margin set to? 20
..................
..................
..................
..................
```

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. Margin must be greater than zonewidth.

11.9 ASK | SET MESSAGELINE

FORMAT:

```
ASK MESSAGELINE num_var
SET MESSAGELINE num_var
```

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.

11.10 SET PORT ON | OFF

FORMAT:

```
SET PORT ON | OFF
```

EXAMPLE:

```
10  PRINT 'About to print 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
```

SUMMARY:

Use to turn the printer port on or off.

11.10.1 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
```

11.11 ASK RESPONSES

FORMAT:

```
ASK RESPONSES num_var
```

SUMMARY:

Asks for the number of completed input responses.

11.12 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:;
    PRINT 'This'
    PRINT 'text'
    PRINT 'will'
    PRINT 'scroll.'
20  SET SCROLL  1,24
30  END
```

SUMMARY:

This statement sets up a scrolling region from line int_expr1 to line int_expr2.

11.13 ASK | SET SEED

FORMAT:

```
ASK SEED num_var
ASK SEED num_expr
```

EXAMPLE:

```
10 RANDOMIZE
   ASK SEED seed_num
   FOR I = 1 TO 3
     PRINT RND(1000)
   NEXT I
   PRINT 'Reset the random sequence'
   SET SEED seed_num
   FOR I = 1 TO 3
     PRINT RND(1000)
   NEXT I
20 END
```

```
RNH
 608
 88
 506
Reset the random sequence
 608
 88
 506
```

SUMMARY:

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.

11.14 ASK | SET SYSTEM

ASK SYSTEM and SET SYSTEM have many options. The options are described in the following sections.

11.14.1 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:

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 loads the INTOUCH operating system comment area with the specified string.

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.

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

SUMMARY:

ASK SYSTEM: DIRECTORY asks the operating system for the current device and directory.

SET SYSTEM: DIRECTORY sets the default device and directory.

11.14.3 ASK | SET SYSTEM, LOGICAL: VALUE

FORMAT:

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

EXAMPLES:

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

RNH
Logical set to FRED


10  ASK SYSTEM, LOGICAL 'SYS$SCRATCH': VALUE SCR$
20  PRINT '"SYS$SCRATCH" points to: '; SCR$
30  END

RNH
SYS$SCRATCH points to USER:[SUZANNE]
```

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.

11.14.4 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:

```
INTERACTIVE
BATCH
NETWORK
OTHER
```

11.14.5 ASK SYSTEM: PARAMETER

FORMAT:

```
ASK SYSTEM: PARAMETER str_var
```

EXAMPLE:

```
$INTOUCH/IMAGE PROOF myfile

The string "myfile" is returned by the
ASK SYSTEM: PARAMETER statement.

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

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

SUMMARY:

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

11.14.6 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 asks the operating system to return a 1 if TRUE or 0 if FALSE in int_var indicating whether or not the last PASS command executed successfully.

11.14.7 ASK | SET SYSTEM: PROCESS

FORMAT:

```
ASK SYSTEM: PROCESS str_var
SET SYSTEM: PROCESS str_expr
```

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.

11.14.8 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: TEST
```

SUMMARY:

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

11.14.9 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.

11.14.10 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: SUZANNE
```

SUMMARY:

ASK SYSTEM: USER returns the operating system name or ID for the user. Under VMS this is the user name.

11.15 ASK | SET WINDOW

ASK WINDOW and SET WINDOW have many options. The options are described in the following sections.

11.15.1 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
50  PRINT X$
60  END

RNH
   Mary had a
   little lamb
Mary had a
          little lamb


10  CLEAR
20  X$ = 'Mary had a' + chr$(10) + 'little lamb'
30  SET WINDOW AREA 6, 5, 7, 15: DATA X$
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.

11.15.2 ASK WINDOW: COLUMN

FORMAT:

```
ASK WINDOW: COLUMN int_var
SET WINDOW: COLUMN int_expr
```

EXAMPLES:

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

RNH
Cursor is at column 10


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

RNH
   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.

11.15.3 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: MYNAME$
      IF _HELP THEN
        ASK WINDOW: CURRENT OLD_W$
        CLEAR AREA BOX: 1, 5, 10, 50
        PRINT AT 3, 10, REVERSE: 'This is some help'
        DELAY
        SET WINDOW: CURRENT OLD_W$
        REPEAT DO
      END IF
30  END DO
```

SUMMARY:

ASK WINDOW: CURRENT saves the screen window image.

SET WINDOW: CURRENT restores the screen window image.

11.15.4 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.

11.15.5 ASK | SET WINDOW: KEYMAP

FORMAT:

```
ASK WINDOW: KEYMAP str_var
SET WINDOW: KEYMAP str_expr
```

EXAMPLE:

```
10  PRINT 'Save the current keymap, reset keymap to default value.'
    ASK WINDOW: KEYMAP old_keymap$
    SET WINDOW: KEYMAP ''

    PRINT 'Changing DOWN key to be the EXIT key'
    SET WINDOW KEYSTROKE 'DOWN': VALUE '_EXIT'
    LINE INPUT 'Press the DOWN key': DOWN$

    PRINT 'Changing dollar sign key to *'
    SET WINDOW KEYSTROKE '$': VALUE '*'
    LINE INPUT 'Press the dollar sign key, then RETURN': E$

    PRINT 'Restore saved keymap'
    SET WINDOW: KEYMAP old_keymap$
    LINE INPUT 'Press the DOWN key': DOWN$
    LINE INPUT 'Press the dollar sign key, then RETURN' :E$
20  END

RNH
Save the current keymap, reset keymap to default value.
Changing DOWN key to be the EXIT key
Press the DOWN key? EXIT
Changing dollar sign key to *
Press the dollar sign key, then RETURN? *
Restore saved keymap
Press the DOWN key?
Press the dollar sign key, then RETURN? $

INTOUCH
```

PURPOSE:

To allow a generalized routine to save the current keymap, change the meaning of keys and then restore the original keymap when done.

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.

11.15.6 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
```

PURPOSE:

SET WINDOW KEYSTROKE allows you to change the meaning of a keystroke within an INTOUCH program. This allows you to completely redefine the keyboard for a given application.

SUMMARY:

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 instance, the CTRL/Z key usually has a value of CHR$(26), and the concept of EXIT. The keystroke value and/or the keystroke concept can be changed. If changing both a value and a concept, separate the two with a comma. You can restore the original meaning of the key by using " ".

The following concepts are supported:

Table 11–1 Concepts—Supported

Concept name	Description
_EXIT	an EXIT key
_BACK	a BACK key
_HELP	a HELP key
_IGNORE	ignore this keystroke
_INVALID	beep when pressed
_TERMINATOR	keystroke is a line terminator

11.15.7 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.

11.15.8 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'
20  END

RNH
.......       (type STA)
showing status
...

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.

11.16 ASK | SET ZONEWIDTH

FORMAT:

```
ASK ZONEWIDTH num_var
SET ZONEWIDTH num_expr
```

EXAMPLES:

```
10 ASK ZONEWIDTH X
   PRINT 'The current print zone width is'; X
20 END

RNH
The current print zone width is 20


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

RNH
1                   2                   3
1         2         3
```

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 at which to set the device's print zones.

12 Loops, Conditionals, Chaining, Branching

12.1 Loops

A loop is a place where code is repeated. There are two types of loops. FOR loops execute a block of code a specific number of times. DO loops execute a block of code either: infinitely, until a specified condition is met, or while a specified condition remains true.

12.1.1 FOR

FORMAT:

```
FOR num_var = num_expr1 TO num_expr2 [STEP num_expr]
        ---
        ---  block of code
        ---
NEXT num_var
```

EXAMPLE:

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

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. STEP is used to change the increment of the loop. The default STEP is one.

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
```

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

SUMMARY:

REPEAT FOR unconditionally repeats all or part of a FOR loop. You might use REPEAT FOR if you want to repeat part of a FOR NEXT loop and you don't 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 10
      PRINT I
      INPUT 'Your name, please': NAME$
      IF NAME$ = 'Chuck' THEN ITERATE FOR
      PRINT 'Hello, '; NAME$
   NEXT I
20 END
```

SUMMARY:

Use ITERATE FOR to skip part of a FOR loop. ITERATE FOR transfers control to the NEXT statement.

12.1.5 DO

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
```

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. WHILE causes the loop to be executed while an expression remains true. UNTIL causes the loop to be executed until an expression becomes true.

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
```

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.

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
```

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
```

SUMMARY:

Use ITERATE DO to skip part of a DO loop. ITERATE DO transfers control to the LOOP statement.

12.2 Conditionals

12.2.1 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 'Enter a number': NUMBER
20 IF NUMBER < 0 THEN
     PRINT 'A negative number'
   ELSEIF NUMBER = 0 THEN
     PRINT 'Zero'
   ELSE
     PRINT 'A positive number'
   END IF
30 END
```

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

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, del, exit)': PRO$
     PRO$=UCASE$(PRO$)  !convert to uppercase
20   SELECT CASE PRO$
     CASE 'ADD'
       PRINT 'Adding...'
     CASE 'DEL', 'DELETE'
       PRINT 'Deleting...'
     CASE 'EXIT'
       PRINT 'Exiting...'
       EXIT DO
     CASE ELSE
       MESSAGE ERROR: 'Procedures are: add, del or exit'
       REPEAT DO
     END SELECT
   LOOP
30 END
```

SUMMARY:

Sometimes 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

12.3.1 GOTO

FORMAT:

```
GOTO target
```

EXAMPLE:

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

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

FORMAT:

```
ON int_expr GOTO target [,target...] [ELSE statement]
```

EXAMPLE:

```
10 ASK: INPUT 'Procedure (1=add, 2=del, 3=exit)': PRO
20 ON PRO GOTO ADD, DEL, DONE
   ADD: PRINT 'Adding...'
     GOTO ASK
   DEL: PRINT 'Deleting...'
     GOTO ASK
30 DONE: END
```

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

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
```

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.

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  ASK: INPUT 'Procedure (1=add, 2=del, 3=exit)': PRO
20  ON PRO GOSUB ADD, DEL, DONE
    GO TO 10
    ADD: PRINT 'Adding...'
         RETURN
    DEL: PRINT 'Deleting...'
         RETURN
30  DONE: END
```

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
```

SUMMARY:

Allow INTOUCH to determine a routine to be executed at run-time. DISPATCH looks at the content of a string expression (str_expr), searches for a routine with that name and dispatches to the routine.

12.6 ROUTINE/END ROUTINE

FORMAT:

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

EXAMPLE:

```
10      CLEAR
20      get_username
100     END

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

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 reexecuted.

12.6.3 Branching and Code Blocks

EXAMPLE:

```
10 DIM NAME$(4)
20 FOR I = 1 TO 4
     INPUT 'Your name, please': NAME$(I)
     IF NAME$(I) = 'NONE' THEN GOTO HELLO    <-- branches within block
     IF _EXIT THEN EXIT FOR                  <-- branches out of block
     HELLO:
       PRINT 'Hello!'
30 NEXT I
40 PRINT 'Finished'
50 END
```

SUMMARY:

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:

12.7 Chaining Programs

12.7.1 CHAIN

FORMAT:

```
CHAIN str_expr
```

EXAMPLE:

```
10 LINE INPUT 'Your name (last, first)': NAME$
20 OPEN #1: NAME 'STORAGE.TMP', ACCESS 'OUTPUT'
   PRINT #1: NAME$
   CLOSE #1
30 INPUT 'Add to CLIENT structure (Y/N)': REPLY$
   IF REPLY$= 'Y' THEN CHAIN "ADD"
40 END
```

SUMMARY:

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

The Next Generation Language  Status :

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

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

Paging File Usage Free  Reservable       Total
(pages):

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

INTOUCH
```

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, on OpenVMS, it passes the string to DCL. The operating system will respond to the string as it would if you entered it at the DCL $ prompt. When the system finishes, control returns to the INTOUCH program.

When you use the NOWAIT option with PASS, the operating system executes the passed command and immediately returns to INTOUCH without waiting for the passed command to finish.

EXAMPLE:

```
10  PASS NOWAIT : 'WAIT 00:01:00'
20  PASS NOWAIT : 'WAIT 00:01:00'
30  PASS NOWAIT : 'SHOW PROCESS/SUB'
40  END

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

Processes in this tree:

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

The PASS NORETURN statement passes a command to the operating system but does not return to INTOUCH.

EXAMPLE:

```
10  PRINT 'B E F O R E'
20  DELAY 2
30  PASS NORETURN: "dir/col=1"
40  END

RNH
B E F O R E

Directory WORK:[DOC.INTOUCH]

RELEASE.NOTES;5
WOMBAT.EXE;1

Total of 2 files.
$
```

13 Files

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

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 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.

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.

- 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 temporary holding files for listings, etc.

  Example:

```
10  OPEN #12: UNIQUE, ACCESS OUTPUT
20  ASK #12: NAME X$
30  PRINT X$
40  CLOSE #12
50  END

RNH
SYS$SCRATCH:TMP_B90001.TMP
```

13.2 INPUT

FORMAT:

```
INPUT #chnl_num: var, var...
```

EXAMPLE:

```
10 OPEN #1: NAME 'LEDGER.LIS', ACCESS INPUT
20 INPUT #1: LINE$
   PRINT LINE$
30 CLOSE #1
40 END
```

SUMMARY:

INPUT gets data from a file so you can use it in your program. Channel number is the channel number associated with the file in the OPEN statement.

13.3 LINE INPUT

FORMAT:

```
LINE INPUT #chnl_num [, EOF num_var]: str_var, str_var...
```

EXAMPLE:

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

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-file has been reached. This eliminates the need for an error handler when reading files.

EXAMPLE:

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

13.4 ASK #chnl_num:

FORMAT:

```
ASK #chnl_num: [NAME str_var][, ZONEWIDTH num_var] [, MARGIN num_var]
               [, CURRENT str_var]
```

EXAMPLE:

```
10 OPEN #3: NAME "STORAGE.DAT", ACCESS OUTPUT
20 ASK #3: ZONEWIDTH X
   PRINT 'The current print zone width is'; X
30 CLOSE #3
40 END
RNH
The current print zone width is 20
```

SUMMARY:

ASK returns a characteristic of a device and stores the value in a numeric 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 a value into the str_var. For example:

  Example:

```
10  OUT_CH = 12
20  OPEN #OUT_CH: NAME 'SYS$SCRATCH:MINUTES.LIS', &
     ACCESS OUTPUT
30  ASK #OUT_CH: NAME X$
40  PRINT X$
50  CLOSE #OUT_CH
60  END

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

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

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.

13.6 CLOSE

FORMAT:

```
CLOSE # chnl_num
```

EXAMPLE:

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

SUMMARY:

CLOSE closes a file. Channel number is the channel number you associate with the file in the OPEN statement.

13.7 PRINT

FORMAT:

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

EXAMPLE:

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

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.

13.8 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
```

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

SUMMARY:

CAUSE causes the exception specified to occur. Exception_number is an integer expression which specifies the number of the exception to generate.

14.2 WHEN EXCEPTION IN

FORMAT:

```
WHEN EXCEPTION IN
   ---
   ---  protected block
   ---
USE
   ---
   ---  handler
   ---
END WHEN
```

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$
20 WHEN EXCEPTION IN
     INPUT 'How old are you': AGE
   USE
     MESSAGE ERROR: 'Not a valid age'
     RETRY
   END WHEN
30 PRINT
   PRINT NAME$; ' is'; AGE
40 END
```

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.

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
20 WHEN EXCEPTION USE FIX_AVERAGE
     AVERAGE=TSALES/NSALES
   END WHEN
30 PRINT 'The average is:'; AVERAGE
40 HANDLER FIX_AVERAGE
     MESSAGE 'Can't divide by zero'
     AVERAGE = 0
     CONTINUE
   END HANDLER
45 PRINT 'Average is'; AVERAGE
50 END
```

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.

14.4 The Exception Handling Statements

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
     MESSAGE ERROR: 'Oops...'
     DELAY 1
     EXIT HANDLER
   END HANDLER
30 END
```

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
```

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
```

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
```

SUMMARY:

RESUME resumes normal program execution at the line or label you specify. RESUME can only be used within a handler routine.

14.5 Calling Routines Written in Other Languages

14.5.1 LIBRARY

FORMAT:

```
LIBRARY "libr_name"
```

EXAMPLE:

```
10 LIBRARY 'BASRTL'
20 PRINT 'Waiting 5 seconds...'
30 CALL BAS$SLEEP(5% BY VALUE)
40 END
```

SUMMARY:

Use LIBRARY statements to specify the libraries from which you will CALL routines. The routines can be written in any VMS language (FORTRAN, BASIC, COBOL, etc.). Libr_name is the name of a library. The library may be one of the VMS supplied libraries, or a user-defined library. The library name must be given as a string constant in quotes. When you call a routine, each library specified is checked in turn for the routine.

14.5.2 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
```

SUMMARY:

Use CALL to call and execute library routines. 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. Arg is an argument you are passing to the routine. If more than one argument is passed, separate the arguments with commas. To pass an optional argument, include it in the argument list. If you do not want to pass an optional argument, type two commas side by side at the appropriate place in the argument list. Pass_mech refers to the mechanism by which values are passed to the routine. The pass mechanisms available are: BY REF (by reference), BY VALUE, and BY DESC (by descriptor).

15 Data in INTOUCH

15.1 The PROGRAM, END, STOP and HALT Statements

15.1.1 PROGRAM

FORMAT:

```
PROGRAM prog_name
```

EXAMPLE:

```
10 PROGRAM Display_Name
20 INPUT 'Your name, please': NAME$
   PRINT NAME$
30 END
```

SUMMARY:

PROGRAM is used to give the program name. Prog_name must meet the rules for variable names.

15.1.2 END

FORMAT:

```
END
```

EXAMPLE:

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

SUMMARY:

Use the END statement to indicate the end of a program. If you have EXTERNAL subprograms or functions, use END to mark the end of the main program unit. When INTOUCH executes the END statement, it writes all active output buffers, closes all files in the current program and ends program execution.

15.1.3 STOP

FORMAT:

```
STOP
```

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$
   INPUT 'How old are you': AGE
   IF AGE < 1 THEN
     PRINT 'Not a valid age'
     STOP
   END IF
20 PRINT NAME$; ' is'; AGE
30 END
```

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.

15.1.4 HALT

FORMAT:

```
HALT
```

EXAMPLE:

```
10 INPUT 'Your name, please': NAME$
   INPUT 'How old are you': AGE
   IF AGE < 1 THEN
     PRINT 'Not a valid age'
     HALT
   END IF
20 PRINT NAME$; ' is'; AGE
30 END
```

SUMMARY:

HALT stops program execution, but it does not close files or write active output buffers.

15.2 Labels

INTOUCH has line numbers and labels. Line numbers can be any numbers in the range of 1 to 99999.

Line numbers must be integers. Labels can be up to 31 characters in length and may consist of letters and numbers.

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

15.3 Multiple Statements

INTOUCH accepts multiple statement lines. There are two ways this can be done:

- Statements can be terminated by a line terminator. Any number of statements may be attached to a program line.

```
10 FOR EACH EMPLOYEE
     PRINT EMPLOYEE(LAST)
   NEXT EMPLOYEE
```

- Multiple statements can fit on a single line, delimited by "\". CHR$(10) also signals the same as a "\". A "\" terminating the line implies an "&". This feature was added for use with the EXECUTE statement and for debugging programs.

  Example:

```
FOR Z1 = 4. TO 7. STEP .5 \ PRINT Z1 \ NEXT Z1
 4
 4.5
 5
 5.5
 6
 6.5
 7

INTOUCH
```

INTOUCH supports the REM statement and the trailing "!" for placing comments in programs. INTOUCH ignores everything on a line, following the "!". This includes line continuation characters.

Statement continuation is denoted by an "&" at the end of a line with an optional "&" at the beginning of the next line. The following examples are equivalent:

```
10 FOR EACH EMPLOYEE             ! a sample line
     PRINT EMPLOYEE(LAST)
   NEXT EMPLOYEE

10 FOR EACH &                    ! a sample line
        EMPLOYEE
     PRINT EMPLOYEE(LAST)
   NEXT EMPLOYEE

10 FOR EACH &                    ! a sample line
   & EMPLOYEE
     PRINT EMPLOYEE(LAST)
   NEXT EMPLOYEE
```

15.4 Constants

INTOUCH has numeric constants. Integer constants may end with a trailing % sign. Only real constants may contain a decimal point or be given in E notation. Examples of valid constants are:

```
375%   123   10e5   .00000000000001   123.543E-23   3.14159
```

String constants are delimited by matching single or double quote marks, ' or ". To enclose the delimiter within the string, repeat it twice. For instance, the string:

```
Fred said, "Hello"
```

can be represented as either of the following:

```
"Fred said, ""Hello"""        'Fred said, "Hello"'
```

15.5 Identifiers

Identifiers may contain any number of characters. Identifier names must start with a letter, and can contain letters, digits, and underscores. Upper- and lower-case characters are considered equivalent.

Identifiers can be one of the following data types

15.5.1 Integer variables

May be explicitly declared, or expressed as an identifier followed by the "%" character. These are represented internally as 32 bit signed integers.

15.5.2 Real variables

May be explicitly declared, or expressed as an identifier without a trailing type character. These are represented internally as 64 bit D-FLOATs.

15.5.3 String variables

May be explicitly declared, or expressed as an identifier followed by the "$" character. These are represented as a contiguous collection of bytes.

15.5.4 Boolean Variables

Boolean variables (they are either TRUE or FALSE) can be expressed with a trailing "?". For example:

```
10  done? = false
    do
      input 'Ready': ans$
      if _exit then done? = true
      if _back then done? = true
    loop until done?
20  end
```

15.5.5 Scalars

May be integer, real, or strings.

15.5.6 Arrays

May be up to 32 dimensions. These may be integer, real, variable length strings, or maximum length strings. Subscripts are rounded by INT(X+0.5), where X is the subscript reference.

15.5.7 Structure

This facility is specific to the INTOUCH system.

15.5.8 Reserved Identifiers

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
```

Users can override INTOUCH supplied identifiers, function names, etc., by redefining their meanings.

The scope of an identifier is the program, external subprogram or external function in which it is defined. The scope of identifiers which refer to subprograms or functions will be the entire program.

The DECLARE statement can be used to declare identifiers as STRING, INTEGER or REAL. Once declared, the identifier does not need to be designated by a type symbol ($ or %).

15.6 Expressions

Numeric expressions are evaluated in an algebraic manner, similar to that of the language FORTRAN and BASIC. All infix operators are left associative, including exponentiation. Thus, A^B^C is the same as (A^B)^C.

Relational functions return a value of 1 for TRUE and 0 for FALSE. The IF statement checks the value for a 1 or 0 (TRUE or FALSE).

The operators are:

```
+   -   *   /   ^   **   =   <   >   <=   =<   >=   =>   <>   ><
```

The **boolean** operators are:

```
AND    OR    NOT    XOR    EQV    IMP
```

String Expressions

Strings may be postfixed with [x:y] to extract the substring which starts at character x and extends through character y of the string.

```
PRINT A$[5:6]
```

Substrings may be used on the left-hand side of an assignment statement. Example:

```
LET A$[5:6]='WXYZ'
```

You can use an ampersand "&" or a plus sign "+" to concatenate strings.

```
10 LET A$="MO"
   LET B$="TH"
20 PRINT A$ & B$ & 'ER'
```

```
RNH
MOTHER
```

15.7 Miscellaneous Statements

15.7.1 RANDOMIZE

FORMAT:

```
RANDOMIZE
```

EXAMPLE:

```
5  RANDOMIZE
10 X=RND(10)
   PRINT X
20 END
```

SUMMARY:

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

15.7.2 DELAY

FORMAT:

```
DELAY num_expr
```

EXAMPLE:

```
10  FOR X = 1 to 20
20  PRINT 'Hello ';
30  DELAY 1.5
40  NEXT X
50  DELAY
```

SUMMARY:

DELAY causes INTOUCH to pause for the specified number of seconds before continuing program execution. Numeric expression specifies the number of seconds to wait before continuing program execution.

A Built-in Functions

INTOUCH has numerous built-in functions:

A.1 System Functions

The following functions return information about the system and user actions:

_BACK

Returns a TRUE or FALSE value. TRUE if the backspace key was pressed at the last prompt.

_CHANNEL

Returns the next available channel number.

Example:

```
10  OUT_CH = _CHANNEL
    OPEN #OUT_CH : NAME 'sys$scratch:test.lis'
    PRINT #OUT_CH : 'This is a test'
    CLOSE #OUT_CH
20  END
```

_DEBUG

Returns a TRUE or FALSE value. TRUE if DEBUG is on. FALSE if DEBUG is off.

_EXIT

Returns a TRUE or FALSE value. TRUE if EXIT was entered at the last prompt.

_EXTRACTED

Tells how many records were extracted in the last extract.

Example:

```
10 OPEN STRUCTURE CL: NAME "TTI_RUN:CLIENT"
20 EXTRACT STRUCTURE CL
30   INCLUDE CL(STATE)='CA'
40 END EXTRACT
50 PRINT 'Number of California clients:'; _EXTRACTED
60 END
```

_HELP

Returns a TRUE or FALSE value. TRUE if HELP or a question mark (?) was entered at the last prompt.

_INTEGER

Sometimes when the user does a CALL, an integer value is returned as a result of the CALL. _INTEGER returns any resulting integer data.

Example:

```
10 LIBRARY 'LIBRTL'
20 CALL LIB$ICHAR('Q')
30 PRINT 'The integer value of "Q" is: ';_INTEGER
40 END
```

_REAL

Some routines return a real value as the result of the CALL. The _REAL function returns any resulting real data.

Example:

```
10 LIBRARY 'MTHRTL'
20 CALL MTH$ALOG(45.3)
30 PRINT 'The log is:';_REAL
40 END
```

_REPLY

Returns the user's reply to the last prompt. The reply is returned as a string.

_STATUS

Returns the value given to INTOUCH by the operating system, for the last operating system request. This function is useful when debugging system level errors. The _STATUS function is often used with the SYSTEXT$ function.

_STRING

Upon the completion of an INPUT MENU statement, the concept _STRING contains the menu path taken by the user when selecting the menu item. (i.e. "#2;#3" means the 3rd item of the 2nd submenu.)

_TERMINATOR

The _TERMINATOR function returns the name of the key that terminated the last INPUT statement. The values returned are:

UP	Up arrow
DOWN	Down arrow
RETURN	Return
FIND	Find (find line, word)
INS	Insert
SEL	Select
PREV	Prev Screen
NEXT	Next Screen
REM	Remove
HELP	Help
DO	Do

PF1, PF2, PF3, PF4

KPO, KP1, KP2, KP3, KP4, KP5, KP6, KP7, KP8, KP9, MIN, COM, ENT, PER

F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, HELP, DO, F17, F18, F19, F20

A.2 Other Functions

ABS(num_expr)

ABS returns the absolute value of a specified numeric expression.

ACOS(num_expr)

ACOS(x) returns the angle whose COS is x. The angle is returned in radians.

ANGLE(num_expr, num_expr)

Given X and Y coordinates, returns the angle from 0,0. The angle is returned in radians.

ASCII(str_expr)

The ASCII function returns the decimal ASCII value of a string's first character. It is returned as an integer.

ASIN(num_expr)

ASIN(x) returns the angle whose SIN is x. The angle is returned in radians.

ATN(num_expr)

ATN returns the angle, in radians, of a specified tangent.

CEIL(num_expr)

CEIL(x) returns the ceiling of x. That is, the smallest integer that is not less than x.

CHANGE$(str_expr1, str_expr2, str_expr3)

CHANGE$ changes specified characters in str_expr1. Str_expr1 is the source string. Str_expr2 contains the target characters, and str_expr3 specifies the substitution characters. CHANGE$ returns the changed string.

CHANGE$ searches for the target characters within the source string and replaces these characters with the substitution characters. The substitution characters are mapped onto the target characters.

Example:

```
10 LET A$ = 'bdbdbdbd'
20 LET B$ = 'b'
30 LET C$ = 'c'
40 LET CHANGED$ = CHANGE$(A$,B$,C$)
50 PRINT A$
60 PRINT CHANGED$
70 END

RNH
bdbdbdbd
cdcdcdcd
```

CHARSET$[(str_expr)]

Returns the character set specified. The optional string expression can be used to specify the character set to return. The available character sets are:

UCASE	all upper-case letters (A-Z)
LCASE	all lower-case letters (a-z)
CONTROL	all control characters (ASCII 0-31)
ASCII	the ASCII character set, in order (0-255)

ASCII is the default character set for CHARSET$.

Example:

```
10 LINE INPUT 'Enter your text': TEXT$
20 ! Change upper-case to lower-case
   CT$ = CHANGE$(TEXT$, &
        CHARSET$('UCASE'), &
        CHARSET$('LCASE'))
   PRINT 'Lower-case version is:'; CT$
30 END
```

CHR$(int_expr1[, int_expr2])

CHR$ returns a string with the specified ASCII value (int_expr1) repeated the specified number of times (int_expr2). If no count is specified, a default count of one is used.

CONVERT(str_expr)

Given a string containing a mapped integer, this function returns the integer value.

Example:

```
10  A$ = 'AB'
    B = CONVERT(A$)
20  END

RNH
16961
```

CONVERT$(int_expr1 [, int_expr2])

Given an integer (int_expr1) and an optional length (int_expr2), which defaults to four, this function returns a string mapping of the integer.

Example:

```
10  A$ = CONVERT$(16961)
    PRINT A$
20  END

RNH
AB
```

The CONVERT and CONVERT$ functions can be used in situations such as building segmented keys consisting of multiple data types.

COMPARE(str_expr1, str_expr2)

The COMPARE function compares two strings and returns a numeric value ranging from 0 (no match) to 100 (an exact match).

Example:

```
 10  options$ = 'LEFT,RIGHT,UP,DOWN'
     best = 0
     best$= ''
     INPUT 'Enter an option': usr_opt$
 20  FOR IDX = 1 TO ELEMENTS(options$)
       opt$ = ELEMENT$(options$, IDX)
       score = COMPARE(opt$, usr_opt$)
       IF score > best THEN
         best = score
         best$= opt$
       END IF
     NEXT IDX
 50  SELECT CASE best
     CASE 0
       PRINT 'Unknown option: '; usr_opt$
     CASE 100
       PRINT 'Option okay, it was: '; usr_opt$
     CASE ELSE
       PRINT 'Misspelled option: '; usr_opt$
       PRINT USING 'Did you mean ? (## percent)': best$, best
     END SELECT
100  END

RNH
Enter an option? dwn
Misspelled option: dwn
Did you mean DOWN (92 percent)
```

COS(num_expr)

COS returns the cosine of an angle the user specifies in radians.

COSH(num_expr)

COSH returns the hyperbolic cosine of a passed real number.

COT(num_expr)

COT(X) is equivalent to 1/TAN(X).

CPAD$(text_str, size [, pad_str])

CPAD$ returns a new string, padded on the left and on the right with pad characters. Text_str is the string to be centered, size is the size of the new string. The default pad character is a space.

Example:

```
10 PRINT CPAD$('123', 9, '0')
20 END

RNH
000123000
```

CSC(num_expr)

CSC(x) is the cosecant of X. It's a shorthand for 1/SIN(x).

DATE

Returns today's date in YYDDD format.

DATE$[(int_expr1, [int_expr2])]

The DATE$ function returns the date in image format. Int_expr1 is a given Julian day number, the default is today's date. Int_expr2 indicates the desired output format for the date. The Julian day—the number of days since January 1, 1600.

Table A–1 DATE$ Function - Integer Values

Value (int_expr2)	Output Date Format
0	YYYYMMDD format
1	MMDDYYYY format
2	DDMMYYYY format
3	dd-Mon-yyyy format
4	Month dd, yyyy format

```
PRINT DATE$(123456) gives you

    19380104
    |   | |
   year mo day

PRINT DATE$ gives today's date

   19910417
   |   | |
  year mo day

PRINT DATE$(123456,1) gives you

    01041938
    | |   |
   mo day year
```

```
PRINT DATE$(123456,2) gives you

    04011938
    | |   |
  day mo  year

PRINT DATE$(123456,3) gives you

    04-Jan-1938

PRINT DATE$(123456,4) gives you

    January 4, 1938
```

To print today's date in a suitable format for letter heading, you can use:

```
PRINT DATE$(DAYS(DATE$),4) gives you

    April 17, 1991
```

DAYS(str_expr [, int_num])

Given a date in CCYYMMDD or YYMMDD format, returns the number of days since January 1, 1600 (this date is day 1). This number is called the Julian day.

Example:

```
10 PRINT DAYS('19930403')
20 PRINT DAYS('930403')
30 END

RNH
 143634
 143634
```

Int_num indicates the desired input format for the date. The default input format is zero. If the century is not included, it assumes 1900 as the century.

Table A–2 DAYS Function - Integer Values

Value (int_num)	Input Date Format
0	CCYYMMDD or YYMMDD
1	MMDDCCYY or MMDDYY
2	DDMMCCYY or DDMMYY
3	DD-Mon-CCYY or DD-Mon-YY
4	Month DD, CCYY

Example:

```
10 PRINT DAYS('19930403',0)
   PRINT DAYS('04031993',1)
   PRINT DAYS('03041993',2)
   PRINT DAYS('03-Apr-1993',3)
   PRINT DAYS('April 3, 1993',4)
20 END

RNH
 143634
 143634
 143634
 143634
 143634
```

DAY$[(int_expr)]

Given an integer expression specifying the number of days since January 1, 1600, DAY$ returns the day of the week. If no integer expression is given, DAY$ returns the day of the week for today's date. The day is returned as a string expression (Friday, Saturday, etc.).

DECODE(str_expr, int_expr)

Given the string representation of a number and the base that the value is in (int_expr), INTOUCH returns the value in base 10. The number is returned as a real number.

DEG(num_expr)

Given an angle in radians, returns the number of degrees.

DIV0(num_expr1, num_expr2)

The DIV0 function divides num_expr1 by num_expr2. If num_expr2 (divisor) is 0, 0 is returned.

Example:

```
10 PRINT DIV0(0.8, 0.000004)
   PRINT DIV0(0.8, 0.0)
   PRINT DIV0(6, 3)
   PRINT DIV0(6, 0)
20 END

RNH
 200000
 0
 2
 0
```

DTYPE(expr)

The DTYPE function returns as an integer value the data type of an expression or dynamic variable: 1 = string, 2 = integer, 3 = real.

Example:

```
10  DECLARE DYNAMIC X
20  X = 45.6
30  PRINT DTYPE(X)
40  END

RNH
3
```

EDIT$(str_expr,int_expr)

EDIT$ performs one or more editing operations, depending on the value of the integer expression, on the supplied string argument. The integer expression is one of the integers below, or a sum of integers below for the desired edit functions:

Table A–3 EDIT$ Function - Operation Values

Value	Edit Operation
1	Trim parity bits.
2	Discard all spaces and tabs.
4	Discard characters: CR, LF, FF, ESC, RUBOUT, and NULL.
8	Discard leading spaces and tabs.
16	Reduce spaces and tabs to One space.
32	Convert lower case to upper case.
64	Convert "[" to "(" and "]" to ")".
128	Discard trailing spaces and tabs.
256	Do not alter characters inside quotes.

ELEMENTS(str_expr1 [, str_expr2])

Returns the number of elements in a string expression which contains a list of elements. Str_expr1 is the string containing the list of elements. Str_expr2 is the separator. A comma is the default separator.

- Leading and trailing spaces are removed from the text.
- Quoted data is treated as one element.

ELEMENT$(str_expr1, num_expr [, str_expr2])

ELEMENT$ returns the element from str_expr1 which is specified by the num_expr. Str_expr1 contains a set of elements with separators between them. The default separator is a comma:

Example:

```
10 LET A$ = ELEMENT$('ADD,DEL,EXIT',2)
20 PRINT A$
30 END

RNH
DEL
```

You may specify a separator other than the comma with str_expr2.

Example:

```
10 LET SENTENCE$ = 'This is a test.'
20 LET A$ = ELEMENT$(SENTENCE$,2,' ')
30 PRINT A$

RNH
is
```

More than one separator in a row returns a null for the corresponding element.

Example:

```
10 LET sentence$ = 'This,, is, a, test'
   PRINT ELEMENT$ (sentence$, 2)
20 END

RNH
```

ENCODE$(num_expr, num_int)

Returns a string containing a number converted to the base specified. Num_expr is the value to convert. Num_int is the base to convert. For instance, '2' indicates binary, etc.

EPS(num_expr)

INTOUCH carries 15 digits of precision. The EPS(number) returns the smallest value, which when added to "number" yields the next number.

EVAL(str_expr)

This function evaluates the expression described in str_expr, and returns the result. If the variable being assigned the result is dynamic, the function puts the result in whatever data type the expression result was in. If the variable is NOT dynamic, and the result is the wrong data type, a "Data type mismatch" error is returned.

Example:

```
10  LINE INPUT 'Enter an expression': A$
    PRINT 'The result is '; EVAL(A$)
20  END

RNH
Enter an expression?  5 + 4
The result is  9
```

EXLABEL$

EXLABEL$ returns the label and line number executing when the last exception occurred, e.g., FRED.4

EXLINE

EXLINE returns the line number executing when the last exception occurred.

EXP(num_expr)

EXP function returns the value of the mathematical constant, "e", raised to a specified power.

EXTEXT$(int_expr)

EXTEXT$ returns explanatory text associated with a specified exception number.

EXTYPE

EXTYPE returns the number of the last exception that occurred. It is returned as an integer.

FALSE

Returns the constant 0. It is returned as an integer.

FILESPEC$(str_expr1 [, str_expr2 [, str_expr3]])

FILESPEC$ parses a file specification and returns either a full file specification or specific file specification fields.

Str_expr1 is the file specification to be parsed. If no file specification is given, the device and directory you are currently running from are returned.

Str_expr2 is a list of field names, separated by commas, which are to be returned. The field names are:

DEVICE	device name
DIRECTORY	directory name
NAME	file name
TYPE	type or extension name
VERSION	file version number
LOCATION	device and directory names
ALL or ""	full file specification

Str_expr3 is the default file specification. This parameter is optional.

FILESPEC$ can be used in various formats.

Example:

```
10 PRINT FILESPEC$('x.y', 'ALL')
   PRINT FILESPEC$('', 'ALL')
20 END

RNH
USER:[FRED]X.Y;
USER:[FRED].;
```

Example:

```
10 x$ = 'TTI_RUN:CLIENT'
   PRINT FILESPEC$(x$, 'ALL', '.dat')
20 END

RNH
STORAGE:[INTOUCH]CLIENT.DAT;
```

Example:

```
10 PRINT FILESPEC$('tti_run:client', 'ALL', '.dat')
20 PRINT FILESPEC$('tti_run:client', 'LOCATION')
   PRINT FILESPEC$('tti_run:client', 'LOCATION, NAME')
   PRINT FILESPEC$('tti_run:client.dat;1')
30 END

RNH
STORAGE:[INTOUCH]CLIENT.DAT;
STORAGE:[INTOUCH]
STORAGE:[INTOUCH]CLIENT
STORAGE:[INTOUCH]CLIENT.DAT;1
```

FINDFILE$(str_expr [, int_expr])

Given a file name to find, returns the complete file specification of the first file found that matches the name given. If no file is found, the function returns a null string.

str_expr	The name of the file to search for. This can be just part of the full file specification.
int_expr	Which file specification to return if multiple files are found. This parameter is optional. The default is to return the first file found.
Result	The complete file specification of the file found.

Example:

```
10  PRINT findfile$('TTI_RUN:*.INT')

RNH
DUA0:[INTOUCH]ALIENS.INT;5
```

Example:

```
10 DO
     LINE INPUT 'File.Spec': &
     SPEC$
     IF _EXIT THEN EXIT DO
     FOR I = 1 TO 9999
       FILE$ = FINDFILE$(SPEC$,I)
       IF FILE$ = '' THEN EXIT FOR
       PRINT FILE$
     NEXT I
   LOOP

RNH
File specification? TTI_RUN:CLIENT.*
DUA0:[INTOUCH]CLIENT.DAT;7
DUA0:[INTOUCH]CLIENT.DEF;3
DUA0:[INTOUCH]CLIENT.INT;1
DUA0:[INTOUCH]CLIENT.SAV;1
DUA0:[INTOUCH]CLIENT.STR;16
```

FORMAT$(expr, str_expr)

Given an expression and a format, returns the result of the expression in the format indicated.

The '@' format character causes the character not to be translated by the formatter. The '<' and '>'are treated like a '@' character. You can justify a character string, but avoid zero suppression and zero insertion.

The FORMAT$ function takes an expression of any data type for the data to format (the first argument), including string expressions.

Example:

```
10  Z$ = FORMAT$("1234567", "###~-####")
    PRINT 'Phone number: '; Z$
20  END

RNH
Phone number: 123-4567
```

The FORMAT$ function returns all asterisks "*" in the case of overflow.

Example:

```
10 z$ = FORMAT$(12.23,"#.##")
   PRINT z$
20 END

RNH
****
```

FORMAT$( ) returns the same string data as given by the following:

```
PRINT USING str_expr: expr
```

FP(num_expr)

Given a number, returns the fractional part of the number.

FULLTIME$[(float_expr, [int_var])]

Given the number of seconds since the INTOUCH base date, this function returns the date and time in one of the formats given below.

Where float_expr is the number of seconds since the INTOUCH base date. The default is the current date and time. January 1, 1600 00:00:00 is considered the second 0.

Table A–4 FULLTIME$ Function - Integer Values

Value (int_var)	Output Data Format
0	CCYYMDD HHMMSS
1	MMDDCCYY HHMMSS
2	DDMMCCYY HHMMSS
3	DD-Mon-CCYY HH:MM:SS
4	Month DD, CCYY HH:MM:SS

```
10  PRINT FULLTIME$
20  SEC = SECONDS('19910621 115042')
30  PRINT FULLTIME$(SEC, 0)
40  PRINT FULLTIME$(SEC, 1)
50  PRINT FULLTIME$(SEC, 2)
60  PRINT FULLTIME$(SEC, 3)
70  PRINT FULLTIME$(SEC, 4)
80  END

RNH

19910621 123756
19910621 115042
06211991 115042
21061991 115042
21-Jun-1991 11:50:42
June 21, 1991 11:50:42
```

HASH$(str_expr1 [, str_expr2 [, int_expr]])

This function changes the plain text in str_expr1 into a hashed eight-byte string value. It can be used to develop one-way hashed passwords. The optional text in str_expr2 and optional int_expr can be used to further make the hashed value unique.

```
10  PASSWORD$ = HASH$('TRUTH')
    INPUT 'Password': PWD$
    IF  HASH$(PWD$) = PASSWORD$  THEN
      PRINT 'That was the correct password.'
    ELSE
      PRINT 'That was not the correct password.'
    END IF
20  END

RNH
Password?  MONEY
That was not the correct password.
```

INT(num_expr)

INT returns the whole portion of a real number as a real number.

INTEGER(num_expr)

INTEGER changes any numeric expression into an integer value and assigns the integer value to the variable specified.

IP(num_expr)

IP truncates the value of a real number at the decimal point and returns the integer portion.

ITEM (str_expr1, str_expr2)

The ITEM function returns the number of the first item that matches the whole or partial item name given. A negative number is returned if more than one item matches.

Examples:

```
10  Z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'MOD')
    PRINT Z
20  END

RNH
4


10  Z = ITEM('ADD,EXIT,EXTRACT,MODIFY', 'EX')
    PRINT Z
20  END

RNH
-2
```

LBOUND(array_name [,int_expr])

Given an array and a dimension number, returns the low bound for that dimension. The default dimension is 1.

LCASE$(str_expr)

LCASE returns a string expression with all letters in lower case.

LEFT[$](str_expr, int_expr)

LEFT$ returns the leftmost **nn** characters from a string. Int_expr is the last character position to be included in the resulting string.

LEN(str_expr)

LEN returns the length of a string. It returns an integer.

LOG(num_expr)

LOG returns the natural logarithm of a specified number.

LOG2(num_expr)

LOG2 returns a number's base 2 logarithm.

LOG10(num_expr)

LOG10 returns a number's common logarithm.

LPAD$(text_str, size [, pad_str])

LPAD$ pads a string on the left with pad characters. The default pad character is a space.

Example:

```
10 PRINT LPAD$('123', 6, '0')
20 END
RNH

000123
```

LTRIM$(str_expr)

Removes all leading spaces (those on the left side of the string).

MATCH(str_expr1, str_expr2)

Str_expr1 contains a list of elements separated by commas. Str_expr2 contains a string. MATCH compares str_expr2 with each of the elements in str_expr1 and gives the number of the element that matches.

Quoted data is treated as one element; the quotes are not removed.

```
10 INPUT &
     'Which procedure (ADD,DEL,QUIT)': &
     PRO$
20 LET X = MATCH('ADD,DEL,QUIT', PRO$)
30 ON X GOSUB 50, 70, 90
40 GOTO 10
50 PRINT 'Adding...'
60 RETURN
70 PRINT 'Deleting...'
80 RETURN
90 END
```

MAX(num_expr, num_expr)

MAX(x,y) returns the larger of the two values x and y.

MAXLEN(str_var)

Returns the maximum number of characters that a string variable can contain. Since all string variables are variable length, with a maximum of 65535, this function always returns 65535.

MAXNUM

Returns the largest number available in this implementation of INTOUCH.

MAXSIZE(array_name)

Returns the total number of elements that can be contained in an array.

MID[$](str_expr, int_expr1 [,int_expr2])

MID$ or MID returns a substring from the middle characters of a specified string, leaving the string unchanged. Int_expr1 is the starting position of the substring, and int_expr2 is the length of the substring. MID$(str_expr,int_expr1) will return the rest of the string.

Example:

```
10  a$ = "beginmiddleend"
    middle$ = MID$(a$,6,6)
    end$ = MID$(a$,6)
    PRINT middle$, end$
20  END

RNH
middle     middleend
```

MIN(num_expr1, num_expr2)

MIN(x,y) returns the lesser of the values x and y.

MOD(num_expr1, num_expr2)

Gives the remainder of one number divided by another.

ORD(str_expr)

Given the ASCII name of a character, returns the location of that character in the ASCII character table. It returns an integer number.

ORDNAME$(int_expr)

Given the location of a character in the ASCII character table, returns the name of that character.

PARSE$(str_expr)

PARSE$ splits a string into tokens and returns each token separated by a space. Letters are upper-cased, except within quotes. Tail comments are ignored.

Example:

```
a$ = 'company$ = abc$ +"and sons" !rnn'
PRINT PARSE$(a$)

RNH
COMPANY$ = ABC$ + "and sons"
```

PATTERN(str_expr1, str_expr2)

Matches any characters in text (str_expr1) with the pattern (str_expr2).

Str_expr1 is the text to search for a given pattern and str_expr2 is the pattern being searched for.

Returns the location of the first character in the text that contains the pattern. If the characters cannot be found, returns zero.

PATTERN Options and Examples

Pattern options can be mixed and matched with unlimited complexity.

?

Matches any character in the text.

```
10 IF PATTERN('The quick brown fox',  &
     'f?x') > 0 THEN
      PRINT 'Found'
   END IF
20 END

RNH
Found
```

*

Matches zero or more of a character that precedes the asterisk.

```
10 IF PATTERN('aaa   03/26/92', 'a* *03/26/92')>0 THEN
      PRINT 'The date is found'
   END IF
20 END

RNH
The date is found
```

{}

Used to define a group of characters. The characters in the enclosed group can be ranges such as {a-z} or individual characters such as {AQX}.

```
10 text$ = 'A1N5V7N0'
   rule_str$ = '{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}'
   IF PATTERN(text$, rule_str$)>0 THEN
      PRINT "Driver's licence is correct"
   END IF
20 END

RNH
Driver's licence is correct
```

{^}

Looks for characters that are NOT {^A-Z}

The result of line 16 below shows the difference between using '?' and {^}.

```
10 PRINT PATTERN('Mary J. Smith','{^Mar}')
15 PRINT PATTERN('Mary J. Smith','J. {^S}')
16 PRINT PATTERN('Mary J. Smith','J. S?')
20 END

RNH
4
0
6
```

~

The '~' (quote) character looks for a pattern of text that IS an * (stands for itself).

```
10 text$='$4,670.00'
   IF PATTERN(text$,'$~4,670.00') > 0 THEN
     PRINT 'Your text is correct'
   END IF
20 END

RNH
Your text is correct
```

{<cc|ccc|c>}

Looks for text in str_expr1 that matches the enclosed groups.

```
10 text$='The area code is 619'
   IF PATTERN(text$,'is {<619|714|916>}') > 0 THEN
      PRINT 'Your area code is on the list'
   END IF
20 END

RNH
Your area code is on the list
```

PI

Returns the value 3.141592653589793.

PIECES(str_expr1 [,str_expr2])

and

PIECE$(str_expr1,num_expr[,str_expr2])

PIECE$ returns an element from str_expr1 specified by num_expr. Str_expr1 contains a list of elements. The separator can be indicated by str_expr2. The default separator is a carriage-return line-feed pair.

These two functions are similar to the ELEMENTS( ) and ELEMENT$( ) functions except that:

- no leading or trailing spaces are removed from the text.
- quoted data is not skipped over.
- the default separator is the two-character sequence. CHR$(13)+CHR$(10), the CR/LF sequence used by WRAP$( ).

Example:

```
10 CLEAR
15 MESSAGE 'Press GOLD/F when done'
20 LINE INPUT AREA 5, 10, 8, 60: TEXT$
30 PRINT AT 10, 1: 'Rewrapped text'
40 WT$ = WRAP$(TEXT$, 1, 30)
50 PRINT 'Number of lines: '; PIECES(wt$)
60 PRINT WT$
70 PRINT
80 PRINT 'First line was: '; PIECE$(wt$, 1)
100     END
```

POS(str_expr1, str_expr2[, int_expr])

POS searches for a substring within a string. It returns the substring's starting character position. Str-exp1 is the string to be searched, str-exp2 is the substring to locate, and int-exp is the OPTIONAL character position at which to begin the search. (The default is the start of the string.)

PRETTY$(str_expr)

PRETTY$ converts text so that the text displays on any terminal. Named control characters show up with their names. Other control characters show up as {X} where "X" is the letter to press or as {XX} where "XX" is the hexadecimal value of the character.

```
10  A$ = "Hello" + CHR$(5) + CHR$(161) + CHR$(7)
20  PRINT PRETTY$(A$)
30  END

RNH
Hello{^E}{A1}{bel}
```

QUOTE$(str_expr)

The QUOTE$ function encloses a string expression in double quotes. If the string expression is already enclosed in double quotes, QUOTE$ leaves it alone. If the string expression is already wrapped in single quotes, QUOTE$ replaces them with double quotes. Elements double quoted within the string expression are given another pair of double quotes.

Example:

```
10  DO
      CLEAR
      MESSAGE 'Enter a line of text to be quoted'
      PRINT 'Text:'
      INPUT '', LENGTH 30: line$
      IF  _BACK OR _EXIT  THEN EXIT DO
      IF  line$ = ''  THEN REPEAT DO
50    PRINT
      PRINT 'Quoted text using the QUOTE$ function...'
      PRINT quote$(line$)
      DELAY
    LOOP
100 END

RNH

Text:
? The little boy cried "wolf!"

Quoted text using the QUOTE$ function...
"The little boy cried ""wolf!"""
```

RAD(num_expr)

Given a measurement in degrees, returns the number of radians.

REAL(num_expr)

REAL changes any numeric expression into a real or floating-point value and assigns the real value to the variable specified.

Example:

```
10 INPUT 'Your age': AGE%
20 LET DECIMAL_AGE = REAL(AGE%)
30 PRINT 'Your number is'; DECIMAL_AGE
40 END
```

REMAINDER(num_expr1, num_expr2)

REMAINDER(x,y) returns the remainder when X is divided by Y. It differs subtly from MOD. MOD(-4,3) = 2 while REMAINDER(-4,3) = -1.

REPEAT$(str_expr, int_expr)

Creates a string composed of the specified string repeated the specified number of times.

REPLACE$(str_expr1, str_expr2 [,str_sep1][,str_sep2])

Searches for a list of patterns in the str_expr1 and replaces them with the output string from str_expr2. Returns the replaced string expression.

Str_expr1 is a list of patterns to search for.

Str_expr2 is the replacement list.

Str_sep1 is the optional separator for replacement items. The default is a comma.

Str_sep2 is the optional separator between the input and output text in items. Default is =.

Example:

```
10 text$='01-Mar-1981'
   PRINT REPLACE$(text$,'r=y 8=9',' ')
20 END

RNH
01-May-1991
```

RIGHT[$](str_expr, int_expr)

RIGHT$ returns the rightmost characters from a string. The int-exp is the character position of the last character to be included in the substring COUNTING FROM THE RIGHT.

Example:

```
10 ans$ = RIGHT$('Daniel', 2)
   PRINT 'rightmost char.= ', ans$
20 END

RNH
rightmost char.= el
```

RND

or

RND(num_expr)

RND returns a random number greater than or equal to zero and less than one. If a numeric expression (num_expr) is given, RND returns a whole number between one and the numeric expression.

ROUND(num_expr [, int_expr])

ROUND rounds a num_expr to the specified number of decimal places (int_expr). The default int_expr is 0.

RPAD$(text_str, size[,pad_str])

RPAD$ pads a string on the right with pad characters. The default pad character is a space.

Example:

```
10 PRINT RPAD$('123', 6, '0')
20 END

RNH
123000
```

RTRIM$(str_expr)

Returns a string without any trailing spaces (those on the right side).

SCAN(str_expr1, str_expr2 [, int_expr])

Scans str_expr1 for the characters in str_expr2 and returns the position at which str_expr2 begins. Int_expr specifies a character position at which the search is to begin.

The characters in str_expr2 need not appear contiguously in str_expr1.

Example:

```
10 LET A$ = 'Cameron Whitehorn'
20 LET B$ = 'amr Wtor'
30 LET POSITION = SCAN(A$,B$)
40 PRINT POSITION

RNH
2
```

SEC(num_expr)

Returns a secant of a given angle (1/COS(num_expr)). Num_expr is a passed angle.

SECONDS(str_expr)

Given a full-time string in CCYYMMDD HHMMSS, YYMMDD HHMMSS, HHMMSS or HHMM format, returns the number of seconds since the INTOUCH base date (January 1, 1600 00:00:00).

The number of seconds is returned as a floating point number.

Example:

```
10 Z  = SECONDS('19931201 103050')
   Z1 = SECONDS('931201 103050')
   Z2 = SECONDS('103050')
   Z3 = SECONDS('1030')
20 PRINT 'Seconds CYMDHMS ='; Z
   PRINT 'Seconds  YMDHMS ='; Z1
   PRINT 'Seconds     HMS ='; Z2
   PRINT 'Seconds     HM  ='; Z3
30 END

RNH
Seconds CYMDHMS = 12430837850
Seconds  YMDHMS = 12430837850
Seconds     HMS = 37850
Seconds     HM  = 37800
```

SEG$(str_expr, int_expr1, int_expr2)

Extracts the substring given a first and last character position.

SGN(num_expr)

Returns the sign of a number. It returns a +1 if the expression is positive, a -1 if the expression is negative, and 0 if the expression is zero.

SIN(num_expr)

SIN returns the sine of an angle specified in radians.

SINH(num_expr)

SINH(X) returns the hyperbolic sine X.

SIZE(array_name [,int_expr])

Returns the number of elements in one dimension of an array.

array-name	Array to examine
int-expr	Dimension to get the size of. The default dimension is 1.

SKIP(str_expr1 [, str_expr2] [, int_expr])

SKIP returns the position of the character following the last skipped character.

Str_expr1 is the text string to be searched.

Str_expr2 contains the list of characters which are to be skipped. If only one argument is given, SKIP will skip over spaces, tabs and nulls.

Int_expr contains the search start position. This parameter is optional.

Example:

```
10 a$ = '31415 hello'
   z = SKIP(a$, '1234567890 ')
20 PRINT mid(a$, z)
30 END

RNH
Hello
```

SORT$(str_expr1 [,str_expr2])

Sorts the elements from a str_expr1 in ASCII value order. Returns a list of the sorted elements.

Str_expr1 contains the list of elements to be sorted.

Str_expr2 is an optional separator. Default is a comma.

Example:

```
10 a$ = "code area is"
   a_sort$ =SORT$(a$," ")
   PRINT a_sort$
20 END

RNH
area code is
```

SPACE$(num_expr)

Returns the number of spaces indicated by num_expr.

SQR(num_expr)

SQR returns the square root of a number.

STR$(num_expr)

STR$ changes a number to a numeric string. The string that is created does not have any extra leading or trailing spaces.

SYSTEXT$

or

SYSTEXT$(int_expr)

SYSTEXT$ returns the text associated with the operating system status specified. If no int_expr is supplied, INTOUCH returns the text for the last system status.

TAB(int_expr)

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to the right to a specified column.

TAN(num_expr)

TAN returns the tangent of an angle that is specified in radians.

TANH(num_expr)

Returns the hyperbolic tangent of the numeric expression given.

TIME(int_expr)

The value returned by the TIME function depends on the value of int_expr.

If int_expr = 0, TIME returns the number of seconds since midnight.

If int_expr = 1, TIME returns the CPU time of the process in tenths of a second.

If int_expr = 2, TIME returns connect time of the current process in minutes.

TIME$

or

TIME$(num_expr)

If num_expr is NOT specified, TIME$ returns the current time of day in HH:MM:SS format.

Num_expr is the number of seconds since midnight. The result is returned in HH:MM format.

Example:

```
10 PRINT TIME$(1800)
   PRINT TIME$(54178)
20 END

RNH
00:30
15:02
```

TIME(5)

Returns the number of seconds since INTOUCH was invoked. This function can be used to time events to the nearest 100th/sec.

TRIM$(str_expr)

Returns the string specified stripped of any leading or trailing spaces and tabs.

Example:

```
10 LET A$ = '    HELLO    '
20 PRINT '*'; A$; '*'
30 LET STRIPPED$ = TRIM$(A$)
40 PRINT '*'; STRIPPED$; '*'

RNH
*    HELLO    *
*HELLO*
```

TRUE

Returns the constant 1. It is returned as an integer.

TRUNCATE(num_expr, int_expr)

Truncates a real number to a given number of decimal places.

UBOUND(array_name [, int_expr])

Given an array and a dimension number, returns the upper bound for that dimension. It returns an integer value. The default dimension is 1.

UCASE$(str_expr)

UCASE returns a string expression with all letters in upper case.

UNQUOTE$(str_expr)

The UNQUOTE$ function removes one set of quotes from a quoted string expression. If the string expression is not quoted, UNQUOTE$ leaves the string alone. UNQUOTE$ does not affect internally quoted elements.

Example:

```
10  DO
      CLEAR
      MESSAGE 'Enter a line of text to be unquoted'
      PRINT 'Text:'
      INPUT '', LENGTH 50: line$
      IF  _BACK OR _EXIT  THEN EXIT DO
      IF  line$ = ''  THEN REPEAT DO
50    PRINT
      PRINT 'Quotes removed using the UNQUOTE$ function...'
      PRINT UNQUOTE$(line$)
      DELAY
    LOOP
100 END

RNH

Text:
? "I will not take these 'things' for granted"

Quotes removed using the UNQUOTE$ function...
I will not take these 'things' for granted
```

VAL(num_str)

VAL returns the floating-point value of a numeric string.

VALID(text_str, rule_str)

VALID is used to validate user responses.

Text_str is the text to be validated.

Rule_str is the list of validation rules.

Multiple validation rules are separated by a semi-colon. If given characters are NOT between quotes, they are upper-cased. VALID returns TRUE or FALSE according to the following rules:

- *ALLOW*

```
ALLOW text1, text2, text3 [to text4]
```

ALLOW compares the 'text_str' with each element on the 'rule_str' list (defaults to CHARACTER) and returns TRUE if found.

For numeric comparison, "NUMBER" or "INTEGER" *must* be used, otherwise, VALID( ) will do a string-compare validation...left justified.

```
10  IF  VALID('ann', 'ALLOW ann, dan, tom') &
          THEN  PRINT 'true'
20  IF  NOT VALID('10',  'NUMBER; ALLOW 1 to 6; MAXLENGTH 2') &
          THEN PRINT 'false'
30  IF  VALID('12', 'NUMBER; ALLOW 1 TO 24,99') &
         THEN  PRINT 'true'
40 END

RNH
true
false
true
```

- *DISALLOW*

```
DISALLOW  text1, text2, text3 [to text4]
```

Returns TRUE if 'text_str' cannot be found on the 'rule_str' list.

```
10 IF VALID ("10","DISALLOW 01,03,05; MINLENGTH 2")&
     THEN PRINT "true"
20 END

RNH
true
```

- *MINLENGTH*

```
MINLENGTH  nn
```

Minimum of nn characters long.

- *MAXLENGTH*

```
MAXLENGTH   nn
```

Maximum of nn characters long.

- *LENGTH*

```
LENGTH nn
```

Exactly nn characters long.

- *CHARACTERS*

```
CHARACTERS   ccc
```

Returns TRUE if the given characters (letters or numbers) are on the 'rule_str' list.

- *NOCHARACTERS*

```
NOCHARACTERS ccc
```

Returns TRUE if the given characters are not on the 'rule_str' list.

- *LETTERS*

Returns TRUE if given 'text_str' consists only of A - Z, a - z or space.

- *LCASE*

Returns TRUE if given 'text_str' consists of lowercase letters.

- *UCASE*

Returns TRUE if given 'text_string' consists of uppercase characters.

- *DIGITS*

Returns TRUE if 'text_str' consists of the numbers 0 - 9.

- *DECIMALS*

Specifies the maximum number of decimal places.

```
DECIMALS nn
```

- *NUMBER*

Indicates that 'text_str' is numeric (validates that number).

- *INTEGER*

Returns TRUE if given integer number is 32 bits long.

- *INTEGER WORD*

Returns TRUE if given integer number is 16 bits long.

- *DATE*

Validates a given date in: YYMMDD or CCYYMMDD format.

DATE	YMD	YYMMDD format
DATE	DMY	DDMMYY format
DATE	MDY	MMDDYY format
DATE	MDCY	MMDDCCYY format

Note

For all dates, use 'LENGTH nn' to control whether a 6- or 8-character date is required.

- *DATE DMONY*

Checks if text_str is a legal date in DD-Mon-YY format and returns TRUE or FALSE.

```
10 PRINT VALID ("01-May-93", "DATE DMONY")
20 END

RNH
1
```

- *DATE DMONCY*

Checks if text_str is a legal date in DD-Mon-CCYY format and returns TRUE or FALSE.

```
10 PRINT VALID ("01-May-1993", "DATE DMONCY")
20 END

RNH
1
```

- *FULLTIME*

Checks if text_str is a legal date and time in CCYYMMDD HHMMSS or YYMMDD HHMMSS format and returns TRUE or FALSE.

```
10 PRINT VALID ("19931231 010101", "FULLTIME")
   PRINT VALID ("931231 010101", "FULLTIME")
20 END

RNH
1
1
```

- *CODE*

Checks for VALID INTOUCH code syntax.

```
10 a$ = 'PRINT "Hello, " name$'
   IF NOT VALID (a$,"CODE") THEN PRINT "false"
30 END
```

- *PATTERN*

Matches 'text_str' with PATTERN's 'rule_str'. For details see PATTERN$( ).

- *REQUIRED*

Returns TRUE if 'text_str' does not contain any characters or spaces.

- *YES/NO*

Returns TRUE if 'text_str' consists only of the following: YES, NO, Y or N.

- *VRULES*

Checks if a text_str is a legal set of validation rules, and returns TRUE or FALSE.

```
10 PRINT VALID("integer", "vrules")
20 END

RNH
1
```

- *PRINTMASK*

Checks the format of text_str and returns TRUE if it is legal or FALSE otherwise.

```
10 text_str$ = '##.##'
   PRINT VALID(text_str$, 'PRINTMASK')
20 END

RNH
1
```

- *EXPRESSION*

Validates a text_str$ and returns TRUE if it is a legal INTOUCH expression or FALSE otherwise.

```
10 text_str$ = "total = a% + 30"
   PRINT VALID (text_str$, "EXPRESSION")
   PRINT VALID ("##~-###", "EXPRESSION")
20 END

RNH
1
0
```

- *MENU*

Validates a text_str$ and returns TRUE if it is a legal menu description for INPUT MENU statement, or FALSE otherwise.

```
10 text_str$= '%multi,a,b,c'
   PRINT VALID (text_str$, 'MENU')
20 END

RNH
1
```

- *FILTER*

Filters the 'text_str' before the validation. The following filters are allowed:

LTRIM$

Remove leading spaces.

RTRIM$

Remove trailing spaces.

TRIM$

Remove both leading and trailing spaces.

UCASE$

Convert to uppercase characters.

LCASE$

Convert to lowercase characters.

REPLACE

```
old_t1=new_t1, old_t2=new_t2, ....
```

If 'text_str' contains a few 'old_t1' characters, all of them will be replaced by 'new_t1'.

REMOVE

Removes any characters from the 'text_str'.

CHANGE

Changes specified characters. For details see CHANGE$( ).

```
10 IF VALID("abcd10", &
     "FILTER REMOVE '10';LETTERS" THEN PRINT "true"
   IF VALID ("ab1cd1", "FILTER REPLACE '1'='e';LETTERS") &
     THEN PRINT "true"
   IF VALID (" 1234  ","FILTER TRIM; NUMBER") &
     THEN PRINT "true"
30 END
```

```
RNH
true
true
true
```

RESTORE

Restores data to pre-filtered state for further validations, can be used for multiple validation rules.

```
10  IF VALID ("123", &
          "FILTER REPLACE '1'='a';RESTORE;NUMBER") &
      THEN PRINT "true"
20  END

RNH
true
```

VALID returns an error if there is an invalid validation rule.

```
"Illegal validation rule" (-4021)
```

WRAP$(str_expr, int_expr1, int_expr2)

Returns a word-wrapped text string, given left and right margins. Each line of the string is separated with a CR/LF.

Where string_expr = text string to wrap, int_expr1 = left margin, int_expr2 = right margin.

XLATE$(str_expr1, str_expr2)

The XLATE$ function translates one string to another by referencing a table you supply. For example, the XLATE$ function can translate from EBCDIC to ASCII. The first str_expr is the string to be translated. The second str_expr is the translation table.

B Conventions Used in this Manual

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

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

Certain conventions are used to represent values which you must supply. These conventions are always shown in lower case. The conventions are listed below.

array	Array element.
block of code	An INTOUCH statement or a series of INTOUCH statements.
chnl	An I/O channel associated with a file.
chnl_num	An I/O channel associated with a file.
cond_expr	Conditional expression.
col	Column. Used in the TAB or AT option to indicate the column to print at.
const	Constant value.
expr	Expression. May be a variable, constant, array element, structure reference, etc. or any combination of the previous separated by operators.
field_expr	Field name expression.
filename	File name. (Described briefly below and in the Command Language and DCL User's Guide of the OpenVMS documentation set).
file_spec	File specification. (See the Command Language and DCL User's Guide of the OpenVMS documentation set).
func	Function.
handl	Exception handler. Pertaining to an exception handling routine.
handler	The block of code in an exception handler which is executed when an exception is generated.
int	Integer value.
int_expr	Integer expression.
int_var	Integer variable.
label	Alphanumeric statement label.
libr	Pertaining to libraries.
line_num	Program line number.
name	Name. Indicates the declaration of a name or the name of a program unit, such as a subprogram.
num	Numeric value. Num indicates that either a real or integer numeric value may be used.
num_expr	Numeric expression. Num_expr indicates that either a real or integer numeric value may be used.

Conventions Used in this Manual

num_var	Numeric variable.
param	An expression which is passed to a function, subprogram or other program unit. A parameter can be any valid INTOUCH expression.
protected block	The block of code protected by an exception handling routine.
real	Real number. Indicates that only a real number may be used.
row	Row. Used in the AT option to specify the row to print at.
str	String. Str indicates that only a string value may be used.
str_expr	String expression.
struc	An INTOUCH structure.
struc_name	Structure name.
struc_expr	Structure expression.
sub	An INTOUCH subprogram.
target	The target point of a branch statement. The target can be either a program line number, an alphanumeric label, or a ROUTINE statement.
uncond_expr	Unconditional expression. Uncond_expr indicates that only an unconditional expression can be used.
var	Variable. May be a simple variable, array element or a structure reference.
[ ]	Optional portions of a statement are enclosed in brackets. `INPUT [PROMPT str_expr] : var`
{ | }	Braces combined with a vertical bar indicate that one of the elements in the braces **must be chosen.** The elements are separated by the vertical bar. `EXIT {FOR | DO}`
[|]	Brackets combined with a vertical bar indicates that one of the options in the brackets can be used. The elements are separated by the vertical bar. Only one of the elements can be used. `DO [WHILE expr | UNTIL expr]`
. . .	An ellipsis indicates that the language element can be repeated. An ellipsis following an option enclosed in braces indicates that you can repeat the format of the whole option. `INPUT [PROMPT str_expr] : var, var...`
dashes	Three sets of dashes indicate a block of code.

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

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

.
.
.

C Making Libraries for Use with INTOUCH

To make libraries for use with INTOUCH, you will need to follow the steps described below:

1. Write the source code using an editor of your choice. The source code may 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". Enter the option UNIVERSAL to specify the modules (subprograms and functions) contained in your routine.

   ```
   UNIVERSAL=module [, module...]
   ```

   If your routine contains more than one module, separate multiple module names with commas. the routine described above has only one module:

   ```
   UNIVERSAL=MULT
   ```

4. Link the library.

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

   For the routine above, if your options file is called 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–20
? • A–20
{^} • A–21
{} • A–20
~ • A–21

A

ABORT • 5–3
ABS • A–3
ACCESS • 3–23
ACOS • A–3
ALPHA • 5–9
ANGLE • A–4
APPLICATION • 3–23
Application development tool • xvii
AREA
  Line Input • 8–3
AREA Option • 9–6
Arrays • 15–5
ASCII • A–4
ASIN • A–4
ASK ERRORS • 11–3
ASK KEYSTROKES • 11–4
ASK MARGIN • 11–5
ASK MESSAGELINE • 11–5
ASK RESPONSES • 11–7
ASK SEED • 11–8
ASK Statement • 11–1
ASK STRUCTURE • 3–21, 3–30
ASK STRUCTURE: ACCESS • 3–30
ASK STRUCTURE: CAPABILITY • 3–25
ASK STRUCTURE: CURRENT • 3–21
ASK STRUCTURE: ENGINE • 3–26
ASK STRUCTURE: EXTRACTED • 3–27
ASK STRUCTURE: FIELDS • 3–22
ASK STRUCTURE: ID • 3–27
ASK STRUCTURE: KEYS • 3–22
ASK STRUCTURE: POINTER • 3–28
ASK STRUCTURE: RECORD • 3–28
ASK STRUCTURE: RECORDSIZE • 3–29
ASK STRUCTURE: TIMEOUT • 3–30
ASK STRUCTURE FIELD: item • 3–22
ASK STRUCTURE struc_name: ACCESS • 3–30
ASK SYSTEM, LOGICAL • 11–10
ASK SYSTEM, PASS: SUCCESS • 11–12
ASK SYSTEM, SYMBOL • 11–13
ASK SYSTEM: COMMENT • 11–9
ASK SYSTEM: DIRECTORY • 11–9
ASK SYSTEM: MODE • 11–11
ASK SYSTEM: PARAMETER • 11–11
ASK SYSTEM: PROCESS • 11–12
ASK SYSTEM: RIGHTS • 11–13
ASK SYSTEM: USER • 11–14
ASK SYSTEM options • 11–8
ASK WINDOW: COLUMN • 11–15
ASK WINDOW: CURRENT • 11–16
ASK WINDOW: DATA • 11–17
ASK WINDOW: KEYMAP • 11–18
ASK WINDOW: ROW • 11–20
ASK WINDOW: TYPEAHEAD • 11–21
ASK WINDOW AREA • 11–14
ASK WINDOW options • 11–14
ASK ZONEWIDTH • 11–22
Assigning variables • 7–1
ATN • A–4
At Option • 9–1
%AT row, column • 8–12
Attributes
  CLEAR AREA • 9–6
  Drawing a box • 9–8
  INPUT • 8–8
ATTRIBUTES • 3–23
  Field • 1–8
%AUTODISPLAY • 8–13
%AUTOVBAR • 8–13
AXP systems • 5–9

B

_BACK • 8–4, A–1
BACKING UP • 8–4
%BAR • 8–12
BLOCK OF CODE • B–1
BOOLEAN OPERATORS • 15–6
BOOLEAN variables • 15–5
BOX OPTION • 9–8
Braces • B–2
BRANCHING
  Within code blocks • 12–13
BREAK • 4–6

BUILT-IN FUNCTIONS • A–1

C

CALL
  DESCRIPTION OF • 14–6
  PASSING ARGUMENTS WITH • 14–6
  See also the _REAL and _INTEGER functions • A–1
CALL SETUP • 1–1, 1–2
  SECURITY • 1–10
CANCEL ADD • 3–12
CANCEL EXTRACT • 3–7
CAUSE EXCEPTION • 14–1
CEIL • A–4
CHAIN • 12–13
CHANGE • A–35
CHANGE$ • A–4
CHANGEABLE • 3–23
_CHANNEL • A–1
CHARACTERS • A–31
CHARSET$ • A–5
CHR$ • A–5
CLASSIFICATION • 3–23
CLEAR • 9–6
  AREA Option • 9–6
CLOSE Statement • 13–5
CLOSE STRUCTURE • 3–4
CODE • A–33
CODE BLOCKS
  Transfer of control • 12–13
Column • 9–6
%COLUMNS • 8–13
COMMANDS
  EXIT • 5–1
  HELP • 5–1
Commas in PRINT statement • 9–5
COMPARE • A–6
COMPILE • 5–9
COMPILED
  Images • 5–9
COMPILED IMAGES • 5–9
  Running • 5–10
CONCEPTS
  Supported • 11–20
CONSTANTS • 15–4
CONTINUE • 14–4
Conventions
  Summary list • B–1
CONVERT • A–5
CONVERT$ • A–5
COS • A–6
COSH • A–6
COT • A–6
CPAD$ • A–7
Creating Unique File Names • 13–5
CSC • A–7
CSET • 7–2
CTRL/xxx • 8–15

D

DATA
  DESCRIPTION OF • 7–3
Database routines
  Specifications • xviii
Data Structure Manual • xvii
DATA STRUCTURES
  General Description • 3–1
DATA TYPE • 1–7
DATATYPE • 3–23
DATE • A–7, A–32
DATE$ • A–7
DATE DMONCY • A–32
DATE DMONY • A–32
DAY$ • A–9
DAYS • A–8
%DEBUG • 6–2
_DEBUG • A–1
DEBUG OFF • 4–2
DEBUG ON • 4–1
DEBUG system • 4–1
DECIMALS • A–32
DECLARE • 7–5
DECLARE DYNAMIC • 7–5
DECLARE PREFIX • 7–7
DECLARE statement • 15–6
DECLARE STRUCTURE • 7–6
DECODE • A–9
DEFAULT • 8–7
DEFAULT Menu Item • 8–7
DEFAULT OPTION • 8–7
Defining Arrays • 10–1
Definitions • 1–13
DEG • A–9
DEGREE • A–9
DELAY • 15–7, 15–8

DELETE STRUCTURE • 3–11
DESCRIPTION • 3–23
Development cycle time • xvii
DEVELOP QUALIFIER • 5–10
DIGITS • A–32
DIM Statement • 10–1
DIRECTIVES • 8–12
DISALLOW • A–31
DISPATCH • 12–11
DIV0 • A–9
DO Construct • 12–3
DO UNTIL • 12–3
DO WHILE • 12–3
DTYPE • A–10

E

EDIT • 5–6
EDIT$ • A–10
EDIT MASK
  See INPUT FROM A TERMINAL, SCREEN Option, format • 8–10
Edit string • A–10
ELAPSED OPTION • 8–8
ELEMENT$ • A–11
ELEMENTS • A–10
ENCODE$ • A–11
END Statement • 15–1
EPS • A–11
ERASE Option • 9–2
EVAL • A–11
Exception
  line • A–12
EXCEPTION HANDLING • 14–1
EXECUTE • 12–10
EXIT • 1–13
_EXIT • 8–4, A–1
EXIT ADD • 3–13
EXIT DO • 12–3
EXIT EXTRACT • 3–8
EXIT FOR • 12–1
EXIT HANDLER • 14–3
EXIT ROUTINE • 12–12
EXLABEL$ • A–12
EXLINE • A–12
EXP • A–12
EXPRESSION • A–34
EXPRESSIONS<Numeric • 15–6
EXTEXT$ • A–12
EXTRACT CONSTRUCT
  PARTIAL KEY • 3–9
_EXTRACTED • A–1
EXTRACT STRUCTURE • 3–8
EXTRACT STRUCTURE, SET, FIELD: KEY • 3–11
EXTRACT STRUCTURE, SET, USING • 3–10
EXTRACT STRUCTURE: APPEND • 3–9
EXTYPE • A–12

F

FALSE
  As a function • A–12
Field Expressions • 3–3
FIELDS
  Multiple Occurrences • 3–3
  Variable Length • 1–8
FILE EXTENSION • 2–1
FILENAME • 2–1
FILES • 13–1
FILESPEC$ • A–12
FILE SPECIFICATION • 2–1
FILE SPECIFICATIONS
  Extensions >.INT • 2–1
  Extensions >Default • 2–1
  in INTOUCH statements and commands • 2–1
FILL-IN AREAS • 8–10
FILTER • A–34
FINDFILE$ • A–13
FOR Construct • 12–1
FOR EACH LOOP • 3–6
FORMAT$ • A–14
FORMAT CHARACTERS • 8–10
  # • 8–10
  @ • 8–10
  ^ • 8–11
FORMAT OPTIONS
  Use of • 8–11
Fourth-generation database language • xvii
FP • A–15
FRAME OFF • 5–7
FRAME ON • 5–7
FULLTIME • A–33
FULLTIME$ • A–15
FUNCTIONS
  Built-in • A–1
  System • A–1

G

GO • 4–6, 4–7
GOSUB Construct • 12–8
GOTO Statement • 12–7
GREATER THAN • 9–4
Guide to the INTOUCH Language • xvii
  Conventions • B–1

H

HALT Statement • 15–2
HASH$ • A–15
HEADING • 3–24
%HEADING • 8–13
HELP • 3–24
_HELP • A–2
HIGHLIGHTING
  In fill-in areas • 8–10
HYPHEN • 9–3

I

ID
  Ask Structure • 3–27
IDE • 5–8
Identifiers • 15–4
IF Construct • 12–5
IMAGES
  Compiled • 5–9
INCLUDE • 3–5
%INCLUDE • 6–1
%INCLUDE CONDITIONAL • 6–1
INDEXED • 3–25
INPUT • 3–25, 8–3
  See also INPUT MENU
  See also LINE INPUT
INPUT #chnl_num • 13–2
INPUT AREA • 8–3
INPUT FROM A SCREEN
  SCREEN Option # • 8–10
INPUT FROM A TERMINAL • 8–1
  _BACK • 8–4
  Default input • 8–1
  Displaying prompts • 8–7
    PROMPT Option • 8–7
INPUT FROM A TERMINAL (cont'd)
  ELAPSED Option • 8–8
  _EXIT • 8–4
  _HELP • 8–4
  Inputting multiple variables • 8–5
  LENGTH Option • 8–7
  SCREEN Option • 8–10
    angle brackets • 8–10
    Example • 8–10
    format • 8–10
    Format characters • 8–10
    Format Options • 8–10, 8–11
    ^ • 8–11
  SCREEN Option @ • 8–10
  See also KEY INPUT • 8–14
  TIMEOUT Option • 8–8
  To a string variable • 8–5
  VALID Option • 8–8
INT • A–16
INTEGER • A–16, A–32
_INTEGER • A–2
Integer variables • 15–5
INTEGER WORD • A–32
INTOUCH language • xvii
INTOUCH Tutorial • xvii
.INT_IMG • 5–9
.INT_IMG_AXP • 5–9
IP num_expr • A–16
ITEM • A–16
%ITEMS • 8–13
ITERATE DO • 12–4
ITERATE FOR • 12–2

K

KEYED • 3–24
KEY INPUT • 8–14
KILL Statement • 13–6

L

LABELS
  Basic description • 15–3
LBOUND • A–17
LCASE • 9–3, A–31
LCASE$ • A–17, A–35
LEFT[$] • A–17

LEN • A–17
LENGTH • 3–24, A–31
LENGTH OPTION • 8–7
LESS THAN • 9–4
LET • 7–1
LETTERS • A–31
Libraries • C–1
LIBRARY
  STATEMENT>DESCRIPTION OF • 14–6
LIBRARY NAME
  DESCRIPTION OF • 14–6
LIBRARY ROUTINES
  PASSING ARGUMENTS TO • 14–6
LIBRARY STATEMENTS
  CALL • 14–6
  LIBRARY • 14–6
LIBR_NAME • 14–6
LINE INPUT AREA • 8–3
LINE INPUT Statement • 13–2
Line numbers
  As labels • 15–3
[LINE]INPUT, ERASE • 8–8
[LINE] INPUT MENU • 8–11
  See also LINE INPUT
LIST • 5–5
LIST STATS • 4–4
LOCK • 3–14
%LOCKSTEP • 8–13
LOG • A–17
LOG10 • A–17
LOG2 • A–17
LOOPS • 12–1
LOW BOUND • 10–3
LPAD$ • A–17
LSET • 7–2
LSET, RSET, CSET FILL • 7–3
LTRIM$ • A–18, A–34

M

? MARK • 9–4
MATCH • A–18
MAX • A–18
MAXLEN • A–18
MAXLENGTH • A–31
MAXNUM • A–18
MAXSIZE • A–18
%MAXWIDTH number • 8–13
MENU • A–34
%MENUBAR • 8–13
MENU Option • 8–3, 8–11
%MESSAGE • 6–2
%MESSAGE ERROR • 6–3
MESSAGE LINE • 9–5
MESSAGE Statement • 9–5
MID • A–19
MID$ • A–19
MIN • A–19
MINLENGTH • A–31
MOD • A–19
%MULTI • 8–12
Multiple Statements • 15–3

N

NAME • 3–24
NEW • 5–2
NOCHARACTERS • A–31
NULL • 3–24
NULL STRING • 3–25
NUMBER • 3–24, A–32

O

OLD • 5–4
ON GOSUB Construct • 12–9
ON GOTO • 12–8
OPEN #int_expr: NAME str_expr, UNIQUE, ACCESS
    OUTPUT • 13–5
OPEN: NAME, OPTIMIZE OFF • 13–5
OPEN Statement • 13–1
OPEN STRUCTURE • 3–3
OPTIMIZED • 3–24
OPTIMIZE OFF • 13–5
OPTION BASE • 10–2
OPTION COMMUNICATE ON |
  OFF • 9–8
OPTION REQUIRE DECLARE • 7–8
ORD • A–19
ORDNAME$ • A–19
OUTPUT • 3–25

P

PARSE$ • A–19
PARTIAL KEY • 3–9
PASS NORETURN • 12–15
PASS [NOWAIT: | NORETURN:] • 12–14
PATTERN • A–33
PATTERN() • A–20
PAUSE • 15–7
PI • A–21
PIECE$() • A–21
PIECES() • A–21
POS • A–22
POSITION • 3–25
PRETTY$ • A–22
PRINT
  Synonym • 9–4
PRINT ERASE • 9–2
Printing Data to a Terminal • 9–1
PRINTMASK • 3–25, A–33
PRINT options • 9–1
PRINT Statement • 9–1, 13–6
PRINT USING • 9–3
PROGRAM Statement • 15–1
PROMPT • 3–25, 8–3
PROMPT Option • 8–7
PROMPTS
  Displaying • 8–7
Prompt Text • 1–9

Q

QUOTE$ • A–23

R

RAD • A–23
Random • A–24
RANDOMIZE Statement • 15–7
READ
  DESCRIPTION OF • 7–3
REAL • A–23
_REAL • A–2
Real variables • 15–5
REDIM Statement • 10–1
REEXTRACT • 3–6
REFRESH • 5–7
RELATIVE • 3–25
REMAINDER • A–23
REMOVE • A–35
RENUMBER • 5–6
REPEAT$ • A–24
REPEAT DO • 12–4
REPEAT FOR • 12–2
REPEAT ROUTINE • 12–13
REPLACE • 5–3, A–35
%REPLACE • 8–12
REPLACE$ • A–24
_REPLY • A–2
REQUIRED • A–33
Reserved Identifiers • 15–6
RESTORE • A–35
  DESCRIPTION OF • 7–4
RESUME • 14–5
RETRY • 14–3
RETURN • 12–8
RIGHT • A–24
RIGHT$ • A–24
RND • A–24
RNH • 5–2
ROUND • A–25
ROUTINE • 12–11, 12–12
ROUTINE NAME
  DESCRIPTION OF • 14–6
RPAD$ • A–25
RSET • 7–2
RTRIM$ • A–25, A–34
RUN • 5–2

S

SAVE • 5–2
Scalars • 15–5
SCAN • A–25
SCREEN
  CLEAR • 9–6
    AREA Option • 9–6
SCREEN MANAGEMENT FACILITY
  See also INPUT FROM A TERMINAL, SCREEN Option • 8–10
SCREENMASK • 3–25
SCREEN OPTION • 8–10
SEC • A–25

Secant • A–25
SECONDS • A–25
SEG$ • A–26
SELECT CASE • 12–6
Semicolons in PRINT statement • 9–5
SET AUTOEXIT • 11–1
SET BACK ON | OFF • 11–2
SET ERROR ON | OFF • 11–2
SET EXIT ON | OFF • 11–3
SET HELP ON | OFF • 11–4
SET MARGIN • 11–5, 13–5
SET MESSAGELINE • 11–5
SET PORT ON | OFF • 11–6
SET SCROLL • 11–7
SET SEED • 11–8
SET Statement • 11–1
SET STRUCTURE • 3–17, 3–30
SET STRUCTURE, FIELD: PARTIAL KEY • 3–18
SET STRUCTURE, SET, USING • 3–15
SET STRUCTURE, SET, USING, FIELD: KEY • 3–17
SET STRUCTURE, SET, USING: EXPRESSION 'owner' • 3–16
SET STRUCTURE, SET: EXPRESSION 'owner' • 3–16
SET STRUCTURE: CURRENT • 3–17
SET STRUCTURE : EXTRACTED 0 • 3–15
SET STRUCTURE: ID • 3–19
SET STRUCTURE: POINTER • 3–20
SET STRUCTURE: RECORD • 3–20
SET STRUCTURE: TIMEOUT • 3–21
SET SYSTEM, LOGICAL • 11–10
SET SYSTEM, SYMBOL • 11–13
SET SYSTEM: COMMENT • 11–9
SET SYSTEM: DIRECTORY • 11–9
SET SYSTEM: PROCESS • 11–12
SET SYSTEM options • 11–8
SETUP
  Access Rules • 1–10
  Call • 1–2
  Create • 1–1
  Create Procedure • 1–10
  Data Type • 1–7
  Define • 1–1
  Define Procedure • 1–5
  Definition File • 1–4
  Definitions • 1–12
  Description • 1–6
  Ending-position • 1–8
  Exit • 1–1, 1–10
  Field Attribute >Date • 1–7
  FIELD ATTRIBUTES • 1–8
SETUP (cont'd)
  Field Length • 1–8
  Field Name • 1–6, 1–10
  Help Text • 1–9
  Print Mask • 1–9
  Procedure • 1–5
  Record System (RMS) • 1–3
  Report Heading • 1–9
  Screen Mask • 1–9
  Show • 1–1
  Show Procedure • 1–12
  Structure Name • 1–3
SETUP prompts • 1–2
SETUP ROUTINE • 1–1
SET WINDOW: COLUMN • 11–15
SET WINDOW: CURRENT • 11–16
SET WINDOW: DATA • 11–17
SET WINDOW: KEYMAP • 11–18
SET WINDOW: ROW • 11–20
SET WINDOW: TYPEAHEAD • 11–21
SET WINDOW AREA • 11–14
SET WINDOW KEYSTROKE • 11–19
SET WINDOW options • 11–14
SET ZONEWIDTH • 11–22
SGN • A–26
SHOW
  Definitions • 1–12
  Field Definitions • 1–12
SIN • A–26
SINH • A–26
SIZE • A–26
%SIZE number • 8–12
SKIP • A–26
SLEEP • 15–7
SORT
  ASCENDING • 3–5
  DESCENDING • 3–5
SORT$ • A–27
SPACE$ • A–27
Specifications • xviii
%SPLIT • 8–13
SQR • A–27
STATEMENTS
  ASK • 11–1
  CHAIN • 12–13
  CLOSE • 13–5
  CONTINUE • 14–4
  CSET • 7–2
  DECLARE • 7–5
  DELAY • 15–7
  DIM • 10–1
  DISPATCH • 12–11

STATEMENTS (cont'd)
END • 15–1
EXECUTE • 12–10
EXIT DO • 12–3
EXIT FOR • 12–1
EXIT HANDLER • 14–3
FOR EACH • 3–6
GOTO • 12–7
HALT • 15–2
ITERATE DO • 12–4
ITERATE FOR • 12–2
KILL • 13–6
LET • 7–1
LINE INPUT • 13–2
LSET • 7–2
MESSAGE • 9–5
OPEN • 13–1
PRINT • 9–1, 13–6
PROGRAM • 15–1
RANDOMIZE • 15–7
REDIM • 10–1
REPEAT DO • 12–4
REPEAT FOR • 12–2
RESUME • 14–5
RETRY • 14–3
ROUTINE • 12–11
RSET • 7–2
SET • 11–1
STOP • 15–1
STATISTICS • 4–1
STATS OFF • 4–4
STATS ON • 4–4
_STATUS • A–2
STEP • 4–6
STOP Statement • 15–1
STR$ • A–27
_STRING • 8–7, A–3
STRING FUNCTIONS
ADDED
DAY$ • A–1
String variables • 15–5
Structure • 15–6
STRUCTURE STATEMENTS
ADD STRUCTURE • 3–12
ASK STRUCTURE • 3–21
CANCEL ADD • 3–12
CANCEL EXTRACT • 3–7
CLOSE STRUCTURE • 3–4
DELETE STRUCTURE • 3–11
EXIT ADD • 3–13
EXIT EXTRACT • 3–8
EXTRACT CONSTRUCT>PARTIAL KEY • 3–9
STRUCTURE STATEMENTS (cont'd)
EXTRACT STRUCTURE • 3–4
LOCK STRUCTURE • 3–14
OPEN STRUCTURE • 3–3
REEXTRACT STRUCTURE • 3–5
SET STRUCTURE • 3–17
UNLOCK STRUCTURE • 3–14
SYSEXT$ • A–27

T

TAB • A–28
TAN • A–28
Tangent • A–28
TANH • A–28
Technical Specifications
Intended audience • xviii
_TERMINATOR • 8–14, 8–15, A–3
TIME • A–28
TIME$ • A–28
TIME(5) • A–29
TIMEOUT • 8–3
TIMEOUT OPTION • 8–8
%TITLE • 8–12
TRACE • 4–1, 4–2
TRACE OFF • 4–3
TRACE ON/OFF • 4–2
TRANSFER OF CONTROL
Within code blocks • 12–13
Translate • A–36
Transparent interface • xvii
Trim
leading spaces • A–18
TRIM$ • A–29, A–34
TRUE
As a function • A–29
TRUNCATE • A–29
TTI_PRINTER_TYPE • 11–6

U

UCASE • 9–3, A–32
UCASE$ • A–29, A–35
UNDERSCORE CHARACTER
In fill-in areas • 8–10
UNFORMATTED Option • 13–1
UNLOCK • 3–14

UNQUOTE$() • A–29
UNSAVE • 5–5

V

VAL • A–30
VALID • A–30
VALID OPTION • 8–8
VARIABLE LENGTH FIELDS • 1–8
%VBAR • 8–13
VRULES • 3–25, A–33

W

WAIT • 15–7
WHEN EXCEPTION IN • 14–1
WHEN EXCEPTION USE • 14–2
WIDE Option • 9–2
%WIDTH number • 8–13
WRAP$() • A–36

X

XLATE$ • A–36

Y

YES/NO • A–33

Z

ZE • 1–7