VAX® INTOUCH®

Application Guide

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

Touch Technologies, Inc.
9988 Hibert Street, Suite 310
San Diego, CA 92131
1-800-525-2527 or 1-619-566-3603

® VAX is a registered trademark of Digital Equipment Corporation.

® INTOUCH is a registered trademark of Touch Technologies, Inc.

NOTICE

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

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

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

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

DYNAMIC TAPE ACCELERATOR INTOUCH REMOTE TAPE FACILITY

DYNAMIC LOAD BALANCER PLUS FRAGMENTATION CONTROLLER REMOTE DEVICE FACILITY

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

RMS Expert

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

DBMS DCL DECNET RDB RMS VAX VMS

Last Revision date: 06-Oct-1994

Copyright ©1993, 1994 Touch Technologies, Inc.

Contents

PREFACE ix

CHAPTER 1 CONTROLLING THE SCREEN APPEARANCE 1–1

1.1 INTOUCH SCREEN FRAME 1–1

1.2 CLEARING THE SCREEN 1–2
1.2.1 Boxes and Screen Frames 1–4

1.3 SAVING AND RESTORING THE SCREEN 1–6
1.3.1 Saving and Restoring Screen Areas 1–8
1.3.2 Other ASK/SET WINDOW Statements 1–10

1.4 FORMATTING THE SCREEN 1–10
1.4.1 Setting the Margin 1–10
1.4.2 Setting Print Zones 1–11

CHAPTER 2 CUSTOMIZING USER INPUT 2–1

2.1 INPUT OPTIONS 2–1

2.2 USING FILL-IN FIELDS 2–4

2.3 USING FREE FORMATTED AREAS 2–5

CHAPTER 3 CREATING MENUS 3–1

3.1 MENU COMPOSITION 3–1

3.2 HINTS FOR CREATING MENUS 3–2

Contents

3.3	USING MENUS	3–3
3.4	MENU EXAMPLES	3–3
3.5	MENU DIRECTIVES	3–11
3.6	DEFAULTING TO A MENU ITEM	3–12
3.7	RETURNING TO A MENU ITEM	3–13
3.8	ASSIGNING VALUES TO MENU ITEMS	3–14

CHAPTER 4 USING MESSAGES AND DELAYS 4–1

4.1	MESSAGE TYPES	4–1
4.2	CONTROLLING MESSAGE PLACEMENT	4–3
4.3	USING DELAY	4–5

CHAPTER 5 CHECKING INPUT RESPONSES 5–1

5.1	CHECKING _REPLY	5–2
5.2	CHECKING THE _TERMINATOR	5–3
5.3	CHECKING FOR _BACK AND _EXIT	5–4
5.4	CHANGING KEY VALUES	5–5
5.5	SUPPORTED KEYS, KEYSTROKE COMBINATIONS	5–7

CHAPTER 6 WORKING WITH INTOUCH STRUCTURES 6–1

6.1 CREATING STRUCTURES 6–1

6.2 OPENING AND CLOSING STRUCTURES 6–2

6.3 REFERENCING STRUCTURE DATA 6–2

6.4 EXAMPLES USING STRUCTURES 6–5

CHAPTER 7 WORKING WITH FILES 7–1

7.1 OPENING AND CLOSING FILES 7–1

7.2 STORING AND READING FILE DATA 7–2

7.3 POSITIONING FILE RECORDS 7–4

CHAPTER 8 WORKING WITH DATA 8–1

8.1 FORMATTING DATA 8–1

8.2 EDITING DATA 8–4

8.3 EXTRACTING DATA 8–6

8.4 CHANGING/REPLACING CHARACTERS/STRINGS 8–9

8.5 CONVERTING DATA 8–10

Contents

CHAPTER 9 LOGIC FLOW AND ROUTINES 9–1

9.1 CONDITIONAL STATEMENTS 9–1

9.2 LOOPED LOGIC 9–3

9.3 ROUTINES 9–5
9.3.1 Calling Routines 9–6
9.3.2 REPEAT, EXIT Routines 9–7

CHAPTER 10 WORKING IN THE INTOUCH ENVIRONMENT 10–1

10.1 INTOUCH EDITOR 10–1

10.2 SWITCHING BETWEEN INTOUCH AND VMS 10–1
10.2.1 Running VMS Commands from INTOUCH 10–2
10.2.2 Passing VMS Commands 10–2

10.3 INTOUCH FILE NAMES 10–3

10.4 INTOUCH PROCEDURES 10–4

10.5 END, STOP, HALT, GO STATEMENTS 10–6

10.6 RECALLING PREVIOUS COMMANDS 10–6

10.7 THE TAB KEY 10–7
10.7.1 Command Completion 10–7
10.7.2 Command Abbreviations 10–7
10.7.3 Correcting Misspelled Words 10–8

10.8 USING THE INTOUCH EDITOR 10–8

CHAPTER 11 DEBUGGING TECHNIQUES 11–1

11.1 ENABLING THE DEBUG SYSTEM 11–2

11.2 SETTING A BREAK POINT TO CHECK VARIABLES 11–2
11.2.1 Checking Variables 11–3

11.3 STATS AND TRACE FEATURES 11–4

11.4 MODIFYING CODE AT RUNTIME 11–5

APPENDIX A SAMPLE PROGRAM A–1

APPENDIX B USER NOTES B–1

INDEX

EXAMPLES
1–1 INTOUCH Screen Frame 1–1
9–1 Example of a Routine 9–6

TABLES
3–1 Menu Directives 3–11
8–1 EDIT$ Function - Operation Values 8–6
10–1 Abbreviated INTOUCH Statements 10–7

Preface

INTOUCH 4GL

INTOUCH is a high performance 4GL, which blends many 3GL and 4GL capabilities, making it the most powerful and flexible language on OpenVMS systems.

INTOUCH is not what you would expect from a typical 4GL. Not only is it fast and efficient at using system resources, its methods of I/O optimization make it speedier than other 4GLs, including many 3GLs on I/O intensive applications.

About this Manual

This is one of several reference manuals describing the INTOUCH system. All 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
- Application Guide

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

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

The *INTOUCH Application Guide* presents the INTOUCH language by application topic. Not only does it help you find the right INTOUCH statements for your application, but it offers hints and provides examples to back them up.

Purpose

The purpose of this *INTOUCH Application Guide* is to help you through building INTOUCH 4GL based applications. This guide assumes that you have a programming background. It also assumes that you have access to the *INTOUCH—A Guide to the Language* manual.

Organization

In order to facilitate its ease of use, the *INTOUCH Application Guide* is organized by application topic. Each chapter of the guide addresses one such topic (i.e. Controlling Screen Appearance, User Input, etc.). Chapter sections then approach the application at the statement level and present you with suggested INTOUCH statements. However, it is suggested that you keep the *INTOUCH - A Guide to the Language* manual handy for detailed reference of statement format. Examples are given to aid in your understanding.

1 Controlling the Screen Appearance

The ability to control the appearance of the screen is important to any applications programmer. INTOUCH gives programmers this control by allowing them to manipulate the entire screen and specific areas of the screen.

The sections which follow briefly explain how to manipulate the screen.

1.1 INTOUCH Screen Frame

When you get into INTOUCH (by typing the word **INTOUCH**), the INTOUCH screen frame is displayed.

Example 1–1: INTOUCH Screen Frame

```
.screen -
INTOUCH~4.1j~~~~~~~~~~~~~~~~~~Program:~NONAME~~~~1~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The~Next~Generation~Language~~Status~:~~~~~~~~~~2~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                        1                        3
                        2                        4
                        3                        5      print at 7,1:
                        4                        6        prints on
    print at 7,1:       5                        7   <--- this line
      prints on         6                        8
      this line  --->   7                        9      when the
                        8                       10      INTOUCH screen
    when the            9                       11      frame is OFF
    INTOUCH screen     10                       12
    frame is ON        11                       13
                       12                       14
                       13                       15
                       14                       16
                       15                       17
                       16                       18
INTOUCH                17                       19
                       18                       20
cursor                 19                       21
~~~~~~~~~~~~~~~~~~~~~~20~~~~~~~~~~~~~~~~~~~~~~22~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~ Messages are displayed on screen line 23 ~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~24~~~~~~~~~\~=~Back~~HELP~=~Help
```

There are 21 available rows on the screen when the INTOUCH screen frame is on. There are 24 rows when the frame is off. There are 80 columns or 132, depending on the terminal's margin setting. Messages are displayed on line 23.

- You can use the **FRAME OFF** statement to turn the INTOUCH screen frame off.
- Use **FRAME ON** to turn the INTOUCH screen frame on.

Note

The examples shown in this manual are run with the INTOUCH screen frame OFF.

1.2 Clearing the Screen

INTOUCH allows you to clear either the entire screen or rectangular areas of the screen using the **CLEAR** statement.

- The **CLEAR** statement clears all text from the screen. If the INTOUCH screen frame is on, everything **within** the frame is cleared. Any messages embedded in the screen frame are also cleared.
- **CLEAR AREA**

  The **AREA** option of the CLEAR statement indicates that only a rectangular portion of the screen is to be cleared, as specified by upper left and lower right coordinates.

If you want to enhance the cleared area of the screen with attributes, **BOLD**, **BLINK**, **REVERSE** and **UNDERLINE**, specify the desired option(s) in the statement followed by the ":" option identifier. Multiple items can be used separated by commas.

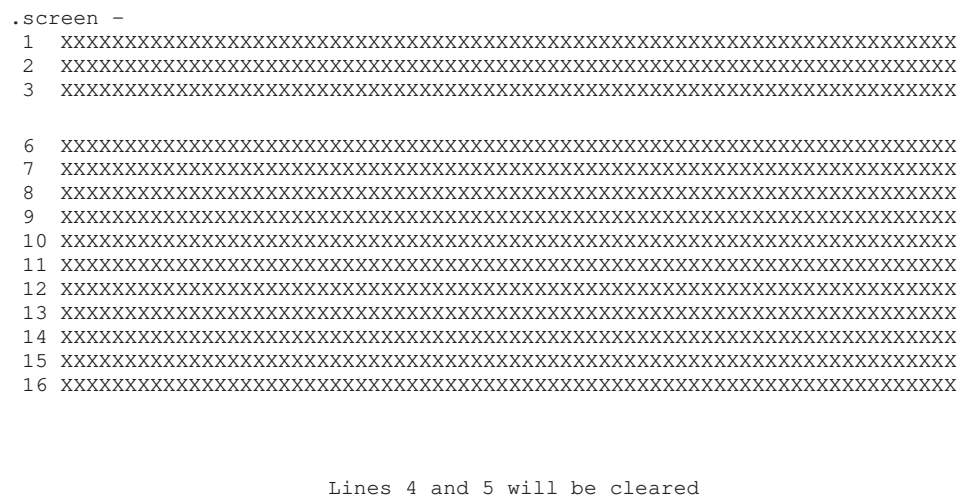
You can also replace the cleared area with a box by using the **BOX** option.

Example

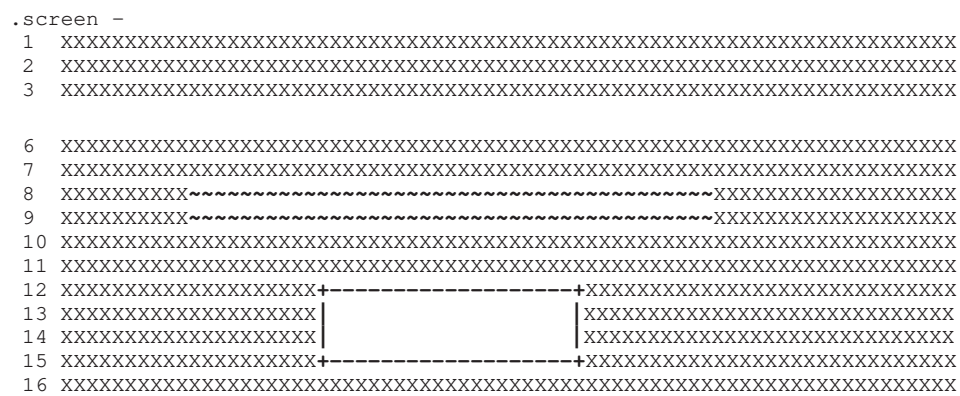
The following example clears the screen and specific areas of the screen.

```
1   program application_1_1
10  clear
20  print at 1,1:;
    for x = 1 to 16
      print x; tab(5); repeat$('X', 70)
    next x
30  message 'Lines 4 and 5 will be cleared'
    delay 5
    clear area 4, 1, 5, 80
40  message 'Parts of lines 8 and 9 will be cleared and reversed'
    delay 5
    clear area reverse: 8, 15, 9, 55
50  message ' Parts of lines 12 thru 15 will be cleared and boxed'
    delay 5
    clear area box, bold: 12, 25, 15, 45
    delay 5
60  end
```

Text is printed to the screen. Rows and areas of the screen are cleared in various ways. Lines 4 and 5 are cleared first.



Parts of lines 8 and 9 are cleared and shown in reverse video. Parts of lines 12 thru 15 are cleared and the area is boxed.



1.2.1 Boxes and Screen Frames

Boxes are often used to highlight data on the screen. A user tends to focus on items which stand out. Boxes can be created to frame a screen or to frame sections of text displayed on the screen.

INTOUCH allows you to create boxes with the **BOX** option of the **CLEAR AREA** statement. Given a rectangular area of upper left and lower right coordinates, INTOUCH outlines the area with a box.

If you want to enhance the box with attributes (i.e. BOLD, BLINK, REVERSE, UNDERLINE), specify the desired attribute(s) as options listed after the **BOX** option and followed by the box coordinates.

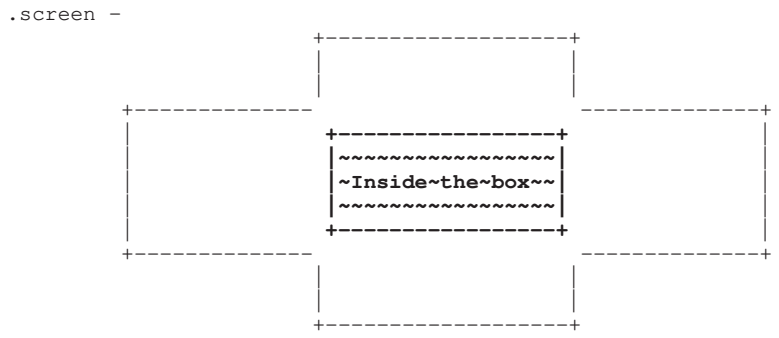
Example

This example draws several boxes and prints data inside a box.

```
1   program application_1_2
10  clear
    clear area box : 6, 10, 12, 60
    message 'Here is the 1st box'
    delay 5
    clear area box : 3, 25, 15, 45
    message 'Here is the 2nd box'
    delay 5
    clear area 6, 25, 12, 45
    message 'A boxed area is cleared'
    delay 5
    clear area box, reverse : 7, 26, 11, 44
    message 'Here is the 3rd box done in reverse video'
    delay 5
    print at 9, 28 : 'Inside the box'
    message 'Printing inside of a box'
    delay 5
20  end
```

```
.screen -
         +-------------------------------------------------+
         |                                                 |
         |                                                 |
         |                                                 |
         |                                                 |
         |                                                 |
         +-------------------------------------------------+


                        Here is the 1st box
```



Printing inside of a box

Screen frames similar to the INTOUCH screen frame are often used in application programs. Screen frames can be used to identify user procedures.

Example

```
1   program application_1_3
10  clear
    z$ = space$(80)
    cset z$ = 'Customer Maintenance'
    print reverse, bold, at 1, 1: z$;
    z1$ = 'EXIT = Exit' + space$(48) + '\ = Back  HELP = Help'
    print reverse, bold, at 24, 1 : z1$;
    print at 3, 1: 'Name    :'
    print at 4, 1: 'Address :'
    print at 5, 1: 'City    :'
    print at 6, 1: 'State   :'
    print at 7, 1: 'Zip     :'
    delay
    clear
20  end
```

```
.screen -
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Customer~Maintenance~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Name    :
Address :
City    :
State   :
Zip     :



                          Press the RETURN key to continue
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

1.3 Saving and Restoring the Screen

There are times when you want to save a screen and restore it later.

The **ASK WINDOW** statement takes a snapshot of the current screen or a part of it and saves it in a string variable.

The **SET WINDOW** statement restores a previously saved screen or part of a screen.

- **ASK WINDOW: CURRENT**

  The **CURRENT** option of the ASK WINDOW statement causes INTOUCH to take a snapshot of the *entire* screen and save it in a specified string variable. If the INTOUCH screen frame is on, messages embedded anywhere in the screen frame will also be saved.

- **SET WINDOW: CURRENT**

  The **CURRENT** option of the SET WINDOW statement causes INTOUCH to restore the entire screen image previously stored in a specified string variable, including messages embedded anywhere in the INTOUCH screen frame.

  If the **CURRENT** option was used in the ASK WINDOW statement, it should also be used in the SET WINDOW statement.

Example

The following example prints a screen, saves the screen display and restores the saved screen.

```
1   program application_1_4
10  clear
20  print at 5,5 : repeat$('*', 40)
    print at 6,5 : repeat$('*', 40)
    print at 7,15 : 'This is screen 1'
    print at 8,5 : repeat$('*', 40)
    print at 9,5 : repeat$('*', 40)
    message 'Screen 1 will be saved and Screen 2 will be displayed'
    delay 5
    ask window: current screen_1$
    clear
30  print at 5,5 : repeat$('#', 40)
    print at 6,5 : repeat$('#', 40)
    print at 7,15 : 'This is screen 2'
    print at 8,5 : repeat$('#', 40)
    print at 9,5 : repeat$('#', 40)
    message 'Screen 2 will be replaced by screen 1'
    delay 5
40  set window: current screen_1$
50  delay 5
60  end
```

Screen 1 is printed and saved:

```
.screen -
    ****************************************
    ****************************************
              This is screen 1
    ****************************************
    ****************************************


              Screen 1 will be saved and Screen 2 will be displayed
```

Screen 2 is printed to the screen:

```
.screen -
    ########################################
    ########################################
              This is screen 2
    ########################################
    ########################################


                    Screen 2 will be replaced by screen 1
```

Screen 1 is restored:

```
.screen -
    ****************************************
    ****************************************
              This is screen 1
    ****************************************
    ****************************************


              Screen 1 will be saved and Screen 2 will be displayed
```

1.3.1 Saving and Restoring Screen Areas

You can also save and restore specific areas of the screen.

- **ASK WINDOW AREA ... : DATA**

  When a specific section of the screen needs to be saved, you can use the **AREA** option with the **DATA** qualifier. Specify the area to be saved as a rectangle defined by upper left and lower right coordinates.

- **SET WINDOW AREA ... : DATA**

  Use the **AREA** option with the **DATA** qualifier to overlay the specified rectangular area onto the existing screen.

  If the ASK WINDOW statement used the **AREA** option and **DATA** qualifier, the SET WINDOW statement must do the same.

Example

The following example shows how areas of the screen are saved and restored.

```
1   program application_1_5
10  clear
20  print at 4,5 : repeat$('*', 40)
    print at 5,5 : repeat$('*', 40)
    print at 6,15 : 'This is area 1'
    print at 7,5 : repeat$('*', 40)
    print at 8,5 : repeat$('*', 40)
    print at 11,5 : repeat$('#', 40)
    print at 12,5 : repeat$('#', 40)
    print at 13,15 : 'This is area 2'
    print at 14,5 : repeat$('#', 40)
    print at 15,5 : repeat$('#', 40)
    message 'Areas 1 and 2 are going to be saved'
    delay 5
30  ask window area 4, 1, 8, 80 : data screen_1$
    ask window area 11, 1, 15, 80 : data screen_2$
    set window area 4, 1, 8, 80 : data screen_2$
    message 'Area 1 has been overlayed by Area 2'
    delay 5
40  set window area 11, 1, 15, 80 : data screen_1$
    message 'Area 2 has been overlayed by Area 1'
50  end
```

Data is displayed on the screen and the areas are saved.

```
.screen -
    ****************************************
    ****************************************
              This is area 1
    ****************************************
    ****************************************

    ########################################
    ########################################
              This is area 2
    ########################################
    ########################################
```

The saved areas are restored to different sections of the screen.

```
.screen -
    ########################################
    ########################################
              This is area 2
    ########################################
    ########################################

    ****************************************
    ****************************************
              This is area 1
    ****************************************
    ****************************************
```

1.3.2 Other ASK/SET WINDOW Statements

The above examples are only two options of the ASK/SET WINDOW statements. There are many other options available and each of the options is explained in detail in the *INTOUCH - A Guide to the Language* manual.

1.4 Formatting the Screen

Controlling margins and print zones is very significant in formatting screen data.

1.4.1 Setting the Margin

INTOUCH allows you to control margins with the **ASK MARGIN** and **SET MARGIN** statements. By default, margins are set to 80 and the device or channel is #0, the screen terminal.

- The **ASK MARGIN** statement returns the current right margin in a specified number variable.

  A device or channel can also be specified. This is useful when you are sending data out through multiple channels. If no device or channel is specified, the default is #0, the terminal.

- The **SET MARGIN** statement sets the margin to the specific number value.

  A device or channel can also be specified. The default is #0, the terminal.

Example

This example shows how to find the margin, set it to something else and set it back to the current margin.

```
1   program application_1_6
10  clear
    print repeat$('*', 200)
    print
20  ask margin current_margin
    print 'The current margin is:'; current_margin
    input 'What do you want the new margin to be set to'; new_margin
30  print
    print 'The new margin is:'; new_margin
    print
    set margin new_margin
    print repeat$('*', 200)
40  set margin current_margin
    print
    print 'The margin has been reset to the original value'
    print
    print repeat$('*', 200)
50  end
```

```
.screen -
********************************************************************************
********************************************************************************
************************************

The current margin is: 80
What do you want the new margin to be set to? 50

The new margin is: 50

************************************************
************************************************
************************************************
************************************************

The margin has been reset to the original value

********************************************************************************
********************************************************************************
************************************
```

1.4.2 Setting Print Zones

The *comma* separator in the PRINT statement splits data into columns of width determined by the value of *ZONEWIDTH*. This parameter is useful when formatting data to the screen or to files. The default ZONEWIDTH for all channels is 20 and the default channel is #0, the screen terminal.

INTOUCH gives you control of the device or channel ZONEWIDTHs with the **ASK ZONEWIDTH** and **SET ZONEWIDTH** statements.

- The **ASK ZONEWIDTH** statement returns the current zone width in a specified number variable.

  A device or channel can also be specified. If no device or channel is specified, the default is #0, the terminal.

- The **SET ZONEWIDTH** statement sets the zone width to the specified numeric value.

  A device or channel can also be specified. If no device or channel is specified, the default is #0, the terminal.

Example

This example shows how to find the zone width, set it to something else and set it back to the current zone width.

```
1   program application_1_7
10  clear
    ask zonewidth current_zonewidth
    print 'ZONE-1', 'ZONE-2', 'ZONE-3', 'ZONE-4', 'ZONE-5', 'ZONE-6', 'ZONE-7'
    print
    print 'The current zone width is:'; current_zonewidth
20  input 'What would you like to set the new zone width to'; new_zonewidth
    print
30  print 'The new zone width is:'; new_zonewidth
    print
    set zonewidth new_zonewidth
    print 'ZONE-1', 'ZONE-2', 'ZONE-3', 'ZONE-4', 'ZONE-5', 'ZONE-6', 'ZONE-7'
40  print
    print 'The zone width is restored to the original value'
    print
    set zonewidth current_zonewidth
    print 'ZONE-1', 'ZONE-2', 'ZONE-3', 'ZONE-4', 'ZONE-5', 'ZONE-6', 'ZONE-7'
50  end
```

```
.screen -
ZONE-1              ZONE-2              ZONE-3              ZONE-4
ZONE-5              ZONE-6              ZONE_7

The current zone width is: 20
What would you like to set the new zone width to? 10

The new zone width is: 10

ZONE-1    ZONE-2    ZONE-3    ZONE-4    ZONE-5    ZONE-6    ZONE_7

The zone width will be restored to the original value

ZONE-1              ZONE-2              ZONE-3              ZONE-4
ZONE-5              ZONE-6              ZONE_7
```

2 Customizing User Input

INTOUCH offers a variety of methods to prompt for and accept input from the user. These methods can be used alone or in combination with others depending on how the user is to be addressed.

The sections that follow give some examples of how INTOUCH can be used to accept and control user input.

2.1 Input Options

There are many options for the **INPUT** statement which can be used to customize user input. We will now explore some common input methods offered as options to the INPUT statement. With these options you can do such things as change the input prompt, position the input, set default values, validate the input and return timing statistics.

Example

The following example shows and describes some forms of the INPUT statement.

```
1   program application_2_1
10  clear
    print at 1, 1:;
    input '1 - What is your favorite month' : answer$
    line input '2 - What are your favorite animals' : answer$
    input prompt '3 - My favorite car is a ' : answer$
    input '4 - My favorite season of the year is ', at 5, 1 : answer$
    input '5 - Do you like snakes (Yes/No)', default 'No' : answer$
    input '6 - What is the date today (MMDDYY)', valid 'date MDY' : answer$
    input '7 - This month has how many days', &
                  valid 'number; allow 28 to 31' : answer
    print
    print '8 - You have 10 seconds to answer this question.'
    input '    What is the 4th letter of the alphabet', timeout 10 : answer$
    print
    print '9 - Answer this question as fast as you can.'
    input '    What day is today (Monday, Tuesday, etc.)', elapsed x : answer$
    print '    It took you'; x; 'seconds to type '; answer$
40  end
```

```
.screen -
❶ - What is your favorite month? September

❷ - What are your favorite animals? dogs, cats, horses

❸ - My favorite car is a porsche

❹ - My favorite season of the year is ? spring

❺ - Do you like snakes (Yes/No)? No

❻ - What is the date today (MMDDYY)? 041294

❼ - This month has how many days? 30

❽ - You have 10 seconds to answer this question.
    What is the 4th letter of the alphabet? d

❾ - Answer this question as fast as you can.
    What day is today (Monday, Tuesday, etc.)? tuesday
    It took you 3.81 seconds to type tuesday
```

❶ **Using TEXT to instruct the user**

You can use **text** to tell the user what to enter. INTOUCH prints your text followed by a **?** and a space. This is one of the most common ways to ask for an input item.

❷ **Enter a line of data**

You can use the **LINE INPUT** command if what is to be entered is a line of data, an address for example.

❸ **Use the PROMPT option to eliminate the ?**

If you do not want a **?** to print, you can use the **PROMPT** option. If you do not want any prompt at all, use the **PROMPT** option with a null string (empty quotes).

❹ **Use the AT option to position input on the screen**

If you need to place the input at a specific area of the screen, you can use the **AT** option followed by the row and column. This is useful for columns of data. With the INTOUCH frame ON, there are 21 available rows and 80 columns (132 if wide). Without the INTOUCH frame, there are 24 rows.

❺ **Use the DEFAULT option to provide the data**

If the same data is being input over and over again, you can use the **DEFAULT** option to provide the data. This speeds up data entry and allows the user to override the default to enter other data when necessary. Another benefit is that the defaults can keep data fields from being null values.

Either a string expression or string variable can be used for the default value.

❻ **Use the VALID option to validate input data**

There are often times when the input data needs to be validated. Dates are a good example. You might want the date entered as month-day-year or year-month-day and, of course, you want a valid date. INTOUCH will allow you to specify date formats and also will validate the input data as a valid date.

Using the **VALID** option allows you to validate not only dates but also data types (i.e. number, integer), acceptable input data items (a list of possible answers), etc. You can select one or more of the rules that are listed in the appendix section "Built-in Functions" of the *INTOUCH - A Guide to the Language* manual. INTOUCH will only allow input data which satisfies the validation rule(s) and will display an error message if invalid data is entered.

Here is an example which validates numbers:

```
1   program application_2_2
10  clear
    print at 1, 1:;
20  input prompt 'Enter a screen row number: ', length 2, &
        valid 'number; allow 1 to 21' : y
30  input prompt 'Enter a screen column number: ', length 2, &
        valid 'number; allow 1 to 80' : x
40  print at y, x, bold : 'X marks the spot'
50  end
```

```
.screen -
 Enter a screen row number: 9
 Enter a screen column number: 50
```

```
                                                X marks the spot
```

The **NUMBER** validation rule is used in the above example. It is further qualified with a range, as indicated by the **ALLOW** clause.

❼ **Use the TIMEOUT option to limit response time**

Setting a specified response time keeps an application from getting stuck on one line of input code. The **TIMEOUT** option is used to set the response time. The time limit is a real number interpreted in seconds. If the specified number of seconds pass without any data entry, INTOUCH displays the error message, **Timeout at line nn**, and generates an exception.

❽ **Use the ELAPSED option to determine response time**

There might be instances when you need to determine the user's response time. The **ELAPSED** option can be used to tell how long a user takes to enter a data item. The clock starts running after the input prompt has been displayed on the screen and stops when all of the input variables have values.

This option is especially useful for applications such as a typing tutorial.

2.2 Using Fill-in Fields

There are other ways to input data which assist the user and also provide some input error control. Fill-in fields can be used to show the user what data to input and also cut down on input keystrokes. Users are partial to the type of input which appeals to the eye and involves little effort on their part.

INTOUCH offers two INPUT options to create fill-in fields—the **LENGTH** and the **SCREEN** options. The **LENGTH** option displays the length of the input field. The **SCREEN** option offers other formatting capabilities.

Example

The following example shows some fill-in formats.

```
1   program application_2_3
10  clear
    print at 1, 1:;
20  attr$ = 'BOLD'
    input prompt '1 - The current year is ', attributes attr$, length 4 : answer$
    print
30  ask paint old_paint_char$
    set paint '*'
    input prompt '2 - Next year will be ', length 4 : answer$
    print
40  set paint old_paint_char$
    input screen '3 - Social Security No.: <aj, digits:###-##-####>' : ss_number
    print
50  set paint ' '
    input screen '4 - Password: <ucase, req, noecho:@@@@@@@@@@@@>' : answer$
    print '    Your password was '; answer$
60  set paint old_paint_char$
70  end
```

```
.screen -
❶ - The current year is ____

❷ - Next year will be ****

❸ - Social Security No.: ___-__-____

❹ - Password:
    Your password was PANDA
```

❶ Use the LENGTH option to control input length

The **LENGTH** option controls how many characters or digits can be entered by displaying the length of the input with the paint character "_". As a character or digit is entered, the paint character is wiped out. INTOUCH will beep if the user attempts to type in more than the specified amount of paint characters.

The input data is further enhanced by using the **ATTRIBUTES** option. This caused the input to be shown as bold characters and distinguishes the input data from from the background data.

❷ **Controlling the PAINT character**

INTOUCH allows you to control the paint character, the "_" character, with the **ASK PAINT** and **SET PAINT** statements. The default paint character is the underscore. The **ASK PAINT** statement will tell you what the current paint character is. To change the paint character from the current value, use the **SET PAINT** statement. You can set the paint character to whatever you want it to be. In example #1, the paint character was "_" and in #2, it was changed to "*".

❸ **Use the SCREEN option to format the input**

If you want to do more field formatting (i.e. digits, upper case, etc.), you can use the **SCREEN** option. This allows you to simulate fill-in fields by providing screen masks. The data can also be filtered using screen format options (i.e. UCASE, LCASE, NOECHO, DIGITS, AJ, REQ, AT) and/or screen format characters (i.e. #, @, ., ^). For details on each screen format option and format character, please refer to the INPUT chapter in the *INTOUCH - A Guide to the Language* manual.

Example #3 uses a *screen mask* for the social security number, allows only *digits* and *automatically jumps* to the next prompt.

❹ Example #4 uses *upper case* to convert lower case to upper case, *requires* the user to input some data and the *noecho* suppresses the printing of the input to the screen. The @ specifies any printable character. This is a good example for applications involving passwords.

2.3 Using Free Formatted Areas

Some applications input data that is informative text (a block of text) or perhaps descriptive of another data field. For example, you might want the user to enter some general information about a customer.

INTOUCH offers a free form type of data entry field which is used with the **AREA** option of the INPUT statement.

The **AREA** option allows the INPUT statement to accept multiple lines of text, as defined by the upper left and lower right coordinates of a rectangular area.

You can also change the rectangular block size using the INTOUCH **WRAP$( )** function. This allows you to resize blocks of text.

Example

The following example shows how to input a block of text and then to resize the text block. The user types in data for the length of the paint character string. INTOUCH automatically goes to the next line when the previous line is full.

```
1   program application_2_4
10  clear
    message 'Press GOLD/F when done'
20  line input area 5, 1, 8, 50: text$
    print at 11, 1: &
        'The text has been resized from 50 to 30 character lines'
    print
30  resized_text$ = wrap$(text$, 1, 30)
    print at 14, 1: resized_text$
40  end
```

The user is presented with the painted input area:

```
.screen -
  ____________________________________________________
  ____________________________________________________
  ____________________________________________________
  ____________________________________________________


                            Press GOLD/F when done
```

The text is entered and the user presses GOLD/F:

```
.screen -
  This customer has a home by the beach.  He likes _
  to swim, fish and surf.  He has a boat and often _
  takes it out on family trips._____________________
  ____________________________________________________

  The text has been resized from 50 to 30 character lines

  This customer has a home by
  the beach.  He likes to swim,
  fish and surf.  He has a boat
  and often takes it out on
  family trips.
```

3 Creating Menus

Menus are another form of input and are often favored by users because they require the least amount of effort. The user simply uses the arrow keys to select the menu item and then presses the Return key. INTOUCH makes working with menus quick and easy with the **MENU** option of the INPUT and LINE INPUT statements.

An INTOUCH menu is comprised of comma-separated elements. These elements can either be menu selection items (items on the menu the user selects from) or directives (menu presentation instructions). Each menu item can call a submenu and the submenu items can also call submenus. The directives can be used to customize your menus (i.e. give the menu a title, headings, screen placement, etc).

A menu can contain an unlimited number of items because the screen scrolls up and down between the items or left and right for a horizontal (bar) menu. Unless otherwise specified, INTOUCH automatically determines menu positioning on the screen, menu width and centering of all titles.

3.1 Menu Composition

A menu can consist of a single string literal containing directives and menu items, such as:

```
1   program application_3_1
10  clear
    the_menu$ = '%title "Animals", dog, cat, horse'
    input menu the_menu$ : answer$
20  end
```

```
.screen -
                +-Animals--+
                |  DOG     |
                |  CAT     |
                |  HORSE   |
                +----------+
```

Menus can consist of string variables which contain directives and menu items, such as:

```
10  the_title$ = '%title "The Animals",'
    the_items$ = 'dog, cat, horse'
20  the_menu$ = the_title$ + the_items$
    input menu the_menu$ : answer$
or
20  the_menu$ = the_title$ & the_items$
    input menu the_menu$ : answer$
```

The **+** sign and the **&** are interchangeable. The elements *must* be separated by commas.

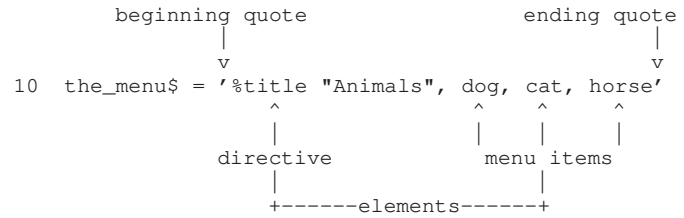
Menus can consist of directives and string variables and literals, such as:

```
10  the_items$ = 'dog, cat, horse,'
    the_menu$ = '%title "Animals",' + the_items$ + 'mouse'
    input menu the_menu$ : answer$
```

Menus can also consist of record fields and input data. An example using record data is included in the menu example section.

3.2 Hints for Creating Menus

A menu or part of a menu that consists of a string literal, is enclosed within quotes. Either the single quote (') or the double quote mark (") can be used. The menu elements (directives and items) are separated by commas. For example:



If the menu items contain spaces or special characters, they MUST be enclosed within quotes. For example:

```
10  the_menu$ = '"the dog", "the cat", "the horse"'
```

INTOUCH will UPPERCASE menu items that are *not* enclosed in quotes. If your menu items contain upper and lower case characters and you want them displayed that way, you will need to enclose them in quotes.

The menu directives are just another element in the menu string. They are preceeded by a "%" and followed by a comma. For example:

```
10  the_menu$ = '%title "Animals", "the dog", "the cat", "the horse"'
```

If the menu is longer than one line, you can continue on the next line in this way:

```
10  the_menu$ = '%title "Animals", "the dog", "the cat", "the horse"' &
              + '"the cow", "the pig"'
```

or in this way:

```
10  the_menu$ = '%title "Animals", "the dog", "the cat", "the horse"' + &
              '"the cow", "the pig"'
```

Use **INPUT MENU** if your menu *item selections* contain one word elements (no spaces or special characters) otherwise, use **LINE INPUT MENU**. For example:

```
10  the_menu$ = 'dog, cat, horse'
20  input menu the_menu$ : answer$
```

```
10  the_menu$ = '"the dog", "the cat", "the horse"'
20  line input menu the_menu$ : answer$
```

If your menu contains an "EXIT", "HELP" or other system function element and the user selects this element, **_EXIT**, **_HELP**, etc. are set to *TRUE* and the input variable (i.e. answer$) will be a null.

3.3 Using Menus

Using INTOUCH menus is similar to other applications in that the Return key, ARROW keys and BACKSLASH (\) key are used. To select menu items:

- Use the UP and DOWN ARROW keys to select an item from a vertical menu and then press the Return key to execute the selected item.
- Use the LEFT and RIGHT ARROW keys to select an item from a horizontal (bar) menu and then press the Return key to execute the selected item.
- In some cases, you will need to move the ARROW keys both up and down *and* across to select a menu item.
- To select multiple items from a menu, use the ARROW keys to select an item and then press the Select key, arrow to the next item and press Select, etc. After all the selections have been made, press the Return key. Pressing the SPACE BAR instead of the Select key will also select the item.
- You can also select a menu item by typing in the first or several characters of the menu item. The highlight bar will immediately move to the first occurrence of that item. Any character(s) typed in after that further specifies the selection.
- Use the \ (backslash) to get out of the menu and backup to a previous menu, submenu or out of the program.

3.4 Menu Examples

The following menu examples will show how to create simple to more complex menus which can be used for different applications. The directives will be described as they are used.

Example 1

This is an example of a common type of menu that would be created for user input.

```
1   program application_3_2
10  clear
20  sort_selections$ = '%title "Sort Selections", "Account",' &
        + '"Division", "Name", "State"'
30  input menu sort_selections$: selection$
40  print 'The menu selection was:  '; selection$
50  end
```

```
.screen -

          +Sort~Selections+
          |  Account      |
          | ~Division~    |
          |  Name         |
          |  State        |
          +---------------+

The menu selection was:  Division
```

The directive used in this menu example was:

%title—gives the menu a title

Example 2

This is an example of a master or main menu which is used as a starting point in many applications.

```
1   program application_3_3
10  clear
20  master_menu$ = '%at 2, 10, %title "Master Menu",' &
        + '%heading "Procedures","Add New Customers",' &
        + '"Change Customer Data",' &
        + '"Delete Customers","Customer Inquiry",' &
        + '%heading "Other Options","Mail","Help","EXIT"'
30  line input menu master_menu$: selection$
40  print at 21,1 : 'The menu selection was:  '; selection$
50  end
```

```
.screen -
         +-------Master~Menu-------+
         |                         |
         |~~~~~~~Procedures~~~~~~~~|
         |  Add New Customers      |
         | ~Change~Customer~Data~  |
         |  Delete Customers       |
         |  Customer Inquiry       |
         |                         |
         |~~~~~~Other~Options~~~~~~|
         |  Mail                   |
         |  Help                   |
         |  EXIT                   |
         +-------------------------+

The menu selection was:  Change Customer Data
```

The directives used in this menu example are:

%at—positions the menu on the screen
%title—gives the menu a title
%heading—gives a menu section a heading

Example 3

This is an example of a menu with submenus and a submenu which has a submenu.

```
1   program application_3_4
10  clear
20  customer_data$ = '%title "Customer Data", "Account",' &
        + '"Name" = {"First", "Middle", "Last"},' &
        + '"Address" = {"Street", "City", "State", "Zip Code",' &
        + '"Country" = {"United States", "Foreign"}},' &
        + '"Phone", "Extension", "Division", "Cost Center", "Contact"'
30  line input menu customer_data$: selection$
40  print at 21,1 : 'The menu selection was:  '; selection$
50  end
```

The first menu is displayed. The **[>** to the right of the menu item indicates that there are options for this menu item.

```
.screen -
  +-Customer~Data--+
  |  Account       |
  |  Name        [>|
  |  Address     [>|
  |  Phone         |
  |  Extension     |
  |  Division      |
  |  Cost Center   |
  |  Contact       |
  +----------------+
```

If the user selects the **Name** item, the following is displayed:

```
.screen -
  +-Customer Data--+
  |  Account     +---Name----+
  | ~Name~       |  First    |
  |  Address     | ~Middle~  |
  |  Phone       |  Last     |
  |  Extension   +-----------+
  |  Division      |
  |  Cost Center   |
  |  Contact       |
  +----------------+

The menu selection was:  Middle
```

If the user selects the **Address** item and the **Country** item, the following is displayed:

```
.screen -
  +-Customer Data--+
  |  Account       |
  |  Name        +---Address---+
  | ~Address~    |  Street     |
  |  Phone       |  City       |
  |  Extension   |  State      |
  |  Division    |  Zip Code +-----Country------+
  |  Cost Center | ~Country~ |  United States   |
  |  Contact     +-----------| ~Foreign~        |
  +----------------+         +------------------+

The menu selection was:  Foreign
```

The directives used in this menu example are:

%title—gives the menu a title
{}—indicates submenu elements

Example 4

This is an example of a horizontal menu with pull-down submenus.

```
1   program application_3_5
10  clear
20  customer_data$ = '%menubar, %autovbar on, %autodisplay off,' &
        + '"Account",' &
        + '"Name" = {%title "", "First", "Middle", "Last"},' &
        + '"Address" = {%title "", "Street", "City", "State",' &
        + ' "Zip Code",' &
        + '"Country" = {"United States", "Foreign"}},' &
        + '"Phone", "Extension", "Division", "Cost Center", "Contact"'
30  line input menu customer_data$: selection$
    print at 21,1 : 'The menu selection was:  '; selection$
40  end
```

When the program is run, the bar menu is displayed:

```
.screen -
+-------------------------------------------------------------------------------+
|~ Account ~~|~~~Name~~~|~~~Address~~~|~~~Phone~~~|~~~Extension~~~~~~~~~~~~~~>>|
+-------------------------------------------------------------------------------+
```

The **>>** indicates that there are more menu items but there was not room to display them on the screen. If you arrow to the right to display the items, there will be **<<** on the left side.

If you select the "Address" item, the address submenu is displayed. If "Country" is selected, the country submenu is displayed.

```
.screen -
+-------------------------------------------------------------------------------+
|~~Account~~~|~~~Name~~~|~~ Address ~~|~~~Phone~~~|~~~Extension~~~~~~~~~~~~~~>>|
+--------------------------+-------------+-------------------------------------+
                           |~~Street~~~~~|
                           |~~City~~~~~~~|
                           |~~State~~~~~~|
                           |~~Zip~Code~+-----Country------+
                           |~ Country ~|~~United~States~~~|
                           +-----------|~ Foreign ~~~~~~~~|
                                       +------------------+

The menu selection was:  Foreign
```

The directives used in this menu example are:

%menubar—creates a menu bar; the menu items are listed horizontally and submenus are pull-down

%autovbar on|off—if **on**, a vertical bar separates the menu items; the default is **off**

%autodisplay off—disables the automatic display of the submenus

{}—indicates submenu elements

The **%title** option with no title was used to suppress the title in the pull-down menus.

In this example the automatic display of the pull-down submenus was turned off. This requires a user to enter more keystrokes than if the submenus are displayed automatically.

Example 5

In this example, the first menu is displayed. After the user makes a selection, the first menu is erased and a submenu is displayed. The user is able to select multiple items from the submenu. To select multiple items, use the arrow keys to make a selection and then press the Select key or the SPACE BAR. This can be done repeatedly. After all the items have been selected, press the Return key.

The submenus are examples of menu items listed in columns.

```
1   program application_3_6
10  clear
20  domestic$ = 'DOMESTIC = {%replace, %columns 3, %width 15,' &
        + '%at 4, 10, %multi, %title "SELECT DOMESTIC CARS", BLAZER,' &
        + 'BRONCO, BUICK, CADILLAC, CORVETTE, LINCOLN, MUSTANG,' &
        + 'MERCURY, THUNDERBIRD},'
30  foreign$ = 'FOREIGN = {%replace, %columns 2, %width 15,' &
        + '%at 4, 10, %multi, %title "SELECT FOREIGN CARS", ACURA,' &
        + 'HONDA, INFINITY, LEXUS, MAZDA, MITSUBISHI, NISSAN, SUZUKI,' &
        + 'SUBARU, TOYOTA, %bar, HYUNDAI, YUGO, %bar, AUDI,' &
        + 'VOLKSWAGON}'
40  car_menu$ = '%title "Car Category",' & domestic$ & foreign$
50  input menu car_menu$: selection$
60  print 'Number of selections: '; pieces(selection$, chr$(10))
    print 'The selected items are: '; selection$
70  end
```

The first screen is displayed.

```
.screen -

    +Car~Category-+
    |  DOMESTIC ..|
    |  FOREIGN... |
    +-------------+
```

If the user selects **DOMESTIC**, the following menu will be displayed. Whatever items are selected by the user will be underlined.

```
.screen -

         +---------------SELECT~DOMESTIC~CARS---------------+
         | ~BLAZER~       |   CADILLAC      |   MUSTANG      |
         |  BRONCO        |  ~CORVETTE~     |   MERCURY      |
         |  BUICK         |   LINCOLN       |  ~THUNDERBIRD~ |
         +-------------------------------------------------+

Number of selections:  3
The selected items are: BLAZER
                              CORVETTE
                                      THUNDERBIRD
```

If the user selects **FOREIGN**, the following menu will be displayed. Whatever items are selected by the user will be underlined.

```
.screen -
          +------SELECT~FOREIGN~CARS-------+
          |  ACURA          |  ~SUBARU~     |
          |  HONDA          |   TOYOTA      |
          | ~INFINITY~      | --------------|
          |  LEXUS          |  ~HYUNDAI~    |
          |  MAZDA          |   YUGO        |
          |  MITSUBISHI     | --------------|
          |  NISSAN         |   AUDI        |
          |  SUZUKI         |  ~VOLKSWAGON~ |
          +---------------------------------+

Number of selections:  4
The selected items are: INFINITY
                                SUBARU
                                      HYUNDAI
                                             VOLKSWAGON
```

The directives used in this menu example are:

%replace—erases the calling menu; pressing "\" will reprint the calling menu
%columns n—creates a multi-column menu
%width n—sets the width of the menu
%at n,n—positions the menu on the screen
%multi—allows multiple menu items to be selected
%title—gives the menu a title
%bar—used to separate menu items

Example 6

This example shows how to create a menu using field data from a file. In this case, the data in the CLIENT file becomes the menu item selections.

```
1   program application_3_7
10  clear
    the_menu$ = '%title "Select a Client", %at 3,20'
20  open structure cl : name 'tti_run:client'
    extract structure cl
      sort by cl(last)
    end extract
    for each cl
      the_menu$ = the_menu$ + ',' + cl(last)
    next cl
    close structure cl
30  input menu the_menu$ : answer$
    print 'The menu selection was:  '; answer$
40  end
```

The program creates the menu items using the last names in the CLIENT file records.

```
.screen -

                    +Select~a~Client+
                    |  ABOTT         |
                    |  BROCK         |
                    |  CASS          |
                    |  DERRINGER     |
                    |  DONALDSON     |
                    |  ERRANT        |
                    |  FARMER        |
                    | ~JOHNSON~      |
                    |  JONES         |
                    |  KENT          |
                    |  PORTER        |
                    |  RODRIGUES     |
                    |  SMITH         |
                    |  WATERS        |
                    +----------------+

The menu selection was:  JOHNSON
```

The directives used in this menu example are:

%at n,n—positions the menu on the screen
%title—gives the menu a title

3.5 Menu Directives

The menu examples used some of the menu directives. The following is a list of all the available menu directives. The *INTOUCH - Guide to the Language* manual explains each of these directives in detail.

Table 3–1 Menu Directives

%at row,column	positions the menu
%autodisplay off	disables the automatic display of submenus
%autovbar on | off	ON prints a vertical bar between each of the menu items in a bar menu; OFF prints no vertical bars; the default is OFF
%bar ["..."]	used to separate menu items; if no text is supplied, a dashed line will be displayed
%columns n	sets the number of columns for the menu
%heading ["..."]	displays a blank line *and* a text string between menu items; if no text is provided, a blank line and a dashed line are displayed
%items n	creates a multi-column menu
%lockstep on | off	controls scrolling for multi-columned menus; normally the columns scroll together, OFF causes the columns to scroll independently; the default is ON
%maxwidth n	controls the maximum width of the entire menu
%menubar	displays the menu horizontally; this is a bar menu

Table 3–1 (Cont.) Menu Directives

%multi	allows multiple menu items to be selected
%replace	used in submenus to erase the calling menu; \ will redisplay the calling menu
%size n	determines the number of menu items in the menu box
%split	starts a new column at this point in the menu
%title "..."	gives the menu or submenu a title
%vbar	used to insert a vertical bar between two menu items in a bar menu
%width n	controls the minimum width of the current column

3.6 Defaulting to a Menu Item

You might want to display the menu to the user with the highlight bar already positioned on an often used menu item. If the user wants that item, they can simply press Return. Otherwise, they can backup to previous menu(s) or use the arrows keys to select another menu item.

INTOUCH allows you to set up a default menu item by using the **DEFAULT** option of INPUT.

Example

In the following example, the default is the fourth item of the first menu and the second item of the submenu. The default item can be as deep as there are existing menu levels.

```
1   program application_3_8
10  clear
20  sample_menu$ = '%width 12, %menubar, %autovbar on,' &
        + 'add, change, delete, inquire = {product, customer, sales}'
30  input menu sample_menu$, default '#4;#2': selection$
    print 'Default was CUSTOMER, selected menu item was:  '; selection$
40  end
```

```
.screen -
+-------------------------------------------------------------------------------+
|~~ADD~~~~~~~|~~~CHANGE~~~~|~~~DELETE~~~~|~~ INQUIRE ~~~~~~~~~~~~~~~~~~~~~~~~~|
+-------------------------------------------+---INQUIRE---+---------------------+
                                            |~~PRODUCT~~~~|
                                            |~ CUSTOMER ~~|
                                            |~~SALES~~~~~~|
                                            +-------------+


Default was CUSTOMER, selected menu item was:  CUSTOMER
```

3.7 Returning to a Menu Item

After a user selects a menu item, some processing is performed. When the the processing steps are completed, the menu can be redisplayed and the user can make the next selection. To get back to the menu and display the previous selection, you use the **DEFAULT** option with **_string** qualifier. **_string** contains the last menu selection.

If you are doing more than one menu sequence, you will need to save earlier **_string** values in string variables. These variables can then be used with the **DEFAULT** option to take you back into a particular menu sequence at any given time.

Example

The following example shows how to display a menu, do some processing and return to the previously selected menu option.

```
1   program application_3_9
10  clear
20  sample_menu$ = '%width 12, %menubar, %autovbar on,' &
        + 'add, change, delete, inquire, exit'
30  input menu sample_menu$ : selection$
40  do
      clear
      if  _exit  then  exit do
      print 'The menu selection was:  '; selection$
      print 'You will now do some '; selection$; ' processing....'
      print
      print 'When you press RETURN, the previous menu and selection '; &
            'will be displayed.  You can select another menu option'; &
            'or select the EXIT option to end this program'
      print
      input prompt '': z$
      clear
      input menu sample_menu$, default _string: selection$
    loop
50  end
```

The menu is displayed:

```
.screen -
+-------------------------------------------------------------------------------+
|~ ADD ~~~~~~|~~~CHANGE~~~~|~~~DELETE~~~~|~~~INQUIRE~~~|~~~EXIT~~~~~~~~~~~~~~~~|
+-------------------------------------------------------------------------------+
```

If the user selects the CHANGE item, this message information is displayed:

```
.screen -

The menu selection was:  CHANGE
You will now do some CHANGE processing....

When you press RETURN, the previous menu and selection
will be displayed.  You can select another menu option
or select the EXIT option to end this program
```

When the user presses Return, the menu will be redisplayed and will show the previous menu selection.

```
.screen -

+-------------------------------------------------------------------------------+
|~~ADD~~~~~~~|~~ CHANGE ~~~|~~~DELETE~~~~|~~~INQUIRE~~~|~~~EXIT~~~~~~~~~~~~~~~|
+-------------------------------------------------------------------------------+
```

3.8 Assigning Values to Menu Items

When a user selects a menu item, the selected menu item is stored in the specified variable, i.e. SELECTION$. For example, if the user selected the menu item "Add New Customers", SELECTION$ would contain "Add New Customers". To make SELECTION$ contain something shorter, such as the word "ADD", you can assign a value to the menu item.

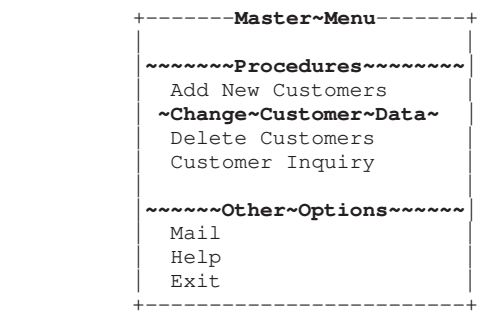
To assign a value to the menu item, add an equal sign (=) and a value to the menu item element.

Example

The following example shows you how to assign values.

```
1   program application_3_10
10  clear
20  master_menu$ = '%at 2, 10, %title "Master Menu",' &
        + '%heading "Procedures","Add New Customers" = ADD,' &
        + '"Change Customer Data" = CHA,' &
        + '"Delete Customers" = DEL, "Customer Inquiry" = INQ,' &
        + '%heading "Other Options","Mail" = MAIL,' &
        + '"Help", "Exit"'
30  line input menu master_menu$: selection$
40  print at 21,1 : 'The menu selection was:  '; selection$
50  end
```

```
.screen -
```




```
The menu selection was:  CHA
```

4 Using Messages and Delays

Messages can be used to give the user specific input instructions, highlight input errors and provide information. Messages can be used in place of print statements.

INTOUCH messages are displayed in bold and centered one line above the bottom line of the screen (i.e. line 23 of a 24-line screen). If the INTOUCH screen frame is on, the message is embedded in the bottom portion of the screen frame.

A message can be up to 80 characters in length.

Note

The examples in this chapter are run with the INTOUCH screen frame ON.

4.1 Message Types

There are basically three types of messages and three INTOUCH statements to display the messages. The INTOUCH statements are:

1 **MESSAGE 'message text'**

This message statement displays a message in bold and centered one line above the bottom screen line (i.e. line 23 of a 24-line screen).

2 **MESSAGE ERROR: 'message text'**

The **ERROR** option rings the device's bell, clears the type ahead buffer and sets the **_ERROR** internal variable to *TRUE*. It also ensures a delay of at least three seconds, to allow the user to read the error message.

3 **MESSAGE DELAY: 'message text'**

The **DELAY** option causes INTOUCH to set an automatic delay, giving the user time to read the message. Starting with a minimum delay of approximately three seconds, INTOUCH increases the delay for lengthier messages.

Example

The following example illustrates the three ways to use messages.

```
1   program application_4_1
10  clear
    print at 2, 20, bold, reverse: 'Examples of Message Usage'
    print
20  message 'Enter the date in the format YYYYMMDD'
    do
      input 'Current date', at 4,1: answer$
      if  answer$ = date$  then
        message delay: 'That is the correct date'
        exit do
      else
        message error: 'Incorrect date'
        clear area 4,1,4,80
        repeat do
      end if
    loop
30  message ''
    clear
40  end
```

This is an example of an instructional message:

```
.screen -
                    Examples~of~Message~Usage
Current date?




~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~ Enter the date in the format YYYYMMDD ~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

This is an example of an error message:

```
.screen -
                    Examples~of~Message~Usage
Current date? 19940211




~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Incorrect date ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

This is an example of a delayed message:

```
.screen -
                    Examples~of~Message~Usage
Current date? 19940513




~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~ That is the correct date ~~~~~~~~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

4.2 Controlling Message Placement

Depending on the type of terminal, the number of screen lines or rows can vary. For example, VT style terminals usually have 24 screen lines on a page, but other types of terminals might have more or less than 24. On some terminals, you can even set the number of screen lines per page. To find out what the page size is, you can use the **ASK PAGESIZE** statement.

- **ASK PAGESIZE num_var**

  This statement returns the number of lines (rows) of the terminal device.

If you do not want to display messages on the normal message line, INTOUCH allows you to control message placement by using the **ASK MESSAGELINE** and **SET MESSAGELINE** statements.

- **ASK MESSAGELINE num_expr**

  This statement returns the line number of where the messages are currently being displayed.

- **SET MESSAGELINE num_expr**

  This statement sets the message line to a specified number. The number can be between 1 and the total screen lines.

- **SET MESSAGELINE 0**

  Setting the message line to zero suppresses message display.

Example

This example shows you how to change the message line number.

```
1   program application_4_2
10  clear
    print at 1,1:;
    ask system: user uname$
    ask pagesize lines
    message 'Hello...'; uname$
    delay 2
20  ask messageline orig_mess_line
30  do
      print at 2,1: 'Total screen lines (page size) is'; lines
      input 'Enter a new message line number from 1 to total lines', &
            at 4, 1: new_line
      if  _back or _exit or new_line = 0  then  exit do
      clear
      set messageline new_line
      message 'The messages will now be displayed on line '; new_line
      delay 3
      message 'Enter another number or enter EXIT to exit'
    loop
40  message 'The original message line will reset in a few seconds...'
    delay 3
    set messageline orig_mess_line
    message 'The message line has been reset to its original position'
50  end
```

```
.screen -
Total screen lines (page size) is 24

Enter a new message line number from 1 to total lines? 12


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Hello...FRED ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

```
.screen -
Total screen lines (page size) is 24

Enter a new message line number from 1 to total lines?


(line 12)           Enter another number or enter EXIT to exit
```

4.3 Using DELAY

There are times when you want to delay a program to make the user aware of specific actions or display some pertinent information. You can use the **DELAY** statement to stop the program and allow the user to continue when they are ready.

Example

```
       1   program application_4_3
       10  clear
       20  print bold, at 3,15 : 'SUMMARY TOTALS'
           print at 5,5 : 'Beginning Balance:  245,234.69'
           print at 7,5 : 'Activity Today   :   89,267.05'
           print at 9,5 : 'Ending Balance   :  334,501.74'
           print at 12,5 : 'Orders Processed :  68'
           print bold, at 16,20 : 'Sales Files will now be updated...'
       30  delay
           clear
       40  end
```

```
.screen -

              SUMMARY TOTALS

    Beginning Balance:  245,234.69

    Activity Today   :   89,267.05

    Ending Balance   :  334,501.74

    Orders Processed :  68

                  Sales Files will now be updated...

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~ Press the RETURN key to continue ~~~~~~~~~~~~~~~~~~~~~~
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

5 Checking Input Responses

Processing a Response

When a user responds to a prompt, they usually type in some data or select a menu item and then press the Return key to signal the end of their response. However, the user might, instead, press a specific key (i.e. \, Help key, etc.) or keys (i.e. Ctrl/Z) and that is their response. As you can see, an application routine must first determine if the user actually entered some data or requested a specific action and then, either validate the data or perform the action. To assist in deciphering a response, INTOUCH provides numerous built-in functions which analyze the response and make it easy for the programmer to determine what a user entered.

INTOUCH processes a response as two separate parts. One part is the *reply* (i.e typed data or nothing) and the other part is the *terminator* (\, Return, Help, etc.). INTOUCH stores the reply and the terminator in the internal variables **_REPLY** (see Section 5.1) and **_TERMINATOR** (see Section 5.2). INTOUCH also analyzes the terminator and, depending on its contents, sets other internal variables to TRUE or FALSE. The programmer can then check the internal variables and take the appropriate action. For example, if the internal variable **_EXIT** is TRUE, you know the user either entered "EXIT" or Ctrl/Z.

This chapter contains information on some of the commonly used INTOUCH system functions and internal variables. The appendix section of the *INTOUCH - A Guide to the Language* manual contains information on all the system and other built-in functions and internal variables.

Concepts and Key Values

In INTOUCH, the concepts of backing up, exiting, getting help, etc. are assigned to specific keys. For example, if you are at the VMS $ prompt and press the \ (backslash) key, nothing happens. But, if you are in an INTOUCH program and you press the \ key, you will back up to a previous prompt or menu. This occurs because INTOUCH has assigned the concept of backing up as the default value for the \ key.

When you are in INTOUCH, you can change the current key values to other values or no value at all (see Section 5.4). For example, the \ key does not have to default to backing up. You can change the value of the "B" key or any other key to default to backing up and the \ key can be changed to another value or no value at all.

5.1 Checking _REPLY

One of INTOUCH's system functions always stores the **last** user entered input (in string format) in **_REPLY**. This internal variable provides you with a generic place to store the user's last reply, and it becomes specially useful when debugging.

The keystroke that terminated the last user input is stored in the internal variable, **_TERMINATOR**.

INTOUCH processes a response in this manner:

Entered response	**_REPLY** contains	**_TERMINATOR** contains
SMITH (Return key)	SMITH	RETURN
Smith (Return key)	Smith	RETURN
25.00 (Return key)	25.00	RETURN
(Return key)	null value	RETURN
(Help key)	null value	HELP
25.00\	null value	\
25.00 (Ctrl/Z)	null value	CTRL/Z

Example

```
1   program application_5_1
10  clear
20  line input 'Add this customer record (Y/N)': answer$
    print
    print 'The reply was     :  '; _reply
    print 'The terminator was:  '; _terminator
    print
30  line input 'Type your name and then press the "\" key': answer$
    print
    print 'The reply was     :  '; _reply
    print 'The terminator was:  '; _terminator
40  end
```

```
.screen -
Add this customer record (Y/N)? yes

The reply was     :  yes
The terminator was:  RETURN

Type your name and then press the "\" key? tester\

The reply was     :
The terminator was:  \
```

5.2 Checking the _TERMINATOR

The concept of terminating input defaults to various (normally) non-character keys such as the Return key, arrow keys, \, Help key, etc. For example, when a user responds to a prompt, they either enter some data and press Return or they enter \ to back up, Help to get help, etc. Whatever keystroke terminates the input is the input terminator and INTOUCH stores this terminator in the **_TERMINATOR** internal variable.

You can check the **_TERMINATOR** variable to find out what keystroke terminated the last input. You can also change any of the default terminating keys to other keys. (See Section 5.4.)

The following is just a few of the non-character keys that can be checked. A complete list can be found in the *INTOUCH - A Guide to the Language* manual.

- Return key = **RETURN**
- Tab key = **TAB**
- UP arrow key = **UP**
- DOWN arrow key = **DOWN**
- Help key = **HELP**
- PF1..PF4 = **PF1** thru **PF4**

Example

```
1   program application_5_2
10  clear
20  input prompt 'Press the up arrow, down arrow or RETURN key': answer$
30  print
    if  _terminator = 'UP'  or  _terminator = 'DOWN'  then
      print 'The '; _terminator; ' arrow key was pressed'
    else
      print 'The '; _terminator; ' key was pressed'
    end if
40  end
```

```
.screen -
Press the up arrow, down arrow or RETURN key

The DOWN arrow key was pressed
```

5.3 Checking for _BACK and _EXIT

You can determine if a user has backed up or exited in an INTOUCH application by checking the **_BACK** and **_EXIT** internal variables.

- In INTOUCH, the concept of backing up defaults to the **** (backslash) and UP arrow keys. When either of these keys is pressed, the internal variable **_BACK** is set to *TRUE* (i.e. 1).

  Your application can check **_BACK** for a *TRUE* value and return the user to a previous prompt, menu, etc.

- INTOUCH has adopted the VMS concept of the Ctrl/Z key. This keystroke combination or the user entered keyword **EXIT**, set the INTOUCH internal variable, **_EXIT**, to *TRUE* (i.e. 1).

  Your application can check **_EXIT** for a *TRUE* value and exit the user out of a routine, program, etc.

Example

```
1   program application_5_3
10  clear
20  do
      print
      input 'What is the current month': answer$
      if  _back or _exit  then  exit do
      print
      input 'What is the current year': answer$
      if  _back  then  repeat do
      if  _exit  then  exit do
    loop
30  print
    print 'The End'
40  end
```

```
.screen -
What is the current month? May
What is the current year? \
What is the current month? June
What is the current year? exit
The End
```

5.4 Changing Key Values

When you are in INTOUCH, you can change any of the current keystroke concepts and values to other concepts and values. For example, you could change the "A" key to the value of "1" or to no value at all. You could also change "A" to the ASCII value CHR$(127) and the concept of "DEL".

Before you change any of the key values, you normally save the **keyboard map**. The keyboard map is a collection of the *current* keystroke concepts and values. The **ASK/SET WINDOW: KEYMAP** statements are used to save and restore keyboard maps.

- **ASK WINDOW: KEYMAP str_expr**

  This statement stores the current keyboard map (current key values and concepts) in the specified variable.

- **SET WINDOW KEYSTROKE str_expr1 : value str_extr2**

  This statement assigns a new value and/or concept to one or more specified keys. You can change the character (A, B, C, etc.) keys and also the non-character keys such as DEL (delete), SP (space), LEFT, RIGHT, UP and DOWN (arrow keys), etc.

  More information and a table of supported concepts can be found in the *INTOUCH - A Guide to the Language* manual.

- **SET WINDOW: KEYMAP str_expr**

  This statement restores the specified keyboard map. If str_expr is a null (''), the original default key values and concepts will be restored.

Note

Changes to the keyboard map are only effective while in the INTOUCH environment.

Example

This example shows how to change key values and concepts. The values of some characters and digits are changed and then the original values are restored. The **_IGNORE** internal variable can be used to give keys no value at all.

```
1   program application_5_4
10  clear
    print at 1,1:;
20  input 'What is your Name': answer$
    input 'What is your Social Security No': answer$
    print
30  set window keystroke 'a,e,i,o,u,y': value '*'
    set window keystroke '0,2,4,6,8': value '_ignore'
    set window keystroke 'q': value '_exit'
40  message 'a,e,i,o,u,y are changed to "*", 0,2,4,6,8 will be ignored'
    line input 'What is your Name': answer$
    line input 'What is your Social Security No': answer$
    message 'The value of the "Q" key has been changed to EXIT'
    line input 'Press the "Q" key': answer$
    print
50  ask window: keymap save_map$
    set window: keymap ''
    message 'The changed key values have been saved and the original ' &
                + 'values restored'
    line input 'What is your Name': answer$
    line input 'What is your Social Security No': answer$
    line input 'Press the "Q" key and the "Return" key': answer$
60  set window: keymap save_map$
    message 'The "CHANGED" key values have been restored'
    print
    line input 'What is your Name': answer$
    line input 'What is your Social Security No': answer$
    line input 'Press the "Q" key': answer$
70  set window: keymap ''
    message 'The original key values have been restored'
80  end
```

```
.screen -
What is your Name? sandy
What is your Social Security No? 123456789

What is your Name? s*nd*
What is your Social Security No? 13579
Press the "Q" key? EXIT

What is your Name? sandy
What is your Social Security No? 123456789
Press the "Q" key and the "Return" key? q

What is your Name? s*nd*
What is your Social Security No? 13579
Press the "Q" key? EXIT
```

```
                The original key values have been restored
```

5.5 Supported Keys, Keystroke Combinations

INTOUCH supports several keystroke combinations, which bear special meaning and are valuable when creating applications around user input.

- Ctrl/Z

  INTOUCH adopted the VMS interpretation of Ctrl/Z. This keystroke combination or the keyword **EXIT**, both set the INTOUCH internal variable, **_EXIT**, to *TRUE* (i.e. 1).

- Help **key,** PF2 **key**

  INTOUCH adopted the VMS interpretation of the Help key. This key or the keyword **HELP** both set the INTOUCH internal variable, **_HELP**, to *TRUE* (i.e. 1).

6 Working with INTOUCH Structures

INTOUCH **STRUCTURES** are made up of **records** *and* **fields**. Each record consists of one or more fields. The INTOUCH data structure statements allow you to manipulate data stored in the fields from within your programs.

An INTOUCH structure file has the extension, **STR** (file_name.STR), and contains:

- the name of the database engine (record management system).
- the name of the dataset (data file).
- the name of the data dictionary (definition file).

6.1 Creating Structures

INTOUCH structures are easy to create. At the INTOUCH prompt, type the words **CALL SETUP**.

```
.screen -
  INTOUCH

  CALL SETUP
```

This will put you into the INTOUCH **SETUP** routine where you can create structures and define fields. You can also modify existing structures and fields. You will be asked for:

- a structure name
- the database engine, e.g. RMS, Rdb, DBMS...
- the dataset name
- the data dictionary name

The SETUP routine then allows you to:

- define the fields in the record
- create a data file
- display field and structure information

Note

You can refer to the SETUP chapter in the "INTOUCH - A Guide to the Language" manual for detail information on structures and fields.

6.2 Opening and Closing Structures

Structures can be opened and closed. Depending on the access mode, structure records can be read, written out, changed and deleted.

The **OPEN STRUCTURE** statement opens a structure and the **CLOSE STRUCTURE** statement closes a structure. **CLOSE ALL** will close any open structures and any other open files.

The OPEN STRUCTURE statement has several options (i.e. lock, datafile, etc.). You can refer to the *INTOUCH - Guide to the Language* manual for more information on OPEN and other structure statements.

```
open structure catalog: name 'catalog', access outin

open structure cust: name 'user:[tester]customer', access output

open structure cl: name 'tti_run:client'
               |             |              |
         structure name      |              |
                    file specification      |
                                      access mode
```

structure name is the name you will use in your application to refer to fields in the structure. This name can be the same as the file specification or a shortened version (i.e. nickname). It is best to use a name that relates to the file specification as opposed to using file_1, file_2, etc. so you can easily identify the file throughout your application.

file specification is the actual name of the data file. This name can include the device and directory or logical which tells where the data file is located.

access mode tells how the data file will be used. The access modes are **INPUT**, **OUTPUT** and **OUTIN**. If no access is specified, the default is INPUT.

6.3 Referencing Structure Data

After you have created your structure and defined the fields, you can reference the field data in the structure using structure references.

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

```
struc_name(field_expr)
```

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

```
CL(PHONE)          CATALOG(PART_NUM)
```

The field_expr can be either a string or numeric expression.

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

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

If you specify the field as an expression, you **must** precede the expression with a pound sign (#). The pound sign tells INTOUCH that the following characters are an expression, not the field name. If you do not include the pound sign, INTOUCH will interpret the characters as a field name.

Here are two examples:

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

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

The following examples show how to open and close a structure and how to use field names and field expressions.

Example A

If you know what fields you want to process, you can reference the data by the field names.

```
1   program application_6_1
10  clear
    open structure cl: name 'tti_run:client'
20  extract structure cl
      print cl(last), cl(first), cl(phone)
    end extract
30  close structure cl
40  end
```

```
.screen -
  Smith              Sam                (809) 555-8789
  Kent               Keith              (619) 967-5021
  Johnson            Paul               (619) 489-5551
  Waters             Wayne              (619) 564-1231
  Rodrigues          Homero             (   )    -   0
  Donaldson          George             (   )    -   0
  Errant             Earl               (408) 844-7676
  Abott              Al                 (202) 566-9892
  Brock              Bud                (218) 555-4322
  Cass               Cathy              (619) 743-8582
  Porter             Pete               (619) 778-6709
  Derringer          Dale               (818) 223-9014
  Farmer             Fred               (305) 552-7872
```

Example B

If you do not know what fields you want to process, you can reference the data using field expressions.

```
1   program application_6_2
10  open structure cl: name 'tti_run:client'
20  do
    clear
    print at 1, 1:;
      gosub ask_fields
      if  _back  or  _exit  then  exit do
      gosub show_data
    loop
    close structure cl
    stop

30  ask_fields:
    do
      if  _error  then  set error off
      print 'Field List:  '; &
              'ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE'
      print
      line input 'Select fields to display': field_list$
      if  _back  or  _exit  then exit do
      for f = 1 to elements(field_list$)
        field$ = element$(field_list$, f)
        ask structure cl, field #field$: number z
        if  z = 0  then
          message error: 'Illegal field: '; field$
        end if
      next f
40  loop while _error
    return
```

```
50  show_data:
    print
    extract structure cl
      for f = 1 to elements(field_list$)
        field$ = element$(field_list$, f)
        print cl(#field$),
      next f
      print
60  end extract
    delay
    return
70  end
```

```
.screen -
Field List:  ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE

Select fields to display? last,first,phone

Smith               Sam                 (809) 555-8789
Kent                Keith               (619) 967-5021
Johnson             Paul                (619) 489-5551
Waters              Wayne               (619) 564-1231
Rodrigues           Homero              (   )     -   0
Donaldson           George              (   )     -   0
Errant              Earl                (408) 844-7676
Abott               Al                  (202) 566-9892
Brock               Bud                 (218) 555-4322
Cass                Cathy               (619) 743-8582
Porter              Pete                (619) 778-6709
Derringer           Dale                (818) 223-9014
Farmer              Fred                (305) 552-7872


                    Press the RETURN key to continue
```

6.4 Examples Using Structures

Once you have located a specific data record in the structure, you can perform the processing steps. Locating the record or records can sometimes be complicated. This section gives you examples of how to use INTOUCH to locate structure records in different situations.

Example 1 - Common Extract and Sort

This example shows a common way to extract and sort records.

```
1   program application_6_3
10  clear
    print at 1, 15: 'Selecting, extracting and sorting CLIENT records'
    print
20  open structure cl: name 'tti_run:client'
30  extract structure cl
      include cl(state) = 'CA'
      exclude cl(phone)[1:3] = '619'
      sort descending by cl(last)
    end extract
40  for each cl
      print cl(id); tab(12); cl(last); ' '; cl(first), cl(city); ' '; &
        cl(state), cl(phone)
    next cl
50  close structure cl
60  end
```

```
.screen -

              Selecting, extracting and sorting CLIENT records

80522      Errant Earl                 Monterey CA         (408) 844-7676
80561      Derringer Dale              Los Angeles CA      (818) 223-9014
```

Example 2 - Using Input Parameters to Select Records

Often, the *user* will select what data they want to see on a report. This example shows how to use input parameters to select records.

```
1   program application_6_4
10  clear
    print at 1, 1:;
20  message 'Enter lst letter of last names to be selected ' &
        + 'separated by commas'
    line input 'Letters (A,B,C)': list$
    print
30  open structure cl: name 'tti_run:client'
40  extract structure cl
      include match(list$, cl(last)[1:1]) > 0
      sort ascending by cl(last)
    end extract
50  for each cl
      print cl(id); tab(12); cl(last); ' '; cl(first); ' '; &
        cl(middle), cl(city); ' '; cl(state), cl(phone)
    next cl
60  close structure cl
70  end
```

```
.screen -

Letters (A,B,C)? a,b,f,s,w

80531      Abott Al A                  New York NY         (202) 566-9892
80542      Brock Bud B                 Duluth MN           (218) 555-4322
80573      Farmer Fred F               Miami FL            (305) 552-7872
20000      Smith Sam S                 Seattle WA          (809) 555-8789
32001      Waters Wayne W              San Diego CA        (619) 564-1231
```

Example 3 - Sorting Fields not in the Main Structure

There are times when you want to list data in a specific order but the field that needs to be sorted is located in another structure. For example, you want to list the invoices from the INVOICE structure in customer name order. INVOICE contains a customer number but not the name. The name is only in the CUSTOMER structure. This example extracts INVOICE records which have an invoice amount greater than $11,000.00 and sorts the invoices by customer name. The **SET STRUCTURE, FIELD: KEY** statement uses the customer number to get the CUSTOMER file record so the name can be sorted. When this statement is used, the *FIELD* must be a key field.

```
1   program application_6_5
10  clear
    print at 1, 1:;
20  open structure inv: name 'tti_run:invoice'
    open structure cust: name 'tti_run:customer'
30  extract structure inv
      include inv(invamt) > 11000.00
      set structure cust, field custnbr: key inv(custnbr)
      sort by cust(name)
    end extract
40  print 'Cust#'; tab(8); 'Customer Name'; tab(40); 'Inv#'; &
          tab(50); 'Inv Amount'
    print
50  for each inv
      set structure cust, field custnbr: key inv(custnbr)
      if  _extracted = 0  then
        cust_name$ = 'UNKNOWN'
      else
        cust_name$ = cust(name)
      end if
      print inv(custnbr); tab(8); cust_name$; tab(40); inv(invnbr); &
            tab(50); inv(invamt)
    next inv
60  close all
70  end
```

```
.screen -

Cust#  Customer Name                  Inv#      Inv Amount

12513  Alpha Products Inc             10376      12,453.00
21670  Anywhere Travel Service        10301      28,712.00
21670  Anywhere Travel Service        10348      13,168.00
24424  Flower Power, Inc              10330      14,255.00
24424  Flower Power, Inc              10347      12,737.00
24424  Flower Power, Inc              10358      12,959.00
20348  MicroNet Solutions             10341      13,143.00
20348  MicroNet Solutions             10345      20,511.00
20348  MicroNet Solutions             10355      18,163.00
20348  MicroNet Solutions             10363      12,472.00
21138  Peabody Plumbing               10395      11,165.00
19279  Sky High Airlines              10384      12,150.00
13727  Tumbleweeds Real Estate        10311      12,010.00
```

Example 4 - Using Keys to Find Records

It is possible to locate specific records using record keys. You can use the PRIMARY key or ALTERNATE keys. The key can be the complete key or a partial key. These examples show two ways to retrieve records using partial keys.

This example extracts *all* records that start with the input partial customer number.

```
1   program application_6_6
10  clear
    print at 1, 1:;
20  open structure cust: name 'tti_run:customer'
30  do
      message 'Enter the first part of the customer number'
      input 'Customer number': cust_number$
      extract structure cust, field custnbr: partial key cust_number$
      end extract
      if  _extracted = 0  then
        message error: 'No numbers found starting with ' + cust_number$
        repeat do
      else
        exit do
      end if
    loop
40  print
    for each cust
      print cust(custnbr), cust(name)
    next cust
50  close structure cust
60  end
```

```
.screen -

Customer number? 2

20341               Loprice Drug Stores, Inc
20348               MicroNet Solutions
21138               Peabody Plumbing
21670               Anywhere Travel Service
24424               Flower Power, Inc
```

This example uses the **SET STRUCTURE** statement. In this case, only the *first* record that matches the partial customer number is extracted.

```
1   program application_6_7
10  clear
    print at 1, 1:;
20  open structure cust: name 'tti_run:customer'
30  do
      message 'Enter the first part of the customer number'
      input 'Customer number': cust_number$
      set structure cust, field custnbr: partial key cust_number$
      if  _extracted = 0  then
        message error: 'No number found starting with ' + cust_number$
        repeat do
      else
        print
        print cust(custnbr), cust(name)
        exit do
      end if
    loop
40  close structure cust
50  end
```

```
.screen -

Customer number? 2

20341               Loprice Drug Stores, Inc
```

Example 5 - Printing to an Output File

This example shows how to create and print to an output file for report purposes. The **_CHANNEL** option lets INTOUCH assign the output channel number for the output file. The **UNIQUE** option lets INTOUCH create the output file name and the **ASK** statement provides the output file name. When reading the data from the output file, the **EOF** option in the "line input" statement checks for the end of the file.

```
1   program application_6_8
10  clear
    print at 1, 1:;
20  message 'Enter lst letter of last names to be selected ' &
        + 'separated by commas'
    line input 'Letters (A,B,C)': list$
    print
30  open structure cl: name 'tti_run:client'
    out_channel = _channel
    open #out_channel: unique, access output
    ask #out_channel: name out_file$
40  extract structure cl
      include match(list$, cl(last)[1:1]) > 0
      sort ascending by cl(last)
    end extract
50  for each cl
      print #out_channel: cl(id); tab(12); cl(last); ' '; cl(first); &
            ' '; cl(middle); tab(45); cl(city); ' '; cl(state)
    next cl
60  close all
70  open #out_channel: name out_file$
    clear
    print at 2, 10: 'The input selections were:  '; list$;
    print at 3, 10: 'The output file name is:  '; out_file$
    print at 4, 10: 'The output file contains the following records'
    print
80  do
      line input #out_channel, eof end_file: line$
      if  end_file  then  exit do
      print line$
    loop
90  close #out_channel
    end
```

```
.screen -

         The input selections were:  b,d,e,f,s,w
         The output file name is:  USER:[TESTER]TMP_360001.TMP
         The output file contains the following records

80542      Brock Bud B                     Duluth MN
80561      Derringer Dale D                Los Angeles CA
56009      Donaldson George
80522      Errant Earl E                   Monterey CA
80573      Farmer Fred F                   Miami FL
20000      Smith Sam S                     Seattle WA
32001      Waters Wayne W                  San Diego CA
```

Example 6 - Moving Around Within a Structure

You might need to return to a previously read record. To do this, you can use the **ASK STRUCTURE** and the **SET STRUCTURE** statements to move around in the file.

This example uses the **ASK/SET STRUCTURE: POINTER** statements. The program extracts records and creates a collection of pointers. Some data is displayed and the user can get additional information on a selected item.

```
1   program application_6_9
10  clear
20  open structure cl: name 'tti_run:client'
30  extract structure cl
      sort by cl(last)
    end extract
40  x = 0
    print at 2,1:;
    for each cl
      x = x + 1
      print x; tab(6); cl(last)
      if  x = 12  then  exit for
    next cl
    tot$ = str$(x)
50  do
      message 'Enter a sequence no. between 1 and ' + tot$
      input 'Sequence No.', at 17,1, &
            valid 'number; allow 1 to '+ tot$ : seq_no
      if  _back  or  _exit  then
        clear
        exit do
      end if
      set structure cl: pointer seq_no
      ask structure cl: pointer ptr
      print
      print 'This record is pointer number '; ptr
      print cl(id); tab(12); cl(last); ', '; cl(first); &
            tab(40); cl(city); ' '; cl(state); tab(65); cl(phone)
      clear area 17,1,17,80
    loop
60  close structure cl
70  end
```

```
.screen -
  1   Abott
  2   Brock
  3   Cass
  4   Derringer
  5   Donaldson
  6   Errant
  7   Farmer
  8   Johnson
  9   Kent
  10  Porter
  11  Rodrigues
  12  Smith


 Sequence No.? 6

 This record is pointer number  6
 80522      Errant, Earl               Monterey CA             (408) 844-7676

                    Enter a sequence no. between 1 and 12
```

This is just one example using **SET STRUCTURE**. There are a number of options for the **ASK STRUCTURE** and **SET STRUCTURE** statements. All of the options are described in detail in the *INTOUCH - A Guide to the Language* manual.

7 Working with Files

In INTOUCH, files contain *unstructured data* and are opened using a channel number. Files have records, but do NOT have fields. INTOUCH can read and store data in structures and in files.

Files provide you with places to store data. They can exist on various devices (i.e. disks, tapes, etc.) and are identified by a specific file specification (name). Data saved to a file is stored in data records. Once a file is open, you have access to all data records stored in the file.

INTOUCH keeps file manipulation simple. Files can be opened and closed as they are needed. Writing new records, as well as reading, updating and deleting existing records are simple tasks once the file is opened with the right access option. If the file does not already exist, it can easily be created with an INTOUCH statement.

7.1 Opening and Closing Files

Before you can do anything to a file in INTOUCH, the file must be opened. Once the file is open, you have access to its data records. The **OPEN** statement opens an existing file or creates a new file.

When you open a file, you need to assign it a channel. You can select from channels 1 through 95. Channel 0 is reserved for the terminal. If you do not know or care what channel is assigned to a file, you can use the INTOUCH **_CHANNEL** built-in function. This function simply grabs the next available channel number and assigns it to the file.

When you open a file, you must specify how the file is going to be used, the **access mode**. The three access modes are:

- **INPUT** - the file can only be read
- **OUTPUT** - the file can only be written to
- **OUTIN** - the file can be read and written to

If no access mode is specified, the access defaults to **INPUT**.

The **OPEN** statement can be used to create and open a new file. If you are creating a new file, the access must be either **OUTPUT** or **OUTIN**.

When you open a file, you need to provide the file name. If you are creating a temporary output file and do not care what the name is, you can specify *UNIQUE* and let INTOUCH create a unique file name for you.

Once you have finished working with the files, all the files should be closed. You can use the **CLOSE** statement to close each file or use **CLOSE ALL** to close all the files.

Example

```
1   program application_7_1
10  clear
    out_ch1 = _channel
20  open #out_ch1: name 'test_file_1.tmp', access output
    ask #out_ch1: name file_1$
    print at 1, 1: 'File '; file_1$; ' has just been opened...'
30  out_ch2 = _channel
    open #out_ch2: unique, access output
    ask #out_ch2: name file_2$
    print
    print 'File '; file_2$; ' has just been opened...'
    print 'This unique file name was created by INTOUCH'
40  out_ch3 = _channel
    open #out_ch3: name 'test_file_3', unique, access output
    ask #out_ch3: name file_3$
    print
    print 'File '; file_3$; ' has just been opened...'
    print 'INTOUCH made the file name TEST_FILE_3 into a unique name'
50  close all
60  end
```

```
.screen -
File USER:[TESTER]TEST_FILE_1.TMP has just been opened...

File USER:[TESTER]TMP_4E0001.TMP has just been opened...
This unique file name was created by INTOUCH

File USER:[TESTER]TEST_FILE_3_4E0002.TMP has just been opened...
INTOUCH made the file name TEST_FILE_3 into a unique name
```

7.2 Storing and Reading File Data

To store data in a file, you use the **PRINT** statement to write out the data. The file must first be opened with the access mode OUTPUT or OUTIN before data can be written to the file.

Writing data to a file is similar to printing data to the screen, except that a channel is specified. A typical **PRINT** statement defaults to channel 0, which is the terminal. Therefore, to send data to a file, just specify the **channel** assigned to the file in the **PRINT** statement.

Once a file is opened in INTOUCH, you have access to all the records until the file is closed. You can always *read* data from the file. After the file is opened, the data can be sequentially read in by element or by line using the **INPUT** and **LINE INPUT** statements, respectively. The OPEN statement makes the first record current (i.e. available).

When reading records from a file, you usually need to know when you have read the last record. INTOUCH allows you to check for the end-of-file condition by using the **EOF** option of the **LINE INPUT** statement.

The **EOF** option of the INPUT statement causes INTOUCH to return **TRUE** (i.e. 1) for a true condition or **FALSE** (i.e. 0) for a false condition, if the end of the file is or is not reached, respectively. The following example shows how the IF/THEN conditional construct tests for an end-of-file condition.

In the following example, the variable "endfile?" could also be "endfile" without the "?". The "?" simply flags the variable as a condition variable meaning that it contains the results of a TRUE or FALSE test. The "?" makes the variable easy to identify.

Example

```
1   program application_7_2
10  clear
    print at 1, 1:;
    out_ch = _channel
20  open #out_ch: name 'temp', unique, access output
    ask #out_ch: name filename$
    for i = 1 to 5
      text$ = 'Record ' + str$(i)
      print #out_ch: text$
      print 'Writing out:  '; text$
    next i
30  close #out_ch
40  open #out_ch: name filename$
    print
    do
      line input #out_ch, eof endfile?: line$
      if  endfile?  then
        print 'End of file - the last record has already been read'
        exit do
      end if
      print 'Reading:  '; line$
    loop
50  close #out_ch
60  end
```

```
.screen -

  Writing out:  Record 1
  Writing out:  Record 2
  Writing out:  Record 3
  Writing out:  Record 4
  Writing out:  Record 5

  Reading:  Record 1
  Reading:  Record 2
  Reading:  Record 3
  Reading:  Record 4
  Reading:  Record 5
  End of file - the last record has already been read
```

Unformatted output files

There are times when an output file is to become the input file for some device (i.e. PC, plotter, etc.). When an output file is created for export, the **UNFORMATTED** option in the open statement can be used to suppress the control characters (i.e. carriage return, line feed) from the output data.

Example

```
1   program application_7_3
10  clear
    print at 1, 1:;
    out_ch = _channel
20  open #out_ch: name 'file', unique, access output, unformatted
    ask #out_ch: name filename$
    for i = 1 to 5
      text$ = 'Item ' + str$(i)
      print #out_ch: text$
      print 'Writing out:  '; text$
    next i
30  print #out_ch: 'This is the end of the file'
    print
40  close all
50  command$ = 'type ' + filename$
    pass command$
    print
60  end
```

```
.screen -

  Writing out:  Item 1
  Writing out:  Item 2
  Writing out:  Item 3
  Writing out:  Item 4
  Writing out:  Item 5

  Item 1Item 2Item 3Item 4Item 5This is the end of the file
```

7.3 Positioning File Records

There are times when you are reading through a file and want to return to a previously read record. Instead of closing the file, reopening it (going back to the first record), and reading each successive record up to the one you want, you can accomplish the same task with two simple INTOUCH statements. Using the **CURRENT** option, with the **ASK** and **SET** statements enables you to switch between records.

- The **ASK channel: CURRENT** statement saves the current record's location in the specified string variable.

- The **SET channel: CURRENT** statement usually follows the **ASK channel: CURRENT** statement in a program. This statement fetches a specific record from the data file. The string variable contains the location of the record to be fetched.

Example

```
1   program application_7_4
10  clear
    print at 1, 1:;
    out_ch = _channel
    filename$ = 'temp.tmp'
20  open #out_ch: name filename$ , access output
    for i = 1 to 10
      text$ = 'Record ' + str$(i)
      print #out_ch: text$
      print 'Writing out:  '; text$
    next i
    close #out_ch
    print
30  open #out_ch: name filename$
    input #out_ch: line1$
    print 'The 1st record is :  '; line1$
    for j = 1 to 3
      input #out_ch: line2$
    next j
    print 'The 4th record is :  '; line2$
40  ask #out_ch: current cur_line$
    for k = 1 to 4
      input #out_ch: line3$
    next k
    print 'The 8th record is :  '; line3$
50  set #out_ch: current cur_line$
    print 'Back to 4th record:  '; line2$
60  close all
70  end
```

```
.screen -

  Writing out:  Record 1
  Writing out:  Record 2
  Writing out:  Record 3
  Writing out:  Record 4
  Writing out:  Record 5
  Writing out:  Record 6
  Writing out:  Record 7
  Writing out:  Record 8
  Writing out:  Record 9
  Writing out:  Record 10

  The 1st record is :  Record 1
  The 4th record is :  Record 4
  The 8th record is :  Record 8
  Back to 4th record:  Record 4
```

8 Working with Data

When data is printed, displayed or stored in record fields, the data is often not in the required format. For example, when dollar amounts are printed on a report, you want to print the amount as *$9,999,999.99* not as *9999999.99*. In another case, you might have a string of data from which you need to extract a piece of information. INTOUCH provides many built-in functions to help you work with data.

8.1 Formatting Data

Often, when data is printed on reports or stored in record fields, the data needs to be changed to different formats. Formatting data can also involve data alignment and field justification.

There are a number of ways to format and align data using INTOUCH.

PRINT USING

The **PRINT USING** statement is most often used to print numbers but it can also be used to print characters. **PRINT USING** formats data according to the mask you provide. This statement allows you to specify the character type, field length, field alignment, etc. You can also insert special characters (i.e. "**$**" for dollar values, "**.**" for numeric data, "**+/-**" for leading characters, etc.).

The *INTOUCH - A Guide to the Language* manual contains descriptions of all the format characters.

FORMAT$ Function

FORMAT$( ) works like **PRINT USING**. The difference is that **PRINT USING** is a print statement and **FORMAT$** is a function. **FORMAT$** can be used in a print statement.

Example

The following example shows how to use different variations of the **PRINT USING** statement and the **FORMAT$** function.

```
1   program application_8_1
10  clear
    phone_no$ = '5551212'
    dollars$ = '999999999'
20  print at 1, 15: 'Different variations can produce the same results'
    print
    print 'Dollar amount:';
    print using '$###,###,###.##': dollars$
    dollar_mask$  = '$###,###,###.##'
    print 'Dollar amount:';
    print using dollar_mask$: dollars$
    print
30  z$ = format$(phone_no$, '###~-####')
    print 'Phone number :  '; z$
    phone_mask$ = '###~-####'
    print 'Phone number :  '; format$(phone_no$, phone_mask$)
40  end
```

```
.screen -

              Different variations can produce the same results

Dollar amount:  $9,999,999.99
Dollar amount:  $9,999,999.99

Phone number :  555-1212
Phone number :  555-1212
```

LSET, RSET and CSET

The **LSET**, **RSET** and **CSET** statements allow you to format and align *string* data. You can also fill the variable or field with a designated value using the **FILL** option.

- **LSET** left-justifies data within a specified variable or field.
- **RSET** right-justifies data within a specified variable or field.
- **CSET** centers data within a specified variable or field.

LPAD$, RPAD$ and CPAD$

The **LPAD$( )**, **RPAD$( )** and **CPAD$( )** functions allow you to pad data on the left, right or both sides with a designated character. The function creates a new string containing the data and pad characters. If no pad character is specified, the default pad character will be a *space*.

- **LPAD$( )** pads a string on the left with the specified pad character.
- **RPAD$( )** pads a string on the right with the specified pad character.
- **CPAD$( )** centers the data in a string and pads the left and right sides with the specified pad character.

Example

This example shows some formatting techniques.

```
1   program application_8_2
10  clear
    print at 1, 1:;
20  name_mask$ = '>@@@@@@@@@@'
    ssn_mask$ = '###~-##~-####'
    h_date$ = space$(15)
    h_page$ = space$(15)
    h_title$ = space$(40)
    page_no = 1
30  input prompt 'Enter your name  : ', at 3,5: name$
    input prompt 'Soc. Security No.: ', at 4,5: ssn$
    input prompt 'Enter a number   : ', at 5,5: r_number
    delay
40  lset h_date$ = date$[5:6] + '/' + date$[7:8] + '/' + date$[3:4]
    rset h_page$ = 'Page ' + str$(page_no)
    cset h_title$ = 'Examples of FORMATTING Data'
    heading$ = h_date$ + h_title$ + h_page$
50  clear
    print bold, at 1, 1: heading$
    print at 3,1: 'Date was LSET  :  '; '('; h_date$; ')'
    print 'Page was RSET  :  '; '('; h_page$; ')'
    print 'Title was CSET :  '; '('; h_title$; ')'
    print '     The 3 parts made up the heading'
    print
60  print bold: cpad$('Examples of PRINT USING', 70, '-')
    print
    print 'Right justified  :',
    print using name_mask$: name$
    print 'Screen formatted :',
    print using ssn_mask$: ssn$
    print 'Preceeding zeros :',
    print using '%%%%%%%%%%': r_number
    print 'Leading "+"      :',
    print using '+##########': r_number
    print 'Dollar format    :',
    print using '$###,###,###.##': r_number
    print
70  print bold: cpad$('Examples of Padding', 70, '-')
    print
    print 'RPAD$ - pads on the right  :  '; rpad$(name$, 30, '*')
    print 'LPAD$ - pads on the left   :  '; lpad$(name$, 30, '*')
    print 'CPAD$ - pads on both sides :  '; cpad$(name$, 30, '*')
    delay
80  end
```

```
.screen -
    Enter your name  : Molly
    Soc. Security No.: 123456789
    Enter a number   : 730528


                        Press the RETURN key to continue
```

```
.screen -
09/18/94              Examples of FORMATTING Data               Page 1

Date was LSET  :  (09/18/94       )
Page was RSET  :  (         Page 1)
Title was CSET :  (       Examples of FORMATTING Data      )
     The 3 parts made up the heading

------------------------Examples of PRINT USING-----------------------

Right justified  :        Molly
Screen formatted :  123-45-6789
Preceeding zeros :  00000730528
Leading "+"      :      +730528
Dollar format    :      $730,528.00

--------------------------Examples of Padding--------------------------

RPAD$ - pads on the right  :  Molly*************************
LPAD$ - pads on the left   :  *************************Molly
CPAD$ - pads on both sides :  *************Molly*************

                        Press the RETURN key to continue
```

8.2 Editing Data

The **TRIM$( )**, **LTRIM$( )**, **RTRIM$( )** and **EDIT$( )** functions can be used to remove spaces, tabs and other characters. **SPACE$( )** can be used to insert spaces.

Example

This example shows how to use the trim and edit functions.

```
1   program application_8_3
10  clear
    print at 1, 1:;
20  dim text$(8,2)
    text$(1,1) = '     TRIM$ discards leading and trailing spaces and t
    text$(1,2) = trim$(text$(1,1))
    text$(2,1) = '     LTRIM$ discards LEADING spaces and tabs     '
    text$(2,2) = ltrim$(text$(2,1))
    text$(3,1) = '     RTRIM$ discards TRAILING spaces and tabs     '
    text$(3,2) = rtrim$(text$(3,1))
    text$(4,1) = ' E D I T - 2 - discards all spaces and tabs '
    text$(4,2) = edit$(text$(4,1), 2)
    text$(5,1) = 'E   D   I   T   -   16 - reduces spaces, tabs to 1 sp
    text$(5,2) = edit$(text$(5,1), 16)
    text$(6,1) = 'edit - 32 - converts lower case to upper case'
    text$(6,2) = edit$(text$(6,1), 32)
    text$(7,1) = ' e d i t - 3 4 - discards spaces, tabs & upper cases '
    text$(7,2) = edit$(text$(7,1), 34)
    text$(8,1) = 'SPACE$ creates spaces'
    text$(8,2) = space$(5) + text$(8,1) + space$(5)
30  for i = 1 to 8
      print '|'; text$(i,1); '|'
      print '|'; text$(i,2); '|'
      print
      delay
    next i
40  end
```

```
.screen -

|     TRIM$ discards leading and trailing spaces and tabs     |
|TRIM$ discards leading and trailing spaces and tabs|

|     LTRIM$ discards LEADING spaces and tabs     |
|LTRIM$ discards LEADING spaces and tabs     |

|     RTRIM$ discards TRAILING spaces and tabs     |
|     RTRIM$ discards TRAILING spaces and tabs|

| E D I T - 2 - discards all spaces and tabs |
|EDIT-2-discardsallspacesandtabs|

|E   D   I   T   -   16 - reduces spaces, tabs to 1 space|
|E D I T - 16 - reduces spaces, tabs to 1 space|

|edit - 32 - converts lower case to upper case|
|EDIT - 32 - CONVERTS LOWER CASE TO UPPER CASE|

| e d i t - 3 4 - discards spaces, tabs & upper cases |
|EDIT-34-DISCARDSSPACES,TABS,UPPERCASES|

|SPACE$ creates spaces|
|     SPACE$ creates spaces     |
```

The following is a list of the **EDIT$** operations that can be performed. You can also combine edit values to perform more than one operation. For example, edit-34 in the above example, consists of edit value 2 (discard all spaces and tabs) plus edit value 32 (convert lower case to upper case).

Table 8–1 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.

8.3 Extracting Data

This section explains how to extract pieces of data from a text string or data field.

ELEMENT$( ) allows you to extract a specific item from a list of items; the items can be separated by a comma (the default), space, etc.

MID or **MID$( )** allows you to extract characters from within a text string or data field; you must provide the start position and, optionally, the end position

SEG$( ) works the same as MID except that you must provide both the start and end positions of the data you want to extract

POS( ) gives you the starting position of specified data in a text string or data field

LEN( ) tells you how long a text string or field is

PATTERN( ) gives you the starting position of a specific pattern of characters that you are looking for or tells you if the data is present in the text string or field; in the following example, we are looking for "arrive ??"; the "??" means that any 2 data characters following "arrive " are acceptable

REPEAT$( ) creates a character the number of times you specify

Note

All of the above functions and many more are described in the appendix section of the "INTOUCH - A Guide to the Language" manual.

Example

The following example shows you how to use functions to extract data.

```
1   program application_8_4
10  clear
    print at 1, 1:;
    dim line$(3), carrier$(3), flight_no$(3), time$(3)
20  line$(1) = 'United Flight #452 is scheduled to arrive at 8:15am.'
    line$(2) = 'American Flight #220 is scheduled to arrive at 9:44am.'
    line$(3) = 'TWA Flight #33 is delayed and will arrive at 7:30pm.'
30  print 'Specific data is extracted from these text lines:'
    print
    for i = 1 to 3
      print space$(5); line$(i)
    next i
    print
    print
40  for i = 1 to 3
      carrier$(i) = element$(line$(i), 1, ' ')
      flight_no$(i) = mid$(line$(i), pos(line$(i), '#')+1, len(line$(i)))
      flight_no$(i) = element$(flight_no$(i), 1, ' ')
      time$(i) = seg$(line$(i), pattern(line$(i), 'arrive ??')+10, len(line$(i))-1)
    next i
50  print 'Carrier', 'Flight#', 'Arrival'
    z$ = repeat$('-', 7)
    print z$, z$, z$
    for i = 1 to 3
      print carrier$(i), flight_no$(i), time$(i)
    next i
60  end
```

```
.screen -

Specific data is extracted from these text lines:

     United Flight #452 is scheduled to arrive at 8:15am.
     American Flight #220 is scheduled to arrive at 9:44am.
     TWA Flight #33 is delayed and will arrive at 7:30pm.

Carrier             Flight#             Arrival
-------             -------             -------
United              452                 8:15am
American            220                 9:44am
TWA                 33                  7:30pm
```

RIGHT$( ) Function

If you need to extract characters from the end of a text string or data field, you can use the **RIGHT$( )** function. This function will give you the rightmost characters. You specify the number of characters you want.

ELEMENTS and PIECES

When you need to work with parts of a text string, you can use either the **ELEMENT$( )** or **PIECE$( )** function to break it down into pieces. If you want to know the number of pieces contained in the text string, you can use either the **ELEMENT( )** or **PIECE( )** function to tell you.

ELEMENT( ) and **ELEMENT$( )** automatically trim leading and trailing spaces, skip over quoted data and set the default separator to a comma. The default separator for **PIECE( )** and **PIECES$( )** is the carriage return/line feed sequence.

- When you need to know the number of elements or pieces in a string expression, use either the **ELEMENTS( )** or **PIECES( )** function. Both functions return the item count.
- When you want to isolate one element or piece from a string expression, use the **ELEMENT$( )** or **PIECE$( )** function.

Example

```
 1   program application_8_5
 10  clear
     text1$ = 'Some Animals in the Zoo'
     text2$ = 'Bear, Elephant, Zebra, Tiger, Lion, Monkey, Giraffe, Gorilla'
 20  print at 3,20: text1$
     print at 5,5: text2$
     print
     print
 30  z = elements(text2$, ' ')
     print 'There are'; z; 'elements in the animal list'
     print 'The 5th element is:  '; element$(text2$, 5, ', ')
     print
 40  z = pieces(text2$, ' ')
     print 'There are'; z; 'pieces in the animal list'
     print 'The 5th piece is:  '; piece$(text2$, 5, ', ')
 50  end
```

```
.screen -

                  Some Animals in the Zoo

    Bear, Elephant, Zebra, Tiger, Lion, Monkey, Giraffe, Gorilla

There are 8 elements in the animal list
The 5th element is:  Lion

There are 8 pieces in the animal list
The 5th piece is:  Lion
```

8.4 Changing/Replacing Characters/Strings

You can change and replace data using the **CHANGE$( )** and **REPLACE$( )** functions.

- **CHANGE$( )** changes characters one at a time, one for one. Since it works character for character, the old and new string lengths are the same.
- **REPLACE$( )** replaces character sequences. It replaces targeted character patterns with new characters. The old and new strings might not be the same length depending on the replacement.

Example

```
1   program application_8_6
10  clear
    print at 1, 1:;
20  print at 3,5: 'This example CHANGEs "at" to "x":'
    text$ = 'My cat is fat'
    old$ = 'at'
    new$ = 'x'
    print at 5,10: 'Before:  '; text$
    print at 6,10: 'After :  '; change$(text$, old$, new$)
30  print at 9,5: 'This example REPLACEs "og" and "as" with "yyy":'
    text$ = 'My dog has fleas'
    old_new$ = '{<og|as>}=yyy'
    print at 11,10: 'Before:  '; text$
    print at 12,10: 'After :  '; replace$(text$, old_new$)
40  end
```

```
.screen -

    This example CHANGEs "at" to "x":

         Before:  My cat is fat
         After :  My cxx is fxx

    This example REPLACEs "og" and "as" with "yyy":

         Before:  My dog has fleas
         After :  My dyyy hyyy fleyyy
```

8.5 Converting Data

It is necessary to convert numeric values to string values and string values to numeric values to perform certain types of processing. For example, some formatting can only be done on string values and calculations can only be done on numeric values.

INTOUCH has several functions you can use to convert data.

- Converting between string and numeric values:

  STR$( ) converts a numeric value to its string equivalent.

```
a = 25
z$ = str$(a)
```

  VAL( ) converts only numeric strings back to their floating point value.

```
a$ = "25"
z = val(a$)
```

- Converting between upper and lower case:

  UCASE$( ) converts each character in the specified string expression to its uppercase equivalent. The EDIT$(...,32) function can also be used to convert lowercase to uppercase.

```
ucase$(text$)
```

  LCASE$( ) converts uppercase to lowercase.

```
lcase$(text$)
```

- Quotation marks:

  QUOTE$( ) wraps the specified string expression with double quotes.

```
quote$(text$)
```

  UNQUOTE$( ) removes any quotes, single or double (paired or unpaired), surrounding the specified string expression.

```
unquote$(text$)
```

9 Logic Flow and Routines

Writing code which has clear logic flow is essential to effective programming. It helps you and others understand the program's goals, thus minimizing time and effort spent on future changes and enhancements.

INTOUCH offers an array of conditionals, loop constructs and branching statements. Each type of statement targets different areas of logic flow.

9.1 Conditional Statements

Programmers often need to check for specific conditions and take some type of action accordingly. INTOUCH provides two types of conditional constructs (statements). The types are:

- the **IF/THEN** construct
- the **SELECT CASE** construct

Which of these two constructs to use depends on the condition that is being checked.

The **IF/THEN** construct is the simplest conditional construct and is most often used for basic checks (i.e. if _EXIT then exit do).

The **SELECT CASE** construct is most often used when the condition can be a number of elements of the same type (i.e. ADD,CHA,DEL,INQ). The **SELECT CASE** construct is much like a specialized **IF/THEN**. It solely relies on one expression, mapping a different path for each possible match.

Therefore, it is advantageous to use the **SELECT CASE** statement instead of the **IF/THEN** when you are dealing with one expression which has more than just a few possible matches.

Example

The following example shows some conditions when the **IF/THEN** and **SELECT CASE** statements might be used.

```
1   program application_9_1
10  clear
20  procedures$ = '%at 2, 10, %title "Procedures",' &
        + '"Add New Customers" = ADD, "Change Customer Data" = CHA,' &
        + '"Delete Customers" = DEL, "Customer Inquiry" = INQ, "Exit"'
30  line input menu procedures$: selection$
40  do
      if  _back  or  _exit  then  exit do
      print at 1, 1:;
      select case selection$
      case 'ADD'
        print 'Adding new customers...'
      case 'CHA'
        print 'Changing customer data...'
      case 'DEL'
        print 'Deleting customers...'
      case 'INQ'
        print 'Customer inquiries...'
      end select
      delay
      clear
      line input menu procedures$, default _string: selection$
    loop
50  message 'The End'
60  end
```

```
.screen -
         +-------Procedures--------+
         | ~Add~New~Customers~      |
         |  Change Customer Data    |
         |  Delete Customers        |
         |  Customer Inquiry        |
         |  Exit                    |
         +--------------------------+
```

The following is displayed if "Change Customer Data" is selected.

```
.screen -
Changing customer data...

                    Press the RETURN key to continue
```

The menu defaults to the previous selection.

```
.screen -
         +-------Procedures--------+
         |  Add New Customers      |
         | ~Change~Customer~Data~  |
         |  Delete Customers       |
         |  Customer Inquiry       |
         |  Exit                   |
         +-------------------------+
```

9.2 Looped Logic

When you create a program and routines in a program, you want the logic of the program and the routines to flow downward. This flow can be achieved by using loops. There are several different types of loops.

- The **FOR/NEXT** loop performs a specified number of times.
- The **DO/LOOP** performs an infinite number of times.
- The **DO/LOOP** with the **DO WHILE** or **UNTIL** clause performs while or until some condition is met.
- The **DO/END DO** is a single iteration loop.

A loop performs a set of instructions over and over. However, there are times when, under certain conditions, you want to branch around in the loop or you want to exit out of the loop. Depending on which type of loop you are in, there are different ways of branching and exiting.

Repeating a Loop

If you are in an INTOUCH loop and want to jump back to the beginning of the loop without executing remaining lines in the loop (and possibly incrementing an index variable), you can use the **REPEAT** statement. The **REPEAT FOR** statement is used for **FOR/NEXT** loops, and the **REPEAT DO** statement is used for the **DO/LOOP** and for the **DO/END DO** loops.

Reiterating the Loop

If you are in a loop and want to jump to the next iteration without executing the rest of the code within the loop, you can use the **ITERATE** statement. The **ITERATE FOR** statement is used for **FOR/NEXT** loops, and the **ITERATE DO** statement is used for **DO/LOOP** and **DO/END DO** loops.

Exiting a Loop

If you are in a loop and want to exit, you can use the **EXIT** statement. The **EXIT FOR** statement is used for **FOR/NEXT**, loops and the **EXIT DO** statement is used for **DO/LOOP** and **DO/END DO** loops.

Example

```
1   program application_9_2
10  do
      clear
      print at 1, 1:;
      count = 0
      answer$ = ''
      password$ = 'PASSWORD'
      do while answer$ <> password$
        input screen 'Password: <ucase, noecho:@@@@@@@@>', at 3,1: answer$
        if  _back  or  _exit  then  exit do
        count = count + 1
      loop until count = 3
      print
      if  answer$ = password$  then
        print 'Login complete'
      else
        print 'Login failed'
        message delay: 'Sorry about that.  Try this word "password"'
        repeat do
      end if
20    end do
30  end
```

```
.screen -
  Password: ________

  Login complete
```

9.3 Routines

Programs normally consist of a series of routines. A routine is a self-contained block of code which performs a task. Routines are convenient to reference, and they can make the main body of the program short and simple.

You can easily create routines in INTOUCH. The following is a list of recommended guidelines to use when writing program routines.

1 Give each routine a standard routine header which clearly identifies the following:

- Routine name
- Routine purpose
- Any local and global variables
- What this routine expects prior to its run
- All possible results of this routine

2 Limit each routine to one simple task (i.e. user input, display, print data, a processing step, etc.).

3 Limit each routine to a certain line size. A good standard is the number of lines contained in one screen (21, with the INTOUCH frame ON).

4 Physically locate all routines after the main body of the program code.

5 Start line numbering at a high enough value (i.e. 12000).

Note

The INTOUCH Editor has many helpful aids to assist you in writing programs. For example, a simple command will create a routine header outline and you need only fill in the appropriate information.

The chapter, "Working in the INTOUCH Environment", explains how to use some of the INTOUCH Editor's program writing tools.

Example 9–1: Example of a Routine

```
.screen -
16500   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! G E T   P R E V   S C R E E N   N U M B E R
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Determine the number of the previous screen.  If this is the
        !   first screen, display an error message.
        !
        ! Expected:
        !       scrn_nbr% = current screen number
        !
        ! Locals:
        !       error = TRUE if screen number is first screen
        !
        ! Results:
        !       scrn_nbr% = previous screen number to display
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine get_prev_screen_number
          error = false
          z1% = scrn_nbr% - 1
          if  z1% < 1  then
            error = true
            message error : "This is the first screen"
          else
            scrn_nbr% = z1%
          end if
        end routine
```

9.3.1 Calling Routines

Routines can be called from anywhere in your INTOUCH program, including from within other routines. There are two basic ways to call an INTOUCH routine by name, and a third way to call a routine whose name is determined only at runtime. Here are the three ways to call routines:

1 **GOSUB routine_name**

INTOUCH routines are primarily called using the **GOSUB** statement.

2 **routine_name (without the GOSUB)**

If the routine name contains at least one underscore "_" character, the routine can be called without specifying *GOSUB*.

3 **DISPATCH str_expr**

INTOUCH offers an alternate method to call routines whose names are not known until runtime. This is the **DISPATCH** statement. **DISPATCH** examines the contents of a string variable which is loaded when the program is run and gosubs to that routine.

9.3.2 REPEAT, EXIT Routines

Branching within INTOUCH routines is accomplished by using the **REPEAT ROUTINE** and **EXIT ROUTINE** statements. Use the **REPEAT ROUTINE** statement to repeat execution of the entire routine, and use the **EXIT ROUTINE** statement to exit the routine without executing the remaining lines.

Note

The appendix section contains a sample program which shows correct logic flow and how routines are used.

10 Working in the INTOUCH Environment

This chapter addresses INTOUCH and its environment. Topics include the basic INTOUCH file types, working in INTOUCH and special commands.

10.1 INTOUCH Editor

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

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

```
$ DEFINE INTOUCH_EDITOR FALSE
$ DEFINE ED*IT == "EDIT/editor_name"
```

10.2 Switching between INTOUCH and VMS

If you are in the middle of editing or debugging a program or you want to check files while the program is running, INTOUCH allows you to suspend your current INTOUCH session and start a VMS process *WHILE* you remain in INTOUCH.

To get to the VMS prompt, enter **$$** at the INTOUCH prompt. This will take you to the "**$$**" prompt. To resume your INTOUCH session, press the Return key or type **EXIT**.

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

This feature enables you to quickly switch between VMS and INTOUCH, depending on what you need to do. The created subprocess goes away upon exit from INTOUCH. The subprocess will also go away if it is not used within a certain amount of time.

10.2.1 Running VMS Commands from INTOUCH

If you are at the INTOUCH prompt and want to run a VMS command (i.e. DIR), you can enter **$** followed by the VMS command. The command will be executed immediately and then the INTOUCH prompt will be displayed.

```
.screen -

INTOUCH

$DIR *.INT

Directory USER:[TESTER]

BOX_TEST.INT;1      DATE_TIME.INT;2     FS1.INT;1           FS2.INT;2
FS3.INT;1           INPUT_TEST.INT;1    MESS_TEST.INT;1

Total of 7 files.
INTOUCH
```

10.2.2 Passing VMS Commands

If you want to execute a VMS command from within your INTOUCH program without exiting INTOUCH, you can use the **PASS** statement. INTOUCH will pass control to VMS and return. You can wait for the VMS command to be completed or use the **NOWAIT** option to return immediately. If you do not want to return to INTOUCH, you can use the **NORETURN** option in the **PASS** statement.

Example

```
1   program application_10_1
10  clear
    print at 1, 1:;
20  message 'Enter lst letter of last names to be selected ' &
        + 'separated by commas'
    line input 'Letters (A,B,C)': list$
    print
30  open structure cl: name 'tti_run:client'
    out_channel = _channel
    open #out_channel: unique, access output
    ask #out_channel: name out_file$
40  extract structure cl
      include match(list$, cl(last)[1:1]) > 0
      sort ascending by cl(last)
    end extract
50  for each cl
      print #out_channel: cl(id); tab(12); cl(last); ' '; cl(first); &
            ' '; cl(middle); tab(45); cl(city); ' '; cl(state)
    next cl
60  close all
70  print 'The output file name is:  '; out_file$
    print 'Passing to VMS to type the output file data...'
    print
    pass_command$ = 'type ' + out_file$
    pass pass_command$
80  print
```

```
            print at 20, 1: 'You are now back in INTOUCH...'
            delay
        90  end
```

```
.screen -
Letters (A,B,C)? w,s,a,b,f

The output file name is:  USER:[JEANNIE]TMP_540006.TMP
Passing to VMS to type the output file data...

80531      Abott Al A                      New York NY
80542      Brock Bud B                     Duluth MN
80573      Farmer Fred F                   Miami FL
20000      Smith Sam S                     Washington WA
32001      Waters Wayne W                  San Diego CA


You are now back in INTOUCH...

                        Press the RETURN key to continue
```

10.3 INTOUCH File Names

The following is a list of the types of files used in the INTOUCH environment.

- **program_name.INT**

  Programs written in INTOUCH are given the default **.INT** extension. When you are bringing an INTOUCH program into the workspace or saving the current program to a file, INTOUCH attaches **.INT** to the file name unless you specify a different extension.

- **file_name.INC**

  Included files contain source code common to programs. They are included into a main program by means of the **%INCLUDE** directive. Included files might or might not contain line numbers. The default extension is **.INC**.

- **program_name.RUN**

  A file with the default **.RUN** extension is created when an INTOUCH program is compiled at the INTOUCH prompt. This file can then be executed either inside or outside the INTOUCH environment.

- **program_name.INT_IMAGE**

  Another type of compiled file is the ***.INT_IMAGE** file. This is a fast-starting memory image file created explicitly from a compile performed outside of INTOUCH. Compiled images have a minimum size of 500 disk blocks, increasing in increments of 256 blocks.

  For AXP (Alpha) systems, the compiled image file will have the extension of **INT_IMAGE_AXP**.

- **file_name.TMP**

  If you transfer to and from an editor while in INTOUCH, temporary files are created which have the extension of **.TMP**.

10.4 INTOUCH Procedures

Once you have entered the INTOUCH environment (typed **INTOUCH**), you can perform different procedures. Some of the procedures are listed here. For a list of all the procedures and more detailed information, you can refer to the *INTOUCH - A Guide to the Language* manual.

NEW ['program_name']

If you want to start writing a new program, enter **NEW** followed by a program name enclosed in quotes. If you do not provide a name, the default program name will be **NONAME**.

```
new 'client_list'
```

If you want to start a new program and already have a program in the workspace, be sure to **SAVE** it. When **NEW** is entered, the workspace is cleared and renamed to the program name you provide or **NONAME**.

OLD ['program_name']

If you want to bring an existing program into the INTOUCH environment, enter **OLD** followed by the program name in quotes.

```
old 'convert_date_time'
```

INTOUCH appends **.INT** to the file name (unless an extension is provided), searches for the file and loads the file into the workspace.

EDIT

You can write or edit a program at the INTOUCH prompt. Use the **EDIT** statement to get into edit mode.

```
    edit
or
    edit ask_date
```

EDIT transfers control to the INTOUCH editor. Exiting the editor returns you to the INTOUCH prompt.

If you want to use an editor in INTOUCH other than the INTOUCH editor, you will need to define the **INTOUCH_EDITOR** logical (see Section 10.1).

Programs can be edited in sections. You can specify line numbers, a label or a routine name. If no section is specified, INTOUCH loads the entire program into the editor's workspace.

SAVE ['program_name']

When you want to save your program, use the **SAVE** statement followed by the program name you want to give your program. The program name is enclosed in quotes. If you do not provide a program name, INTOUCH will use the workspace name as the program name (i.e. the name you provided or NONAME).

```
    save
or
    save 'client_list'
```

INTOUCH writes out the program as a text file with the extension **.INT** unless you provide a different extension.

REPLACE ['program_name']

If you change your program and want to save the changes, use the **REPLACE** statement. Use **SAVE** to save the program the *first* time and use **REPLACE** thereafter.

```
    replace
or
    replace 'client_list'
```

RUN ['program_name']

You can run your program by entering **RUN**. If **RUN** is followed by a program name, INTOUCH will **OLD** in the program and then **RUN** it.

```
    run
or
    run 'convert_date_time'
```

INTOUCH programs can also be run at the VMS prompt ($). To run an INTOUCH program outside of the INTOUCH environment, you need either a compiled file (*.RUN) or a compiled image file (*.INT_IMAGE).

10.5 END, STOP, HALT, GO Statements

The **END** statement is used to end program execution. The **END** statement does not have to be the last physical line in the program. Routines, subprograms, functions, etc. can follow the **END** statement. However, **END** marks the end of the source code for the main program unit.

The **STOP** statement brings program execution to an end the same as the **END** statement. However, **STOP** does not mark the end of the program. The **STOP** statement is sometimes used to stop program execution when a fatal data error is found.

The **HALT** statement interrupts program execution but does not close files or write active output buffers. The INTOUCH prompt is displayed when the **HALT** statement is executed. Use **HALT** if you want to interrupt the program to check values or modify code and then continue. Use the **GO** statement to continue with program execution.

10.6 Recalling Previous Commands

Programming can be a tedious task, when you are dealing with lengthy statements which you type repeatedly. INTOUCH reduces the typing time by offering its own command recall history feature.

The INTOUCH environment supports command recall a little differently than VMS. The last 50 previously entered INTOUCH statements can be recalled using a variety of methods.

There are three basic methods to scan the recall list. Each method uses one key (or key combination) to move one statement up and one key (or key combination) to move one statement down in the recall list.

- UP and DOWN keys
- Prev and Next keys
- Ctrl/B and Ctrl/V key combinations

If you remember part of the command you want to recall, you can have INTOUCH search the command recall list for the statement.

To search the command recall list for a particular statement, type the partial statement and press the Find key or the PF2 key. INTOUCH will locate the first statement which matches the pattern you typed in. PF2 is also associated with the Help key.

10.7 The Tab Key

Typing code can be a time consuming task. INTOUCH tries to do much of the work for you. The multi-function Tab key is used for this purpose. Not only can you speed up the time it takes to type the INTOUCH program, but you also have an entire INTOUCH command reference library at your fingertips.

The Tab key functions can be used when at the INTOUCH prompt or while in the INTOUCH editor.

10.7.1 Command Completion

One shortcut alternative to typing the full INTOUCH command is to type part of the command followed by the Tab key. If only one command matches the pattern, that command will be completed.

```
entering:   30  dis Tab
produces:   30  dispatch
```

If there are several INTOUCH commands which match the pattern, the command choices are displayed in a menu. You can then select one of the items. If you want the first menu item, press the Tab key and that item will automatically be selected.

10.7.2 Command Abbreviations

INTOUCH offers abbreviations for a number of statements. These abbreviations are in the form of one or two alpha characters followed by a period (.). To use an abbreviation, type in the characters, the period and press the Tab key. The abbreviation will be expanded.

The abbreviated statements are:

Table 10–1 Abbreviated INTOUCH Statements

os.	OPEN STRUCTURE
es.	EXTRACT STRUCTURE
ee.	END EXTRACT
in.	INCLUDE
ex.	EXCLUDE
fe.	FOR EACH
e.	EDIT
l.	LIST
p.	PRINT
pu.	PRINT USING
ss.	SET STRUCTURE
pa.	PRINT AT

Table 10–1 (Cont.) Abbreviated INTOUCH Statements

li.	LINE INPUT
pc.	PRINT #
oc.	OPEN #
pe.	PRINT _EXTRACTED
lm.	LINE INPUT MENU
ls.	LINE INPUT SCREEN

10.7.3 Correcting Misspelled Words

The Tab key can be used to correct misspelled words. Position the cursor on the misspelled word and INTOUCH will attempt to correct the spelling. In some cases, pressing the Tab key a second time will display a list of choices.

10.8 Using the INTOUCH Editor

The INTOUCH editor offers you numerous aids to cut down the time it takes to write programs. The following is only one example.

Creating Routine Headers

Creating routine headers is very time consuming and most programmers detest this chore. If you are in the INTOUCH editor, you can let the editor create the routine headers for you and then just fill in the appropriate information. Here is how this is done.

1. At the INTOUCH prompt, enter **EDIT** to get into the INTOUCH editor
2. Press the **DO** key
3. At the command prompt, enter **ROUTINE**
4. At the routine prompt, enter **INITIALIZE**

The routine header and framework will be created for the INITIALIZE routine.

```
.screen -
12000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! I N I T I A L I Z E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !
        ! Expected:
        !
        ! Locals:
        !
        ! Results:
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine initialize

        end routine
```

You can then add the program information.

```
.screen -
12000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! I N I T I A L I Z E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Initialize some constant variables.
        !
        ! Expected:
        !
        ! Locals:
        !
        ! Results:
        !   group_count         count of customers in a group
        !   final_count         count of all customers in all groups
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine initialize
          group_count = 0
          final_count = 0
        end routine
```

Some comments about the **ROUTINE** command:

- If you enter a routine name (at the routine prompt) that already exists in your program, the editor will take you to that routine. Otherwise, if this is a new routine name, the routine framework will be created.
- The editor starts the line numbers with 12000 and increments by 1000. However, if you want to insert a routine between two existing line numbers, the editor will assign a line number that fits into the range. For example, if line numbers 12000 and 12500 exist, the editor will assign 12100 to the new routine.

NOTE: If you are *inserting* a new routine between two existing routines, you *MUST* place the cursor where you want the routine to be located before you use the ROUTINE command.

```
12100   !%%%
```

11 Debugging Techniques

Debugging a program can often take a lot of time and resources. A good debug utility packaged within a language is therefore, very helpful. The INTOUCH debug utility has several features which help in writing code and debugging programs.

Using the Line Interpreter

INTOUCH automatically interprets each line as it is entered at the INTOUCH prompt. If any syntax errors are detected, the error location and cause are displayed.

```
.screen -

INTOUCH

10  open structure cl name 'tti_run:client', access input

10  open structure cl name 'tti_run:client', access input
.....................1
Expected ':'
```

Compile Errors

Running programs containing syntactically incorrect statements produces compile errors. A list of INTOUCH error messages can be found in the appendix section of the *INTOUCH - A Guide to the Language* manual.

Checking Variable Values

Some language interpreters do not allow you to see variables local to routines. Not only does INTOUCH allow you to trace local variables, it also keeps all variables defined even *after* the program has completed running. This allows you to verify the values of program variables after runtime.

11.1 Enabling the Debug System

The **DEBUG ON** statement enables INTOUCH's DEBUG system. When DEBUG is enabled, all of the DEBUG features are available. DEBUG remains enabled until a **DEBUG OFF** statement is executed or until a DEBUG feature is executed which disables it.

DEBUG ON can be issued in immediate mode (at the INTOUCH prompt) or as a statement in a program. If **DEBUG ON** is used in a program, INTOUCH enables DEBUG when the **DEBUG ON** statement is encountered.

11.2 Setting a Break point to Check Variables

If you want to check variables at a specific point in the program, you can use the **BREAK** statement to interrupt program execution. When **BREAK** is encountered, the INTOUCH prompt is displayed and you can print the variables. When you are ready to continue program execution, enter **GO** and the program will resume execution.

After a **BREAK** and when at the INTOUCH prompt, you can use the **STEP** statement to step through lines of your program. If **STEP** is followed by a number (i.e. STEP 10), INTOUCH will execute that number of lines and issue a **BREAK**.

BREAK statements are *ONLY* activated when **DEBUG** is **ON**. If **DEBUG** is **OFF**, **BREAK** statements are ignored.

BREAK statements are sometimes put into routines where problems might occur. Then, if a problem does occur, **DEBUG** can be set **ON**, the program rerun and the programmer can step through the code, check variables, etc. at the break points.

Example

```
1   program application_11_1
10  clear
20  open structure cl: name 'tti_run:client'
30  extract structure cl
      include cl(state) = 'CA'
      exclude cl(phone)[1:3] = '619'
      sort descending by cl(last)
    end extract
    break
40  for each cl
      print cl(id); tab(12); cl(last); ' '; cl(first), cl(city); ' '; &
        cl(state), cl(phone)
    next cl
50  close structure cl
60  end
```

```
.screen -

INTOUCH

run                              <--- run the program

80522      Errant Earl                Monterey CA         (408) 844-7676
80561      Derringer Dale             Los Angeles CA      (818) 223-9014

INTOUCH

debug on                         <--- activate the DEBUG system
```

The INTOUCH screen frame is displayed. You can now RUN the program again.

```
.screen -
INTOUCH~4.1j~~~~~~~~~~~~~~~~~~Program:~NONAME~~~~~~~~~~~~~~~~~~~~~~~~~~~~DEBUG~
The~Next~Generation~Language~~Status~:~~BREAK~at~40.1~~~~~~~~~~~~~~~~~~~~~~~~~
INTOUCH

run                              <--- run the program again

BREAK at 30.5                    <--- 1st break caused by program code

INTOUCH

print _extracted                 <--- display a variable
 2

INTOUCH

step                             <--- give STEP command
                                       shows break, line number at top of screen
INTOUCH                                shows code at bottom of screen

go                               <--- Enter GO to resume program execution

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              print cl(id); tab(12); cl(last); ' '; cl(first), cl(city); ' '; &
EXIT~=~Exit~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\~=~Back~~HELP~=~Help
```

11.2.1 Checking Variables

INTOUCH allows you to check your variables either during runtime (i.e. at BREAK) or after your program has completed running.

If the program variables are such that they do not change during the course of the program, you do not need to turn the DEBUG system on to see the variables. Once the program has completed its run, the variables still contain their last active values.

You can check user input two ways. First, you can print the variable the input is stored in. However, more conveniently, you can print out the **_REPLY** INTOUCH function, which always contains the last user entered input in string format.

11.3 STATS and TRACE Features

Although we make it a point to write efficient code, we are not always successful. Locating performance bottlenecks (i.e. what is causing a program to run so long) can be complicated. INTOUCH, however, provides the **STATS** debug feature, which enables you to pinpoint problem areas in the source code.

Once enabled, the **STATS** feature tracks the number of times each line of code is executed and how long it takes to execute. This feature goes above and beyond the **TRACE** debug feature, which lists the line number as it is executed. To list the statistic information gathered during runtime, you use the **LIST STATS** statement.

- **STATS ON** and **STATS OFF**

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

  STATS ON can be executed in immediate mode (at the INTOUCH prompt) or in a program. If **STATS ON** is executed in immediate mode, DEBUG is automatically switched on. If **STATS ON** is executed in a program, and DEBUG is off, INTOUCH ignores the statement.

- **TRACE ON** and **TRACE OFF**

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

  To use TRACE, you must have the frame ON.

  TRACE ON enables the trace feature. **TRACE ON** can be executed as a command or as a statement. If **TRACE ON** is executed as a command, DEBUG is automatically switched on. If **TRACE ON** is executed as a statement, and DEBUG is off, INTOUCH ignores the trace statement.

Example

The following example shows how to set **STATS ON** and what information is displayed.

```
.screen -
INTOUCH

stats on                <--- turning STATS ON, turns on DEBUG

INTOUCH

run                     <--- run the program

INTOUCH

list stats              <--- list the statistics
```

```
.screen -
NONAME         31-MAY-1994 16:22
          1      0.00           10  clear
          1      0.41           20  open structure cl: name 'tti_run:client'
         15      0.06           30  extract structure cl
         14      0.00                 include cl(state) = 'CA'
          8      0.00                 exclude cl(phone)[1:3] = '619'
          2      0.00                 sort descending by cl(last)
         14      0.00               end extract
          3      0.00           40  for each cl
          2      0.01                 print cl(id); tab(12); cl(last); ' '; cl(f
irst), cl(city); ' '; &
                                        cl(state), cl(phone)
          2      0.00               next cl
          1      0.01           50  close structure cl
          1      0.01           60  end

INTOUCH
```

11.4 Modifying Code at Runtime

If you want to interrupt your program at various points to add/check code or variables and do NOT want to get into the DEBUG system, you can use **HALT** statements. The **HALT** statement works like the **BREAK** statement except that:

HALT will *always* interrupt the program.

BREAK will *only* interrupt the program if DEBUG is ON.

Both statements keep all channels open and allow you to modify any of the code in your program.

A Sample Program

Here is a sample program which shows correct logic flow and how routines and loops are used. The routine headers tell what the routines do.

```
1       program application_b_1
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! Program: CONVERT_DATE_TIME.INT
        ! Package: INTOUCH Examples
        ! Author : Fred Jones
        ! Date   : June 1, 1994
        ! Purpose: Ask the user for a date and time.  Convert the date
        !          and time to various formats.
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

1000    !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! M A I N   L O G I C
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Set up the first action and dispatch to that action.
        !
        ! Expected:
        !
        ! Locals:
        !
        ! Results:
        !       action$ = contains the name of the first routine
        !                 to be executed
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        action$ = 'ask_date'

        do until action$ = 'exit'
          dispatch action$
        loop
        stop
```

Sample Program

```
12000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! A S K   D A T E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Clear the screen.  Ask the user for a date in MMDDYYYY format.
        !   Change the input date to YYYYMMDD format.  Convert the date to
        !   various formats.
        !
        ! Expected:
        !
        ! Locals:
        !       in_date$ = user input date in MMDDYYYY format
        !
        ! Results:
        !       user_date$ = user input date in YYYYMMDD format
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine ask_date
          clear
          message 'Enter a date in MMDDYYYY format'
          line input 'Date', at 3,1, valid 'date MDY', length 8: in_date$
          if  _back  or  _exit  then
            action$ = 'exit'
            exit routine
          end if
          user_date$ = in_date$[5:8] + in_date$[1:4]
          convert_date
          action$ = 'ask_time'
        end routine

13000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! A S K   T I M E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Ask the user for a time in HHMM 24-hour format.  Validate the
        !   input time.  Convert the 24-hour format to 12-hour format.
        !
        ! Expected:
        !
        ! Locals:
        !       in_time$ = user entered time in 24 hour format - HHMM
        !
        ! Results:
        !       hour$ = hour
        !       minute$ = minute
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine ask_time
          message 'Enter the time in 24-hour HHMM format - 0030, 1215, 2245'
          input 'Time', at 4,1, valid 'digits', length 4: in_time$
          if  _back  then
            action$ = 'ask_date'
            exit routine
          end if
          if  _exit  then
            action$ = 'exit'
            exit routine
          end if
          hour$ = in_time$[1:2]
          minute$ = in_time$[3:4]
          if  val(hour$) > 24  or  val(minute$) > 60  then
            message error: 'Invalid time'
            repeat routine
          end if
          convert_time
          action$ = 'print_date_time'
        end routine
```

```
14000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! P R I N T   D A T E   T I M E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Print the date and time formats to the screen.
        !
        ! Expected:
        !       u_day          = "julian" of entered date
        !       u_day$         = day of week (Monday, Tuesday, etc.)
        !       u_yyyymmdd$    = yymmdd version of entered date
        !       u_mmddyyyy$    = mmddyy version of entered date
        !       u_ddmonyyyy$   = dd-mon-yyyy version of entered date
        !       u_monthddyyyy$ = month dd, yyyy version of entered date
        !                        (padded to 18 characters)
        !       u_time$ = HH:MM AM  or  HH:MM PM
        !
        ! Locals:
        !
        ! Results:
        !       dates are displayed on the screen
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine print_date_time
          print at 6,1: 'Date formats:';
          print bold, at 6,15: u_day; '- Julian date'
          print bold, at 7,16: u_day$; ' - day of week'
          print bold, at 8,16: u_yyyymmdd$; ' - yyyymmdd version'
          print bold, at 9,16: u_mmddyyyy$; ' - mmddyyyy version'
          print bold, at 10,16: u_ddmonyyyy$; ' - dd-mon-yyyy'
          print bold, at 11,16: rtrim$(u_monthddyyyy$); ' - month day, year'
          print at 13,1: 'Time format :';
          print bold, at 13,16: u_time$
          action$ = 'exit'
        end routine

15000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! C O N V E R T   D A T E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Convert the date to various date formats.
        !
        ! Expected:
        !       user_date$ = input date in YYYYMMDD format
        !
        ! Locals:
        !
        ! Results:
        !       u_day          = "julian" of entered date
        !       u_day$         = day of week (Monday, Tuesday, etc.)
        !       u_yyyymmdd$    = yymmdd version of entered date
        !       u_mmddyyyy$    = mmddyy version of entered date
        !       u_ddmonyyyy$   = dd-mon-yyyy version of entered date
        !       u_monthddyyyy$ = month dd, yyyy version of entered date
        !                        (padded to 18 characters)
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine convert_date
          u_day          = days(user_date$)
          u_day$         = day$(u_day)
          u_yyyymmdd$    = date$(u_day, 0)
          u_mmddyyyy$    = date$(u_day, 1)
          u_ddmonyyyy$   = date$(u_day, 3)
          u_monthddyyyy$ = rpad$(date$(u_day, 4), 18)
        end routine
```

Sample Program

```
16000   !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        ! C O N V E R T   T I M E
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        !
        ! Brief description:
        !   Convert the time (hour and minute) to a 12-hour time format.
        !
        ! Expected:
        !       hour$ = hour
        !       minute$ = minute
        !
        ! Locals:
        !
        ! Results:
        !       u_time$ = HH:MM AM  or  HH:MM PM
        !
        !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        routine convert_time
          select case hour$
          case is < '12'
            u_time$ = hour$ + ':' + minute$ + ' AM'
          case is = '12'
            u_time$ = hour$ + ':' + minute$ + ' PM'
          case is > '12'
            z1% = val(hour$)
            z1% = z1% - 12
            u_time$ = lpad$(str$(z1%), 2, '0') + ':' + minute$ + ' PM'
          end select
        end routine
```

```
.screen -
Date? 07041994
Time? 1645

Date formats:  144091 - Julian date
               Monday - day of week
               19940704 - yyyymmdd version
               07041994 - mmddyyyy version
               04-Jul-1994 - dd-mon-yyyy
               July 4, 1994 - month day, year

Time format :  04:45 PM
```

B User Notes

Index

A

Abbreviated INTOUCH statements • 10–7
Access modes • 7–1
ALLOW validation rule • 2–3
Alpha system • 10–3
AREA option • 2–5
ASK channel: CURRENT • 7–4
ASK MARGIN • 1–10
ASK MESSAGELINE • 4–3
ASK PAGESIZE • 4–3
ASK PAINT • 2–5
ASK STRUCTURE • 6–10
ASK WINDOW: CURRENT • 1–6
ASK WINDOW: KEYMAP • 5–5
ASK WINDOW AREA • 1–8
ASK ZONEWIDTH • 1–12
AT row, column • 2–2
ATTRIBUTES option • 2–5
AXP system • 10–3

B

_BACK • 5–4
Boxes • 1–4
Branching in a routine • 9–7
BREAK • 11–2
Break point • 11–2

C

Calling routines • 9–6
CALL SETUP • 6–1
CHANGE$ function • 8–9
Changing characters • 8–9
Changing keyboard values • 5–5
Channel • 7–1
_CHANNEL • 6–9
Channel #0 • 1–10
Channel number • 6–9, 7–1
Checking responses • 5–1
Checking variables • 11–3
Checking variable values • 11–1
CLEAR • 1–2
  AREA option • 1–2
  BOX option • 1–2
Clearing screens • 1–2
CLOSE • 7–1
CLOSE ALL • 6–2, 7–1
CLOSE STRUCTURE • 6–2
Closing files • 7–1
Closing structures • 6–2
Command abbreviations • 10–7
Command recall • 10–6
Compile errors • 11–1
Concepts and key values • 5–1
Conditonal statements • 9–1
Controlling length of input data • 2–4
Controlling message placement • 4–3
Controlling the input format • 2–5
Controlling the screen • 1–1
Converting data • 8–10
CPAD$ function • 8–2
Creating data files • 6–1
Creating menus • 3–1
Creating structures • 6–1
CSET • 8–2
CTRL/B • 10–6
CTRL/V • 10–6
CTRL/Z • 5–7

D

Debugging techniques • 11–1
DEBUG OFF • 11–2
DEBUG ON • 11–2
Default menu item • 3–12
DEFAULT option • 2–2
Defining fields • 6–1
DELAY • 4–5
Determining response time • 2–3
Directives • 3–11
DISPATCH str_expr • 9–6

DO/END DO loop • 9–3
DO/LOOP loop • 9–3
DO WHILE/UNTIL loop • 9–3
DOWN key • 10–6
Drawing a box • 1–4

E

EDIT • 10–4
EDIT$ function • 8–4
EDIT$ function table • 8–6
Editing data • 8–4
Editor, INTOUCH • 9–5, 10–8
EDIT symbol • 10–1
ELAPSED option • 2–3
ELEMENT$ function • 8–6
ELEMENTS function • 8–8
Enabling the debug system • 11–2
END • 10–6
End of file condition • 7–2
EOF option • 6–9, 7–2
Error messages • 11–1
_EXIT • 5–4, 5–7
EXIT DO • 9–4
EXIT FOR • 9–4
Exiting a loop • 9–4
Exported files • 7–4
Extracting Data • 8–6
Extracting records • 6–5

F

Fields in structures • 6–2
File access modes • 7–1
File channels • 7–1
File names for INTOUCH files • 10–3
Files • 7–1
FILL-IN fields • 2–4
Find key • 10–6
FOR/NEXT loop • 9–3
FORMAT$ function • 8–1
Formatting data • 8–1
FRAME OFF • 1–2
FRAME ON • 1–2
Free formatted input areas • 2–5

G

GO • 10–6
GOSUB • 9–6
GOSUB routine_name • 9–6

H

HALT • 10–6, 11–5
_HELP • 5–7
HELP key • 5–7

I

IF/THEN construct • 9–1
_IGNORE • 5–5
.INC extension, files • 10–3
%INCLUDE directive • 10–3
Include files • 10–3
Input
    Customizing • 2–1
INPUT
    AREA • 2–5
    AT row, column • 2–2
    ATTRIBUTES • 2–4
    DEFAULT • 2–2
    ELAPSED • 2–3
    LENGTH • 2–4
    MENUS • 3–1
    PROMPT • 2–2
    SCREEN • 2–5
    text • 2–2
    TIMEOUT • 2–3
    VALID • 2–2
INPUT examples • 2–1
INPUT fill-in field examples • 2–4
Input paint character • 2–5
INPUT statement • 2–1, 2–2, 7–2
.INT extension, files • 10–3
INTOUCH editor • 9–5, 10–1, 10–8
INTOUCH environment • 10–1
INTOUCH file names • 10–3
INTOUCH procedures • 10–4
INTOUCH screen frame • 1–1

INTOUCH structures • 6–1
INTOUCH_EDITOR logical • 10–1, 10–4
.INT_IMAGE extension, files • 10–3
ITERATE DO • 9–3
ITERATE FOR • 9–3

K

Keyboard map • 5–5
Keyboard values • 5–5
Keystrokes, combinations • 5–7
Key usage • 6–8

L

Last user input in _REPLY • 11–3
LCASE$ function • 8–10
LEN function • 8–6
LENGTH option • 2–4
Limiting response time • 2–3
LINE INPUT • 2–2, 7–2
  AREA • 2–5
Line interpreter • 11–1
List of menu directives • 3–11
Locating performance bottlenecks • 11–4
Logical INTOUCH_EDITOR • 10–1, 10–4
Logic flow • 9–1
Loops • 9–3
LPAD$ function • 8–2
LSET • 8–2
LTRIM$ function • 8–4

M

Margins • 1–10
Masking data • 8–1
Menu composition • 3–1
Menu creation hints • 3–2
Menu directives • 3–11
Menu examples • 3–3
Menu item assigned value • 3–14
Menus • 3–1
MESSAGE • 4–1
MESSAGE DELAY • 4–1
MESSAGE ERROR • 4–1
Message row • 1–1
Messages • 4–1
Message types • 4–1
MID function • 8–6
Modifying code at runtime • 11–5

N

NEW • 10–4
Next key • 10–6
NUMBER validation rule • 2–3

O

OLD • 10–4
Opening files • 7–1
Opening structures • 6–2
OPEN statement • 7–1
OPEN STRUCTURE • 6–2

P

Page size • 4–3
Paint character • 2–4, 2–5
Partial key usage • 6–8
PASS • 10–2
Passing VMS commands • 10–2
PATTERN function • 8–6
Performance bottlenecks • 11–4
PF2 key • 5–7, 10–6
PIECES function • 8–8
POS function • 8–6
Positioning user input • 2–2
Prev key • 10–6
Print mask • 8–1
PRINT statement • 7–2
PRINT USING • 8–1
Print zones • 1–11
Processing a response • 5–1
PROMPT option • 2–2

Q

QUOTE$ function • 8–10

R

Reading file data • 7–2
Recalling commands • 10–6
Referencing structure data • 6–2
Reiterating the loop • 9–3
REPEAT$ function • 8–6
REPEAT DO • 9–3
REPEAT FOR • 9–3
Repeating a loop • 9–3
REPLACE • 10–5
Replacing strings • 8–9
_REPLY • 5–1, 5–2, 11–3
Responses • 5–1
Restoring screen areas • 1–8
Restoring screens • 1–6
Returning to a menu item • 3–13
RIGHT$ function • 8–7
Routine branching • 9–7
Routine example • 9–6
Routines • 9–1, 9–5
RPAD$ function • 8–2
RSET • 8–2
RTRIM$ function • 8–4
RUN • 10–5
.RUN extension, files • 10–3
Running VMS commands from INTOUCH • 10–2

S

Sample program • A–1
SAVE • 10–5
Saving screen areas • 1–8
Saving screens • 1–6
Scanning the command list • 10–6
Screen
  Boxes • 1–4
  Clearing • 1–2
  Formatting • 1–10
  Margins • 1–10
  Print zones • 1–11
  Restoring • 1–6
Screen (cont'd)
  Saving • 1–6
Screen appearance • 1–1
Screen frame • 1–1
SCREEN option • 2–4, 2–5
Searching the command list • 10–6
SEG$ function • 8–6
SELECT CASE construct • 9–1
SET channel: CURRENT • 7–4
SET MARGIN • 1–10
SET MESSAGELINE • 4–3
SET PAINT • 2–5
SET STRUCTURE • 6–10
SET STRUCTURE, FIELD: KEY • 6–7
Setting a break point • 11–2
SETUP routine • 6–1
SET WINDOW: CURRENT • 1–6
SET WINDOW: KEYMAP • 5–5
SET WINDOW AREA • 1–8
SET WINDOW KEYSTROKE • 5–5
SET ZONEWIDTH • 1–12
Sorting records • 6–5
SPACE$ function • 8–4
Spelling corrections • 10–8
STATS OFF • 11–4
STATS ON • 11–4
STEP • 11–2
STOP • 10–6
Storing data in files • 7–2
STR$ function • 8–10
Structures • 6–1
Supported keys, combinations • 5–7
Switching to VMS from INTOUCH • 10–1

T

Tab key • 10–7
_TERMINATOR • 5–1, 5–2, 5–3
TIMEOUT option • 2–3
.TMP extension, files • 10–4
TPU editor • 10–1
TRACE OFF • 11–4
TRACE ON • 11–4
TRIM$ function • 8–4

U

UCASE$ function • 8–10
UNFORMATTED output file option • 7–4
Unique file names • 7–1
UNIQUE output file name • 6–9
UNQUOTE$ function • 8–10
Unstructured data • 7–1
UP key • 10–6
User reply stored • 5–2
Using keys to find records • 6–8
Using menus • 3–3
Using parameters to select records • 6–6
Using the line interpreter • 11–1

V

VAL function • 8–10
Validating input • 2–2
VALID option • 2–2
Varible? • 7–3
VMS • 10–1

W

Working with data • 8–1
Working with files • 7–1
Working with structures • 6–1
WRAP$ • 2–5