Programming in R:BASE

Programming In R:BASE

by R:BASE Technologies, Inc.

Welcome to Programming In R:BASE!

This is a guide to the R:BASE programming language. It is organized, as much as possible, in the order of information you need to program applications in R:BASE. It will describe good program structure, including physical appearance, program sections, documentation, and exit methods.

"Using Variables" explains the nature of variable use in R:BASE programming. "Program Communication" describes all of the R:BASE commands that allow you to communicate with a program. "Manipulating Text Strings" describes the commands you use to manipulate and structure text. "Control Structures" describes the two R:BASE looping structures-IF.. .ENDIF and WHILE.. .ENDWHILE. "Accessing Rows in a Table" explains how you use the SET POINTER command to process rows in a program. "Error Handling" describes the R:BASE error handling commands and variables. "Creating and Using Menus" describes the different options for creating a menu system for your R:BASE application. "Debugging Your Program" explains how you use the error handling techniques and other methods to make sure your program executes correctly. "Optimization Techniques" provides some concepts for optimizing your application file code. "Executing External Programs" describes how you can use the LAUNCH command to execute non-R:BASE programs from a program and as an application menu option.

If you are new to R:BASE, work through the R:BASE Tutorial in the main R:BASE program for a basic overview. Then, read this entire document. If you are an experienced programmer, read through this document, looking for new information and for procedures specific to R:BASE programming. Once you become familiar with the commands and how to use them, refer to the Command Index section in the main R:BASE Help for more commands.

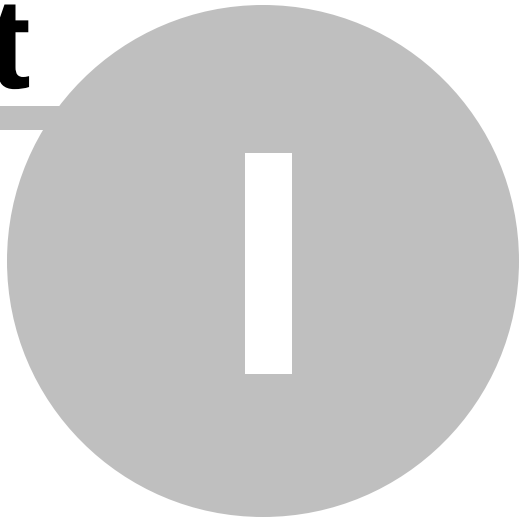
Programming In R:BASE

Printed: May 2024 in Murrysville, PA

First Edition

Table of Contents

Part



1 Introduction to Programming in R:BASE

The main purpose of programming in R:BASE is to save complex database management tasks in a form which may be repeated as needed. Essentially, all of the tasks you might want to accomplish with your database can be done by typing the commands at the R> Prompt. A program, whether it is a stand-alone command file, a form containing button menus, or an application file (.RBA), has a single purpose: to record the steps needed to accomplish a task so that the task may be repeated whenever necessary.

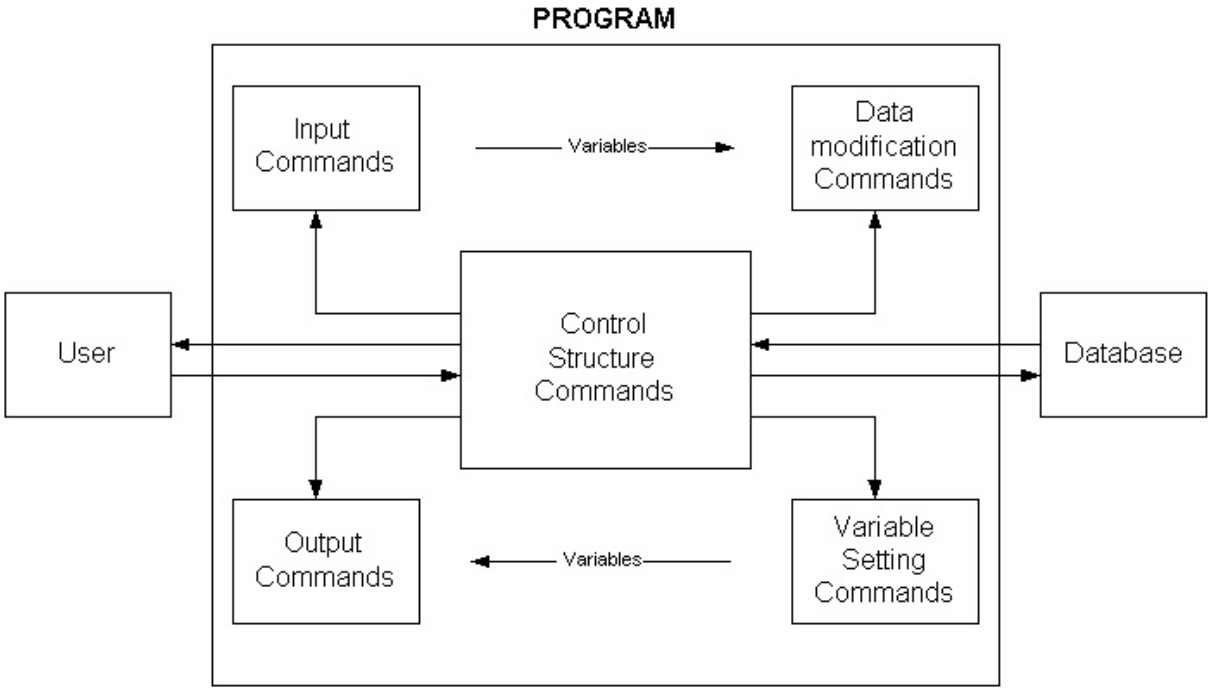
To fulfill the need to repeat tasks, R:BASE provides many different methods to store your series of R:BASE commands, which can be used to create a program of sorts:

- Command Files
- Application Files
- Forms
- Stored Procedures
- External Form Files

R:BASE provides a complement of structural commands which help you design and write the applications that make accomplishing these tasks far easier.

This chapter briefly describes some of the tools R:BASE provides to help you fulfill the requirements of your particular application.

The following figure illustrates the basic requirements for all tasks you want to perform with your database.



- Input commands provide R:BASE with the basic information it needs to perform the task.
- Variables carry data and instructions from the input commands to R:BASE.
- Control structure commands route execution through the various steps needed to accomplish the task. The control structure commands are used to control the input commands, the data modification commands, the variable setting commands, and the output commands.
- Data modification commands add to or change the data in the database.
- Variable setting commands fill variables with the information to be used by the program.
- Output commands display or print the data in a usable format.

You are probably familiar with some or many of the commands. For example, you may already be familiar with the **ENTER** command for using forms or the **LOAD** command for direct entry into a database. These are examples of input commands. You also probably already know how to use the **EDIT** command for using forms or the **UPDATE** command. These are data modification commands. The **SET VARIABLE** command is the most common variable setting command and the **PRINT** and **SELECT** commands are data output commands.

1.1 Program Structure

A command file (program, macro, etc.) or application file is a logical set of instructions which causes a specific task to be performed. Both command and application files must have a logical flow and a structure that fulfills a few basic requirements. First, the environment must be defined; second, the actual task must be performed; and third, the environment should be reset to its original configuration.

The following figure shows the basic program structure; defining the logical flow of a command file or application:

First, you should outline the general shape of a program. This can be done using any standard flowchart method. The most common appears similar to the above flowchart. Each task is written in a box with lines and arrows indicating the flow of the program as it executes. While there are formal flowcharting methods you can use, the method is not as important as the planning. Even a written outline is better than just starting to enter code. It is important to define what a program is to do and to organize how it is to go about doing it.

Keep in mind these general concepts when you actually begin to enter code:

- Document your program with comments (see Documenting the Program)
- Indent code and leave blank lines for easier reading (see the coding examples in this chapter)
- Spell out commands until you can easily recognize all commands by their abbreviations or use the full command names
- Use the debugging techniques described in this chapter (see Debugging Your Program)
- Before testing, make backup copies of tables affected by your program

1.2 Setting the Execution Environment

1.2.1 Connecting to a Database

When you run an application, the required database may not always be connected. If you are not sure whether the correct database will be open every time a program is used, it is a good idea to connect to the database in each program you expect to execute; otherwise, the program will abort when a required database is not connected.

There are exceptions to this practice. If, for example, your main application program executes other programs using the **RUN** or **INPUT** commands, you may wish to omit the **CONNECT** command from the subsidiary programs since this command causes the program to read the disk for the database information. This may also change the R:BASE environment by resetting the default environment. Any time you can avoid disk access in your programs, you allow your program to execute faster.

Your program may be a general purpose program that can operate with more than one database. In this case, you may want the operator to provide the database name. To do this, you use a **DIALOG** command to accept entry of the database name and place the database name in a variable. Use the following commands to do this:

```
DIALOG 'Enter the database name: ' vDBName VEndKey 1
CONNECT .vDBName
```

Notice the dotted variable used in the **CONNECT** command to refer to the contents of the variable.

1.2.2 Setting Message Status

Once you have connected to the database or determined that the database is already connected, you should set the message status commands. These two commands are:

```
SET MESSAGES OFF
SET ERROR MESSAGES OFF
```

The first command suppresses display of normal system messages. These messages are not error messages; they note the result of the execution of an R:BASE command. For example, if you execute a **PROJECT** command, the system responds with the message, "*Successful project operation n rows generated*". This type of message is not displayed if you SET MESSAGES OFF.

The second command suppresses the display of error messages. You might think that you would want to display error messages when a program is executing. If you want R:BASE to display error messages for you, you can leave error message display on. However, if you need to closely control your screen display positions, you should turn off error message display and capture errors in an error variable. See "Error Handling" in this document.

When you first test your program, you may wish to enable these commands until you are relatively certain that your program is executing properly. The displayed messages and error messages can help you to correct (debug) your program.

1.2.3 Setting Variables

The next step in setting the execution environment is to define and set the data types for variables you use in the program. Not all variables need definition at the beginning of the program. However, entering all of your variable definitions in one place in the program helps keep track of variable names and their purposes. Of course, you may not know what all of the variables are until you have written the code to perform the task. But you can add the variable definitions to the top of the program as you go along.

1.3 Documenting the Program

It is a good idea to internally document your program by adding comments within the text of the code as you go along. To do this, you can use several available R:BASE comment formats as shown in the following syntax.

```
-- comment
{comment}
*(comment)
```

R:BASE does not process comments in any manner except to check for double hyphens, matching squiggly brackets, or parentheses. Therefore, you can use the full ASCII character set. You can continue comments onto second and subsequent lines.

As complex expressions are built with parentheses, or even several sets of parentheses, it is recommended that you use either of the first two comment methods. The first option, using 2 hyphens would be used for single line comments. The second option, using the squiggly brackets, would be used for multiline comments.

The following examples are all valid comments:

```
-- Comment on a single line
*( Comment on a single line)

{Comment spread across two lines by continuing the text
on a second or subsequent line}
```

Comments can appear anywhere in a program. However, if you are using **CODELOCK,** the first or second line cannot be used for comments. Comments should not be included in screen or menu blocks or files. Be sure you do not embed an executable command in a multiline comment because the command will not execute. For example, in the following lines, the embedded **SET VARIABLE** command is ignored:

```
{ set the following variable to hold a counter
SET VARIABLE ctr INTEGER for the WHILE loop}
```

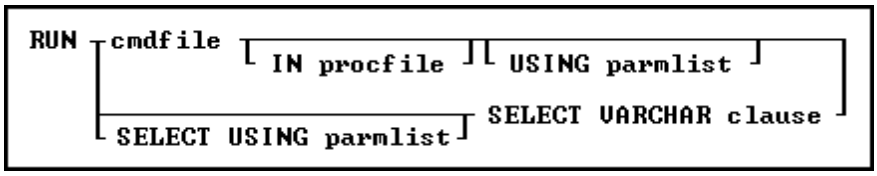
When you are using the R:BASE Editor, you will notice the changes in the font when commenting is used in your command files. The text color for commented lines with turn pink and the text will become italicized. The above valid comments will actually look like this:

```
-- Comment on a single line
*( Comment on a single line)

{Comment spread across two lines by continuing the text
on a second or subsequent line}
```

This highlighting can be altered by changing the R:BASE Editor settings. There are also several other highlighting options for keywords, identifiers, numbers, and more.

1.4 Executing a Command File or Application

Use the **RUN** command to execute command and procedure files. The format of the command depends on the type of file being executed. This is the syntax of the RUN command:



- A command file: *Cmdfile* is a complete file specification including a drive letter and directory if needed. The *In procfile* clause is not used. Use the *USING parmlist* clause if the program requires special parameter passing not handled by global variables.
- A procedure file: *Cmdfile* is the main command block name. The *IN procfile* clause is used if the block being executed is not in the currently open procedure file. *Procfile* is a complete file specification, including a drive letter and directory if needed. The *USING parmlist* clause may be used if the procedure requires parameter passing.

If you are using a **CODELOCK** application, you cannot use the **RUN** command to execute menu or screen blocks or files. Execute menu blocks with the **CHOOSE** command and screen blocks with the **DISPLAY** command. The **INPUT** command can be used to execute independent command files in either ASCII or compiled format but cannot be used to execute a command block in a procedure file.

While you are at the Database Explorer screen, under Command Files, clicking on the "Run File" option will run the command file which is currently selected in the right pane. You can also press the [Enter] key to run the command file, which is currently selected in the right pane.

1.4.1 Passing Parameters

Parameters are values passed to a command file when it is executed using the **RUN** command. Parameters are assigned to a special variable type in the form *%n* where *n* is a number from one (1) to eighteen (18) designating the relative position of the parameter in the optional **USING** clause of the **RUN** command. The **INPUT** command cannot pass parameters.

The following is an example of a simple command file to run at the R> Prompt, which expects parameters to be passed:

```
--simple.cmd
SET VARIABLE concat = (.%1 + .%2)
SHOW VARIABLE concat
CLEAR ALL VARIABLES
```

The command file may be executed using the RUN command like this:

```
RUN simple.cmd USING param1 param2
```

The two passed values held in the system variables *%1* and *%2* after the **RUN** command is executed may be any data types. The following table shows possible RUN commands and the result.

RUN Command	**Result**
`RUN simple.cmd USING 5 10`	15
`RUN simple.cmd USING Ann Baxter`	AnnBaxter
`RUN simple.cmd USING 13.2 $1.98`	$15.18
`RUN simple.cmd USING 99.9 14.205`	114.105
`RUN simple.cmd USING ABC 12.2`	Error condition; the data types are different

R:BASE includes parameter variables in the global variable list and displays them using the **SHOW VARIABLE** command. Each parameter variable name ends with a suffix representing the nesting level of the program using the variable. See "Nesting Considerations" in this chapter for a discussion of program nesting.

The datatype of a parameter variable is determined internally each time a value is passed. Parameter variables cannot be pre-datatyped.

Global variables are accessible to any program. If you run a program from another program, you can use the same variable names, if the variable values are already set, rather than passing the variable data in a parameter list.

1.5 Exiting from the Program

Before you complete the task portion of a program, you can write the code used to exit from the program. This consists of resetting message displays and clearing the variables used in the program. You can also close the database if you connected to it in the program.

There are five ways to exit from an executing program:

1. **RETURN** passes control back either to a calling program or to the R> Prompt, whichever initiated the **RUN** command.
2. **QUIT** exits to the R> Prompt unless the *TO filespec* clause is added. In this case, control is passed to the program file specified by *filespec*. **QUIT TO filespec** cannot be used to execute a command block in a procedure file. It clears all currently open **IF** and **WHILE** blocks and resets the nesting levels to -1. See "Nesting Considerations" in this chapter.
3. **INPUT TERMINAL** or **INPUT SCREEN** returns control to the R> Prompt.
4. When the program has no more code to execute, control returns to a calling program or the R> Prompt, whichever initiated the **RUN** command.
5. **EXIT** exits from R:BASE, disconnecting the currently connected database and terminating all command file nesting levels.

These methods are illustrated in program examples used in this chapter. See especially "Control Structures."

The following illustrates an example of what a program might look like after you have entered the commands to set the environment and exit the program.

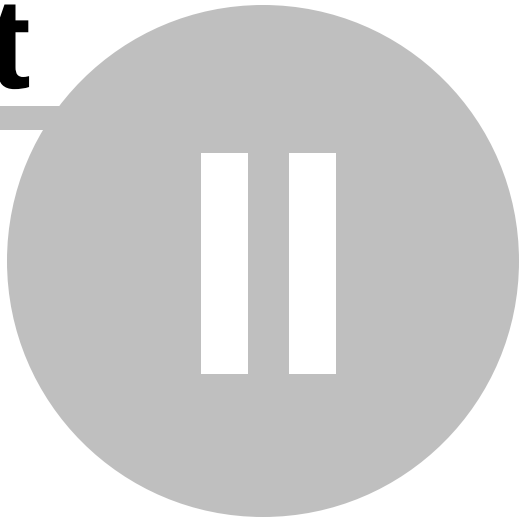
```
{ MVPROG.CMD-Statement of program purpose }
SET MESSAGES OFF
CONNECT ConComp
SET ERROR MESSAGES OFF
SET VARIABLE vToday = .#DATE
--
--Task section of program
--
CLEAR vToday
SET MESSAGES ON
SET ERROR MESSAGES ON
RETURN
```

1. Comment line identifies the program and its purpose (allowed only in the command file format).
2. Sets the program environment by turning off messages, connects to the database, and turning off error messages. Error messages are not suppressed until after opening the database to see if any error results from the **CONNECT** command.
3. Defines any variables needed in the program.
4. Resets the environment to normal R:BASE operations by clearing the variables used in the program and turning on messages and error messages.

Now that you have a general idea of how a program is structured, you can use the next chapter as a reference for dealing with specific tasks. The next section provides more information on using variables in R:BASE. This is followed by sections discussing the use of specific commands. Read "The R:BASE

Commands" section for an overview of the programming commands. Then, see individual sections for specific information and examples on each command's use.

Part



2 Using Variables

In R:BASE programs, variables are either global or error. This section discusses the use of global variables in programming. See "Error Handling" and "Accessing Rows in a Table" for information on error variables.

Variables are symbols used to identify a value that can change or vary. You determine variable values by equating a variable to an expression or by using a variable in a command that automatically defines a variable. Once a global variable is defined, it remains in existence until it is deleted using the CLEAR command or until R: BASE is exited. The contents of a variable may be derived from a constant value, an arithmetic equation, a string expression, or a column value. The following points define the nature of variables used in R:BASE.

- Use variables to hold the results of:
  - A constant value
  - Another variable's contents
  - A column value
  - An arithmetic expression
  - A string expression
  - The value of a system variable
- The data type of a variable is either explicitly defined by the user or implicitly determined by the nature of the data used to set the variable value.
- TEXT type variables may be concatenated (combined with other TEXT variables), constructed (from another TEXT variable using the string functions), or manipulated with the TEXT functions.

2.1 Variable Names

A variable name may be up to 18 characters long in R:BASE X.5. In R:BASE X.5 Enterprise, a variable name may be up to 128 characters long. The name should not include any of the arithmetic operators (+, -, /, *, **, %) or concatenation operators (+, &) because R:BASE may mistake the variable name for an expression or as an **&** variable. There are three different formats for using variable names:

1. Use *VarName* to set a variable value or check the value of a variable as in an **IF...ENDIF** loop.
2. Use .*VarName* if you "want the value" of the variable. For example:

```
SET VAR vDB TEXT = 'MyDB'
CONNECT .vDB
```

   The dot in front of the variable name indicates that R:BASE is to use the "value" of the variable, in this case MyDB, in the **CONNECT** command.

3. Use &*VarName* to hold a list of items. Each item in the list acts independently. When R:BASE sees the **&**, it assumes the variable contains at least a portion of an executable command. This is useful when you want to include a list of column names an operator selects in a command. Briefly, an **&** variable can be used in the following manner:

```
SET VAR vColName TEXT = NULL
SET VAR vColName = 'X'
SET VAR vList TEXT = NULL
WHILE vColName <> 'END' THEN
  DIALOG 'Enter a column name (or END): ' vColName vEnd 1
  IF vColName <> 'END' THEN
    SET VAR vlist = (.vlist & .vColName)
  ENDIF
ENDWHILE
IF SLEN(.vList) > 0 THEN
  SELECT &vList FROM tblname
ENDIF
```

This sequence of commands repetitively requests a column name and adds that name to variable *vList* until the operator enters END. When the process is stopped by the operator entering END, then *vList* is checked to make sure it is not null (at least one column name has been entered) by using the **SLEN** function. The **SELECT** command is executed using the column names entered into *vList* as the columns to display from the table. Notice that *vList* is used as a dotted variable when the names are being added with the concatenation operator & and as an & variable when the **SELECT** command is executed.

Use variables in command code to hold and manipulate values needed to accomplish whatever task a program was designed to perform.

In the following commands, the variable name is used without the dot or & because you are setting or displaying the value of the variable and are not using the already assigned value.

```
CLEAR VarName. . .
CHOOSE VarName. . .
SELECT colname INTO VarName. . .
DIALOG 'Message' VarName. . .
SET VARIABLE VarName. . .
SET ERROR VARIABLE ErrVar
SHOW ERROR ErrVar. . .
SHOW VARIABLE VarName
```

If a variable is being compared to a value, it is used without the dot (Examples 1 and 3), as in the control structure commands. If a variable is the value being compared against (Examples 2 and 4), the dot is used. The "*op*" below represents operator (e.g. >, <, =, etc.).

```
(Example 1)
IF VarName op Value THEN
  task
ENDIF

(Example 2)
IF VarName1 op .VarName2 THEN
  task
ENDIF

(Example 3)
WHILE VarName op Value THEN
  task
ENDWHILE

(Example 4)
WHILE VarName1 op .VarName2 THEN
  task
ENDWHILE
```

2.2 Defining and Deleting Variables

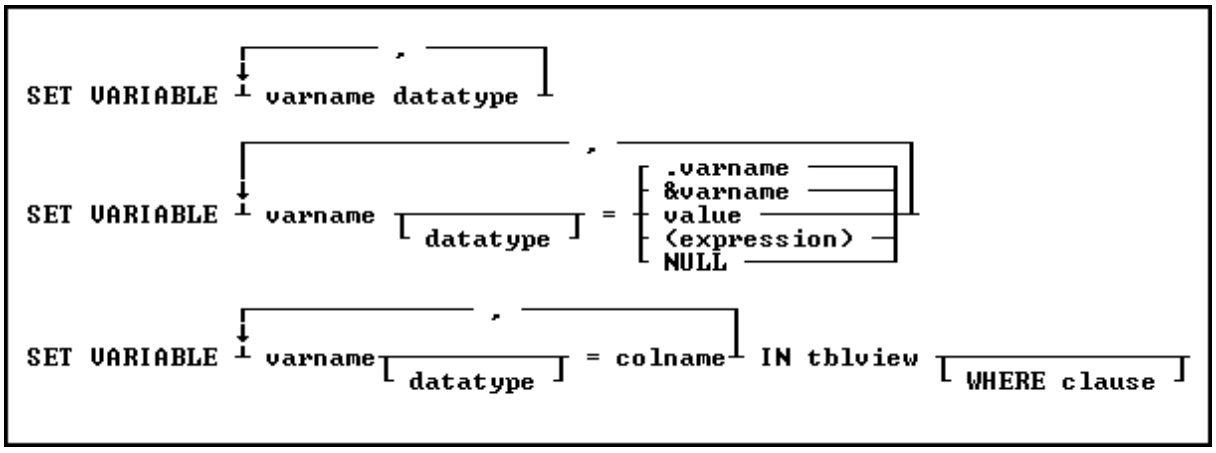
You can define variables by using any of the commands shown in the table below. The data type of a variable is established after the command is executed unless the variable is explicitly typed by a **SET VARIABLE** command or is used in a **CHOOSE** or **SET ERROR VARIABLE** command. The commands shown in the table set variable data types by interpreting the result of the calculation.

Command	Purpose	Examples of Use
CLEAR	Removes variables from the global variable list.	CLEAR varname
SELECT	Calculates on a column and sets the value of varname. The data type depends on the computation that is being	SELECT SUM(col) INTO varname FROM tblname

	done and the data type of the column.	
CHOOSE	Draws a menu and enters the operator's choice into varname. The data type depends on the type of menu. Vertical menus set the variable to INTEGER. Horizontal menus set the variable to TEXT.	CHOOSE varname FROM #LIST
DIALOG	Accepts operator input and enters the value into variable name. The data type must be set to TEXT before the command is issued.	DIALOG 'message' varname vend 1
SET ERROR VARIABLE	Defines a variable to hold the status value of the current operation. The data type is INTEGER.	SET ERROR VARIABLE errvar
SET VARIABLE value	Defines a variable and sets the data type to the type implied by the data.	SET VARIABLE varname = 10.1 SET VARIABLE varname = 'Text string'
SET VARIABLE datatype	Defines a variable with a null value and sets the data type as specified. If the variable is already defined, changes the variable's data type.	SET VARIABLE varname CURRENCY SET VARIABLE varname DATE SET VARIABLE varname DOUBLE SET VARIABLE varname INTEGER SET VARIABLE varname NOTE SET VARIABLE varname REAL SET VARIABLE varname TEXT SET VARIABLE varname TIME

The other commands that use variables do not set the value of the variable. The **IF. . . ENDIF** and **WHILE. . . ENDWHILE** commands use variables whose value is set to determine whether to perform other actions. The **SHOW VARIABLE** and **SHOW ERROR** commands display the current value of a variable.

The **SET VARIABLE** command is most commonly used to define a variable. This is the syntax for the **SET VARIABLE** command:

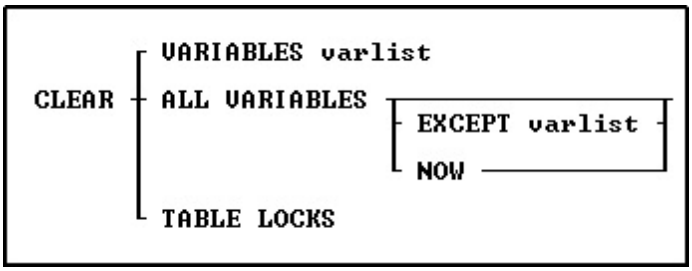


Following are examples of each form of the SET VARIABLE command:

```
SET VARIABLE var1 INTEGER            *(see "Explicit Data Typing")
SET VARIABLE var1 = 10
SET VARIABLE var1 = .var2
```

```
SET VARIABLE var1 = &var2
SET VARIABLE var1 = colname1 IN tblname WHERE colname2 = 1234
SET VARIABLE var1 = colname1 IN #2
SET VARIABLE var1 = ((.var1 + 1) * 12)
SET VARIABLE var1 = 'This is a character string'
SET VARIABLE var1 = ('This is a' & 'concatenation.')
```

Global variables may be deleted individually or all currently defined variables may be deleted together. You can even delete all defined variables and include an exception list. To delete variables, you use the **CLEAR** command. **CLEAR** completely removes the variable, not just the value of the variable. This is the syntax for the CLEAR command:



2.3 Variable Data Types

2.3.1 Implicit Data Typing

In addition to defining a variable value, the **SET VARIABLE** command is used to establish the data type of a variable. Quite often the data type is implied by the variable contents.

R:BASE uses the following rules in the order listed to determine the implicit data type of a variable.

- A **TEXT** data type results if the variable has more than 30 characters or has zero characters (null variable).
- An **INTEGER** data type results if a variable value is a non-decimal number and has nine or less digits.
- A **DOUBLE** data type results if the variable value is numeric with 15 or less digits and one of the following:
  - The variable value contains a decimal point
  - The variable value is a decimal point (interpreted as 0.0)
  - The variable value is in scientific notation using a decimal point (for example, 1.0E6)
- A **DATE** data type results if the variable value matches the current date input format and contains a valid date value.
- A **TIME** data type results if the variable value matches the current time input format and contains a valid time value.
- A **DATETIME** data type results if the variable value matches the current date/time input format and contains a valid date / time value.
- A **CURRENCY** data type results if either of the following is true:
  - The variable value contains the current currency symbol followed by numerals
  - The value is 10 to 17 numeric places
- A **TEXT** data type results if the variable value cannot be assigned any other data type based on the above rules.

2.3.2 Explicit Data Typing

Some variable definitions may not be obvious from the value assigned. You might, for example, use the following **SET VARIABLE** command.

```
SET VARIABLE var4 = 100.50
```

This value could represent a real number, a double-precision number, a currency figure, or a text string. In cases like this, R:BASE has difficulty determining the type of variable you are defining. It is safest, therefore, to preset the data type of a variable whose type may be in question. You cannot assign data types to parameter variables. To explicitly set a variable data type, use the **SET VARIABLE** command in the following form before defining the variable value:

```
SET VARIABLE var4 CURRENCY
```

2.3.3 Changing a Variable Data Type

You can change the current data type of a variable to another type. When you change a variable data type, R:BASE restores the value in the correct internal format for the new type. If the new data type is incompatible with the variable value, R:BASE stores the variable with a null value. The following table illustrates the result of some data type changes:

Original Data Type	New Data Type	Original Variable Value	New Variable Value
TEXT *	any type other than TEXT	This is a string	null value
TEXT	INTEGER	5555.11	5555
TEXT	REAL **	5555.11	5555.11
TEXT	CURRENCY	5555.11	$5,555.11
TEXT	TIME	5555.11	null value
TIME	TEXT	12:30:30	12:30:30
TEXT	DATE	5555.11	null value
DATE	TEXT	03/12/2005	03/12/2005
DATE	INTEGER	01/01/2006	null value
INTEGER	TEXT	5555	5555
INTEGER	REAL	5555	5555.0
REAL	DOUBLE	5555.11	5555.11000000000

* The NOTE data type may be used instead of TEXT
** The DOUBLE data type may be used instead of REAL

2.4 System Variables

R:BASE presets certain variables, which always exist while the program is running. These variables are described in the following table:

Variable Name	Use	Example of Use
#DATE	Holds the current system date.	SET VAR vToday = .#DATE
#TIME	Holds the current system time.	SET VAR vTimer = (.#TIME + 10)
#PI	Holds the value of as a DOUBLE data type (3.14159265358979)	SET VAR vNum = (.#PI *.rad**2)

SQLCODE	Holds the result of the previous SQL command	`SELECT netamount FROM transmaster +`
		`  WHERE netamount IS NULL`
		`IF SQLCODE <> 100 THEN`
		`  --Perform Task here`
		`ENDIF`
		
		The IF...ENDIF condition checks if any rows exist. If valid rows are found, SQLCODE is set to 100, and the control passes to the command after ENDIF. If data is found, SQLCODE is set to 0.

SQLSTATE	Holds a 5-character long return code string that indicates the status of the previous SQL statement	
#NOW	Holds the current system date and time	SET VAR vRightNow = .#NOW

You can verify these values by typing **SHOW VARIABLES** at the R> Prompt.

Form System Variables
When you launch a form and make certain changes or move the cursor, there are R:BASE System Variables that are specific to only Forms.

Variable Name	Description
RBTI_DBGRID_COLUMN	This variable holds the name of focused Column on DB Grid.
RBTI_DIRTY_FLAG	Returns 1 if any DB Control value(s) in form were changed or 0 if nothing was changed.
RBTI_FORM_ALIAS	This variable holds the name of focused form, if used AS alias.
RBTI_FORM_COLNAME	This variable holds the name of focused column DB Control on form.
RBTI_FORM_COLVALUE	This variable holds the value of focused column DB Control on form.
RBTI_FORM_COMPID	This variable holds the value of focused DB/Variable Edit control's Component ID, if defined.
RBTI_FORM_DATATYPE	This variable holds the data type of focused column DB Control on form.
RBTI_FORM_DIRTYVAR	Returns 1 if any Variable Control value(s) in form were changed or 0 if nothing was changed. The variable can only become dirty if a user starts typing using the "keyboard keys" and not by pre-defining or assigning a value (by any means) without actually typing it.
RBTI_FORM_FORMNAME	This variable holds the name of the current form. Particularly useful when using the form-in-a-form technique and multiple forms are running at the same time.
RBTI_FORM_MODE	This variable holds the value of current mode of the form, such as ENTER, EDIT or BROWSE. When the form is brought-up as ENTER USING formname ... the value for RBTI_FORM_MODE will be returned as 'ENTER'. When the form is brought-up as EDIT USING formname ... the value for RBTI_FORM_MODE will be returned as 'EDIT'. When the form is brought-up as BROWSE USING formname ... the value for RBTI_FORM_MODE will be returned as 'BROWSE'.
RBTI_FORM_TBLNAME	This variable holds the name of the current table in a form. This is especially useful when used within a multi-table form.
RBTI_FORM_VARNAME	This variable holds the name of focused Variable Control on form.
RBTI_FORM_VARVALUE	This variable holds the value of focused Variable Control on form.

You can verify these values by typing **SHOW VARIABLES** at the R> Prompt after running a form.

Report System Variables
When you launch a report, the following R:BASE System Variables is created.

Variable Name	Description
RBTI_REPORT_NAME	This variable holds the name of the current report.

You can verify these values by typing **SHOW VARIABLES** at the R> Prompt after running a report.

2.5 Expressions

Expressions are calculations used to determine a value. An expression may contain multiple operators, operands, and functions. Spaces are not required. However, if an expression contains spaces or is a text string, it must be enclosed in parentheses or quotation marks (based on your current QUOTES setting). For clarity, the examples in this manual are shown with internal spaces between operands and operators, and the expression is surrounded with parentheses.

Expressions can be up to 1,000 characters long in R:BASE X.5. In R:BASE X.5 Enterprise, expressions can be up to 2,000 characters long. Expressions can contain up to 50 operators, operands, and functions.

Be sure that constants that contain any separator characters (+, -, *, /, **, &, %, (,), or comma) are contained within quotes marks. This includes text strings. For example, use the following format to include text in an expression.

```
SET VAR vDueDateMess TEXT = ('The Due Date is: ' + (CTXT(.#DATE + 30)))
```

2.6 R:BASE Functions

R:BASE provides a wide range of predefined functions. A function differs from an operator in that a function provides a predefined complex expression to evaluate standard mathematical, trigonometric, financial, or logical functions without requiring the user to enter the formula in a complete R:BASE expression. Function names are reserved words.

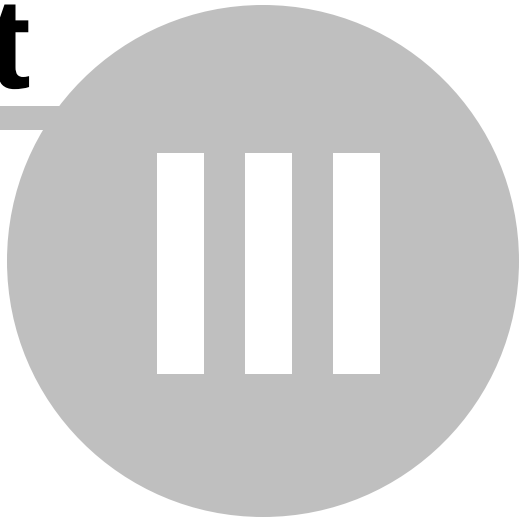
For a complete list of R:BASE Functions, please refer to the "Function Index" located within the main Help and in your R:BASE program directory (default: C:\RBTI\RBGX5 or C:\RBTI\RBGX5E). The file name is **FunctionIndex.PDF**.

2.7 R:BASE Commands

All R:BASE commands are programming commands. Most can be used independently at the R> Prompt, some are used only in define mode, and those discussed in this chapter are generally used only in programs.

For a complete list of R:BASE Commands, please refer to the "Command Index" located within the main Help, or in your R:BASE program directory (default: C:\RBTI\RBGX5 or C:\RBTI\RBGX5E). The file name is **CommandIndex.PDF**.

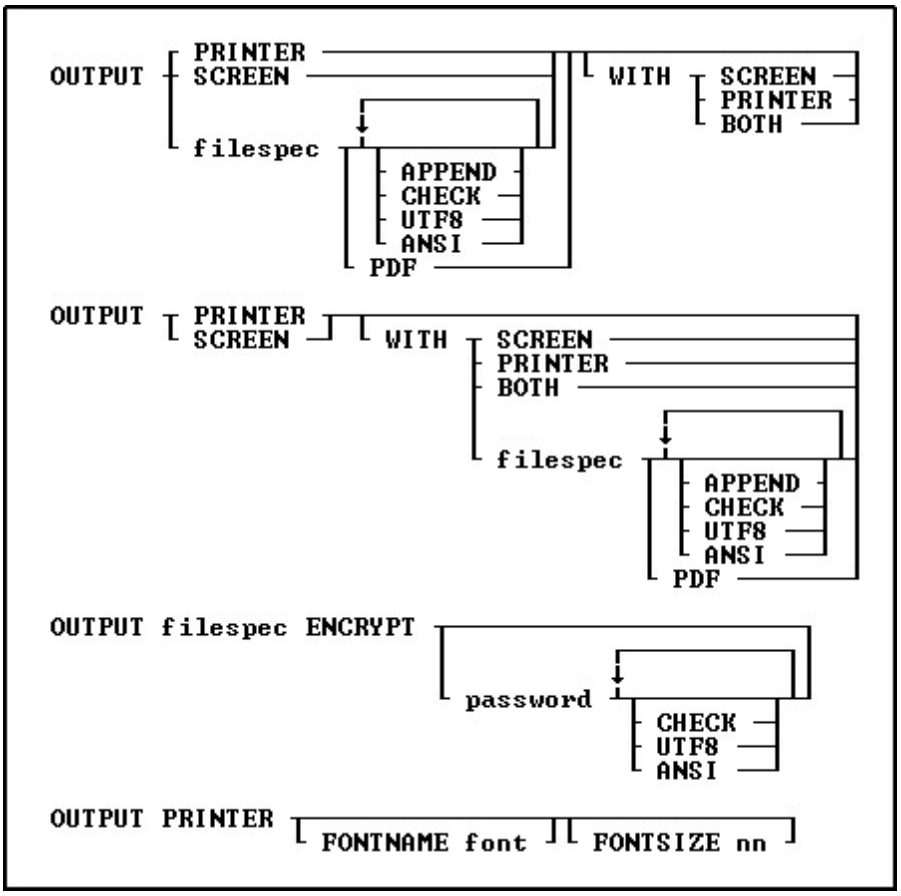
Part



3 Program Communication

3.1 The OUTPUT Command

Most programs must communicate in some manner with one or more output devices. These devices are the computer screen, a printer attached to the computer, or a file. The default output device is the computer screen. To redirect output, enter one or more valid output devices as shown in the syntax. This is the syntax of the OUTPUT command:



Suppose you want to send table data to the printer in your program. You first set the output device to the printer, capture the data records, and then reset the output device to the screen. The sequence of commands is:

```
OUTPUT PRINTER
SELECT firstname, lastname, homephone FROM employee
OUTPUT SCREEN
```

3.2 The PRINT Command

Suppose you want to print a report in your program. The PRINT command offers many different output options for your reports. The following list of output options are available with the PRINT command:

- SCREEN
- PRINTER
- BMP
- EMF
- ETXT (Report Emulation Text)

- GIF
- HTML
- JPG
- PDF
- RTF
- TIFF
- TXT
- WMF
- XHTML
- XLS

Using the OPTION parameter in your PRINT command, you would specify the output options, along with any additional optional parameters for the particular output.

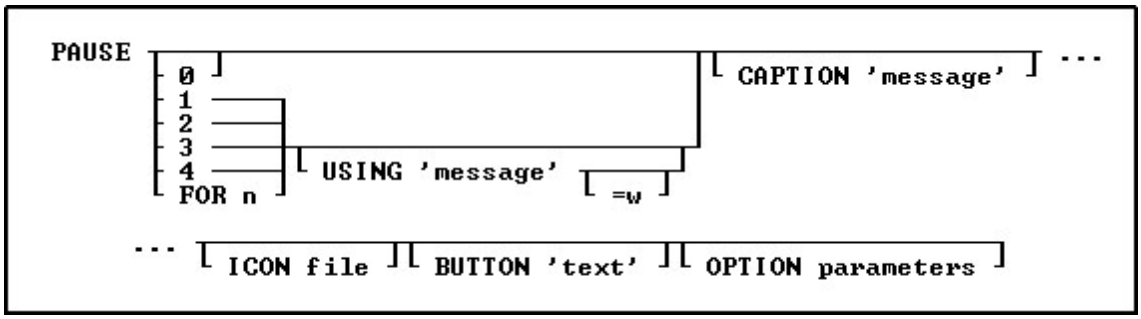
Some command examples are:

```
PRINT reportname WHERE clause... +
  ORDER BY clause ...  +
  OPTION SCREEN|WINDOW_STATE MAXIMIZED

PRINT reportname WHERE clause ... +
  ORDER BY clause ... +
  OPTION PRINTER|COPIES 2|PRINTER_NAME hp laserjet 1230

PRINT reportname WHERE clause ... +
  ORDER BY clause ... +
  OPTION PDF|FILNAME Report.PDF|TITLE My PDF Report
```

This is the same technique you use to print a report from the R> Prompt. In a program you may want to provide more flexibility by allowing the operator to select which output device or devices to use. In this case, you can use a **DIALOG** or **CHOOSE** to provide various output options and then, based on the value of the returned variable, issue a specific OPTION command parameter to the PRINT command. To do this, you need to know how an **IF. . .ENDIF** structure operates. See the chapter on Control Structures.

3.3 The PAUSE Command

Use the PAUSE command to display a message and can also suspend the running of the command file. For assistance with building your PAUSE commands, refer to the R:Pause Builder Plugin.



Options

0
Pauses without a message.

1
Displays "Press OK to continue" or any message using the USING option in a dialog box. The R> Prompt screen then clears with the next keystroke.

2
Same as the 1 option, except the R> Prompt screen does not clear with the next keystroke.

3
Does not pause the running of the command file and therefore does not wait for the next keystroke.

4
If a PAUSE 3 dialog already exists only the message will be repainted to avoid flickering. Otherwise, this is exactly like the PAUSE 3 option.

FOR n
Sets the pause duration in seconds; **n** must be a positive integer. Any keystroke interrupts the pause, regardless of duration. In the absence of a USING clause, no message is displayed. With a USING clause, the message is displayed in a dialog box. When using subsequent PAUSE FOR n USING commands, the CLS command is needed between the PAUSE commands to ensure optional parameters do not carry over from one dialog to the next.

USING 'message'
Displays the specified message in a dialog box. This value can also be passed as a variable.

Using this default parameter the message text is limited to one line. However, if you need to display a multi-line PAUSE window, you can create separate lines with the ASCII characters for a carriage return and indent ([Tab] key). A sample is provided below.

=width
Specifies the wrap width for the dialog box message.

CAPTION 'message'
Specifies the text of the message to display in the dialog box caption. The value can also be passed as a variable.

ICON *value*

Icon "*value*" Parameter	Icon
APPS	
ATTENTION	
CONFIRM	
ERROR	
HELP	
INFO	
QUESTION	
SERIOUS	
STOP	
WARNING	
WINDOWS	

Additional OPTION parameters

Additional parameters are available to increase the visual display of the PAUSE window. To use the graphic PAUSE Builder, choose "Utilities" > "Plugins" > "Internal Plugins" > "PAUSE Builder" from the main Menu Bar. All OPTION parameters and values must be separated by the "|" (pipe) character.

CMPAUSE Setting
In instances where several PAUSE dialogs will appear, perhaps within a loop, the CMPAUSE setting is available to display the messages in a cascade modal mode.

Examples:

```
PAUSE 2 USING 'PAUSE window with the APPS icon.' CAPTION 'PAUSE Command' ICON APPS

PAUSE 2 USING 'You can customize the button message!' CAPTION 'PAUSE Command' ICON
INFO BUTTON 'Your customized message here...'
```

-- Standard PAUSE Command with ICON

```
PAUSE 2 USING 'Message Text' +
CAPTION 'Caption Text' +
ICON HELP OPTION +
BUTTON 'Button Text' +
|BACK_COLOR WHITE +
|MESSAGE_COLOR WHITE +
|MESSAGE_FONT_COLOR GREEN +
|BUTTON_COLOR WHITE +
|BUTTON_FONT_COLOR GREEN +
|TRANSPARENCY 255
```

-- Standard PAUSE Command with custom ICON

```
PAUSE 2 USING 'Message Text' +
CAPTION 'Caption Text' +
BUTTON 'Button Text' +
OPTION ICON_FILE path\directory\filename.bmp +
|BACK_COLOR WHITE +
|MESSAGE_COLOR WHITE +
|MESSAGE_FONT_COLOR GREEN +
|BUTTON_COLOR WHITE +
|BUTTON_FONT_COLOR GREEN +
|TRANSPARENCY 255
```

-- Multi-Line PAUSE Command

```
-- (CHAR(009)) = Tab Key (Indent)
-- (CHAR(013)) = Carriage Return

SET VAR vMsg = +
('Line 1:'+(CHAR(009))+(CHAR(009))&'Contents of Line 1'+(CHAR(009))+(CHAR(013))+ +
 'Line 2:'+(CHAR(009))+(CHAR(009))&'Contents of Line 2'+(CHAR(009))+(CHAR(013))+ +
 'Line 3:'+(CHAR(009))+(CHAR(009))&'Contents of Line 3'+(CHAR(009))+(CHAR(013))+ +
 'Line 4:'+(CHAR(009))+(CHAR(009))&'Contents of Line 4'+(CHAR(009))+(CHAR(013))+ +
 'Line 5:'+(CHAR(009))+(CHAR(009))&'Contents of Line 5'+(CHAR(009))+(CHAR(013))+ +
 'Line 6:'+(CHAR(009))+(CHAR(009))&'Contents of Line 6'+(CHAR(009))+(CHAR(013))+ +
 'Line 7:'+(CHAR(009))+(CHAR(009))&'Contents of Line 7'+(CHAR(009))+(CHAR(013))+ +
 'Line 8:'+(CHAR(009))+(CHAR(009))&'Contents of Line 8'+(CHAR(009))+(CHAR(013))+ +
 'Line 9:'+(CHAR(009))+(CHAR(009))&'Contents of Line 9'+(CHAR(009))+(CHAR(013)))

PAUSE 2 USING .vMsg +
CAPTION 'Caption Text' +
ICON APP +
```

```
BUTTON 'Button Text' +
OPTION BACK_COLOR WHITE +
|MESSAGE_COLOR WHITE +
|MESSAGE_FONT_COLOR GREEN +
|BUTTON_COLOR WHITE +
|BUTTON_FONT_COLOR GREEN +
|TRANSPARENCY 255
```

-- Standard PAUSE Command with custom ICON with Themes

```
PAUSE 2 USING +
'Now you can add themes to PAUSE windows!' +
CAPTION 'New PAUSE Command' ICON INFO +
OPTION themename Longhorn
```

-- PAUSE Command with Meter Progress Bar

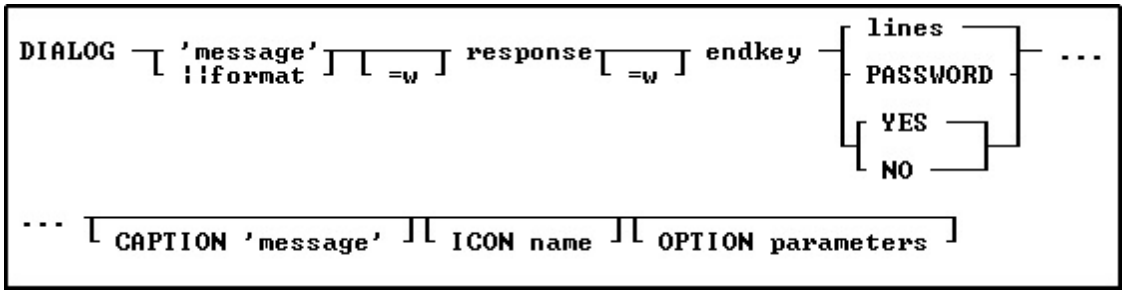
```
PAUSE 3 USING +
'PAUSE Command Text with More OPTIONS - Imagine the Possibilities!' +
CAPTION ' ' +
OPTION METER_VISIBLE ON +
|METER_VALUE .vProgress +
|METER_TYPE BAR3D +
|METER_BACK_COLOR WHITE +
|METER_BAR_COLOR NAVY +
|METER_FONT_COLOR RED +
|BACK_COLOR WHITE +
|MESSAGE_COLOR WHITE +
|MESSAGE_FONT_COLOR GREEN
```

-- PAUSE Command with GAUGE Progress Bar

```
PAUSE 3 USING 'Calculating ... Please Stand By ...'  +
CAPTION '                    Pause 3 with Gauge '  +
ICON WINDOWS  OPTION GAUGE_VISIBLE ON  +
|GAUGE_COLOR RED  +
|GAUGE_INTERVAL 10  +
|MESSAGE_FONT_NAME VERDANA  +
|MESSAGE_FONT_SIZE 10  +
|MESSAGE_FONT_COLOR WHITE  +
|THEMENAME Steel Blue
```

3.4 The DIALOG Command

Use the DIALOG command to display a dialog box on the screen to accept text entry from a user. For assistance with building your DIALOG commands, refer to the R:Dialog Builder Plugin, or the R:Dialog with Buttons Builder Plugin.



Options

'message'
Specifies the text of the message to display in the dialog box. The value can also be passed as a variable.

||format
Also known as EditMask. You can apply the mask specified by the EditMask parameter to the text string specified by the Value parameter.

=w
Specifies the wrap width for the dialog box message.

response
Specifies the variable that will contain the dialog box entry.

=w
Specifies the width for the dialog box entry.

endkey
Specifies the variable containing the final keystroke ([Enter] or [Esc]) in a dialog box.

lines
Specifies the number of lines to display for the text entry in a dialog box.

password
Displays the dialog box entry as asterisks.

YES
Creates a Yes/No dialog box and sets the default response to Yes.

NO
Creates a Yes/No dialog box and sets the default response to No.

CAPTION 'message'
Specifies the text of the message to display in the dialog box caption. The value can also be passed as a variable.

ICON *value*

Icon "*value*" Parameter	Icon
APPS	
ATTENTION	
CONFIRM	
ERROR	
HELP	
INFO	
QUESTION	
SERIOUS	
STOP	

WARNING	
WINDOWS	

Additional OPTION parameters
Additional parameters are available to increase the visual display of the DIALOG window. To use the graphic DIALOG Builder, choose "Utilities" > "Plugins" > "Internal Plugins" > "DIALOG Builder" from the main Menu Bar. All OPTION parameters and values must be separated by the "|" (pipe) character.

Examples

Example 01: (Dialog with custom button text)

```
CLS
DIALOG 'Enter Last Name' vLastName=26 vEndKey 1 +
CAPTION 'Search Employee by Last Name' +
ICON 'APP' +
OPTION MESSAGE_FONT_COLOR BLACK +
|TRANSPARENCY 255 +
|WINDOW_BACK_COLOR WHITE +
|BUTTON_OK_CAPTION '&Search' +
|BUTTON_CANCEL_CAPTION '&Cancel'
```

Example 02: (Dialog with PASSWORD option)

```
CLS
DIALOG 'Enter Password' vPassword=26 vEndKey PASSWORD +
CAPTION 'Database Maintenance' +
ICON 'APP' +
OPTION MESSAGE_FONT_COLOR BLACK +
|TRANSPARENCY 255 +
|WINDOW_BACK_COLOR WHITE +
|BUTTON_OK_CAPTION '&Process' +
|BUTTON_CANCEL_CAPTION '&Cancel'
```

Example 03: (Dialog with Multi-Line Message with the TOP LEFT parameters)

```
-- (CHAR(009)) = Tab Key (Indent)
-- (CHAR(013)) = Carriage Return
CLS
SET VAR vMsg = +
('Line 1:'+(CHAR(009))+(CHAR(009))&'Contents of Line 1'+(CHAR(013))+ +
 'Line 2:'+(CHAR(009))+(CHAR(009))&'Contents of Line 2'+(CHAR(013))+ +
 'Line 3:'+(CHAR(009))+(CHAR(009))&'Contents of Line 3'+(CHAR(013))+ +
 'Line 4:'+(CHAR(009))+(CHAR(009))&'Contents of Line 4'+(CHAR(013))+ +
 'Line 5:'+(CHAR(009))+(CHAR(009))&'Contents of Line 5'+(CHAR(013))+ +
 'Line 6:'+(CHAR(009))+(CHAR(009))&'Contents of Line 6'+(CHAR(013))+ +
 'Line 7:'+(CHAR(009))+(CHAR(009))&'Contents of Line 7'+(CHAR(013))+ +
 'Line 8:'+(CHAR(009))+(CHAR(009))&'Contents of Line 8'+(CHAR(013))+ +
 'Line 9:'+(CHAR(009))+(CHAR(009))&'Contents of Line 9'++(CHAR(013)))
DIALOG .vMsg vYesNo vEndKey YES +
CAPTION ' Your Dialog Caption Here ...' +
ICON 'APP' +
OPTION MESSAGE_FONT_COLOR BLACK +
|TRANSPARENCY 255 +
```

```
|WINDOW_BACK_COLOR WHITE +
|BUTTON_YES_CAPTION '&Start' +
|BUTTON_NO_CAPTION '&Cancel' +
|BUTTON_YES_COLOR GREEN +
|BUTTON_NO_COLOR RED +
|BUTTON_YES_FONT_COLOR WHITE +
|BUTTON_NO_FONT_COLOR WHITE +
|TOP 50 +
|LEFT 50
```

Example 04: (Dialog with button images)

```
CLS
SET VAR vResponse TEXT = NULL
SET VAR vEndKey TEXT = NULL
DIALOG 'DIALOG Message Here ...' vResponse=26 vEndKey 1 +
CAPTION 'DIALOG Caption Here ...' ICON 'APPS' +
OPTION MESSAGE_FONT_COLOR GREEN +
|MESSAGE_FONT_NAME ARIAL +
|MESSAGE_BOLD OFF +
|WINDOW_BACK_COLOR WHITE +
|BUTTON_OK_CAPTION '&Continue' +
|BUTTON_CANCEL_CAPTION 'C&ancel' +
|BUTTON_YES_COLOR WHITE +
|BUTTON_NO_COLOR WHITE +
|BUTTON_YES_FONT_COLOR GREEN +
|BUTTON_NO_FONT_COLOR RED +
|BUTTONS_SHOW_GLYPH ON
```

Example 05: (No Caption Window)

```
CLS
DIALOG 'DIALOG Message Here ...' vResponse=26 vEndKey 1 +
CAPTION 'DIALOG Caption Here ...' ICON APPS +
OPTION MESSAGE_FONT_COLOR GREEN +
|WINDOW_CAPTION OFF +
|MESSAGE_FONT_NAME ARIAL +
|MESSAGE_BOLD OFF +
|WINDOW_BACK_COLOR WHITE +
|BUTTON_OK_CAPTION '&Continue' +
|BUTTON_CANCEL_CAPTION 'C&ancel' +
|BUTTONS_BACK_COLOR WHITE +
|BUTTON_FONT_COLOR GREEN
```

Example 06: (Dialog window using Themes)

```
DIALOG 'DIALOG Message Here ...' vResponse=26 vEndKey 1 +
CAPTION 'DIALOG Caption Here ...' ICON 'APPS' +
OPTION THEMENAME Longhorn
```

Part

4 Manipulating Text Strings

R:BASE offers a variety of methods for manipulating the contents of TEXT variables. You can perform the following text manipulation functions with R:BASE:

- Combine variables using the concatenation operators + or &
- Fill, locate, retrieve, strip, and otherwise manipulate text strings using the R:BASE text and string manipulation functions (see the Functions Index for a complete list of functions)
- Move text or any portion of text from one variable to another
- Use a variable to contain a list of items such as two or more column names

4.1 Concatenating Text Strings

You can combine two or more text strings into a single variable using either of the concatenation operators + or &. The table below illustrates the use of concatenation. Parentheses are not required as long as the expression has only two operands.

Variable Values	Concatenation	Result
var1 = gas	SET VAR vCombine = (.var1 + .var2)	combine = gaslight
var2 = light	SET VAR vCombine = (.var1 & .var2)	combine = gas light
	SET VAR vCombine = (.var2 + 'bulb')	combine = lightbulb
	SET VAR vCombine = ('sees the' & .var2)	combine = sees the light
	SET VAR vCombine = (.var1 + "," & .var2)	combine = gas, light

The concatenation operators allow complex expressions; that is, more than one operator may be used in an expression. You can mix and match the + and & operators as shown in the last example above.

4.2 Using the Text Functions

R:BASE provides the following text manipulation functions:

Function	Description
CHAR	Returns the TEXT value of an ASCII decimal code
CTR	Centers a string by padding with blanks on either side
CTXT	Converts various non-text values into TEXT characters
FORMAT	Applies picture format to text strings
ICAP	Converts only the first word in a text string with an initial capital letter
ICAP1	Converts a text string to lower case with an initial capital letter on the first word
ICAP2	Converts a text string to lower case with an initial capital letter on each word
ICHAR	Returns the ASCII decimal code for single text characters
IHASH	Creates an INTEGER value from a text string
ISALPHA	Checks for an alphanumeric value within the first character of a text string
ISDIGIT	Checks for a numeric value within the first character of a text string
ISLOWER	Checks for a lower case letter within the first character of a text string
ISSPACE	Checks for a space within the first character of a text string
ISTR	Returns the ASCII integer value for a specified value in a text string

ISUPPER	Checks for a upper case letter within the first character of a text string
ITEMCNT	Counts the number of items in a text string separated by the current delimiter
LJS	Left-justifies a string by padding with blanks on the right
LTRIM	Trims leading blanks from text, returning a text string
LUC	Converts a text string to upper case
RJS	Right-justifies a string by padding with blanks on the left
RTRIM	Trims trailing blanks from text, returning a text string
SFIL	Fills a character string with a specified character
SGET	Returns a selected set of characters from a text string
SKEEP	Keeps characters within the source string, using case sensitivity
SKEEPI	Keeps characters within the source string, without using case sensitivity
SLEN	Returns the length of a character string
SLOC	Returns the position of a substring within a text string
SLOCP	Locates the exact position of a given string and occurrence in a text string
SMOVE	Moves a substring of a specified length from one text string beginning at any position to another text string at any position

Refer to the Functions Index for a description and examples of all R:BASE Functions. This section shows an example using the **CTXT** and **SLEN** functions.

CTXT and **SLEN** are single operand functions. **CTXT** converts various non-text values into TEXT characters. **SLEN** returns an INTEGER value equivalent to the length of a text string. The following syntax shows an example of how to use these two functions to determine the length of a TEXT string. Assume the largest value contained in column col1 in table tbl1 is the INTEGER value 3642960.

```
SELECT MAX(col1) INTO var1 FROM tbl1            -- 1.
SET VAR tvar1 = (CTXT(.var1))                   -- 2.
SET VAR tlen = ((SLEN(.tvar1)) + 1)             -- 3.
SET VAR tcol = ('col1=' + CTXT(.tlen))          -- 4.
SELECT &tcol col2 col3 FROM tbl1 ORDER BY col1  -- 5.
```

This sequence of commands does the following:

1. Finds the greatest value of col1: 3642960.
2. Converts that value to text: "3642960".
3. Finds the length of the text value and adds 1 to allow a space between columns: 8.
4. Concatenates the column name and length (converted to text) in variable *tcol*.
5. Executes the SELECT command. If the maximum value in col1 is seven characters long then tcol contains the value col1 = 8. The ampersand (&) before the variable tcol instructs R:BASE to parse the variable-that is, to see it as three parts (col1, =, and 8) and not merely as a text string.

4.3 Moving Text Between Variables

The **SMOVE** function is used to take text from any position in a TEXT variable and to move that text to any position in another TEXT variable. The syntax of the **SMOVE** function is:

```
SMOVE(text,pos1,nchar,string,pos2)
```

Text is the text string containing the characters to copy, *pos1* is the position of the first character of the text string to copy, *nchar* is the number of characters to copy, *string* is the text string to which the characters are to be copied, and *pos2* is the position to which to copy the characters.

For example, suppose the TEXT variable *textvar1* has the value ABCXXXDEF and *textvar2* contains GHIJKL. If you want to move the XXX from *textvar1* into the first through third positions of *textvar2*, use **SMOVE** like this:

```
SET VAR textvar2 = (SMOVE(.textvar1,4,3,.textvar2,1))
```

The three characters XXX begin in the fourth position of variable *textvar1*. These three characters are moved to variable *textvar2* beginning at character position 1. After this command is executed, *textvar1* still contains ABCXXXDEF and *textvar2* contains XXXJKL.

If you do not want to change the value of the original variable *textvar2*, you can assign the new text value to another variable like this:

```
SET VAR textvar3 = (SMOVE(.textvar1,4,3,.textvar2,1))
```

After this command is executed, *textvar1* and *textvar2* are unchanged and *textvar3* contains XXXJKL.

The **SMOVE** function may be used with dates if the date is first converted to a TEXT data type. The following commands show how the month and year are separated from the full date format and then used to select dates falling in the same month. This example assumes that the date format is MM/DD/YYYY.

```
SET VAR thismnth = .#DATE
SET VAR thismnth TEXT
SET VAR thismnth = ((SMOVE("??",1,2,.thismnth,4))+ 1)
SELECT ALL FROM mthtbl WHERE coldate = .thismnth
```

1. Places the current date in variable *thismnth*.
2. Converts the date value to TEXT.
3. Moves two question marks into the day position in variable *thismnth* so that it looks like this: 06/??/2004.
4. Selects rows from mthtbl using *thismnth* as a mask to compare the dates in coldate against those in June 2004.

Part V

5 Control Structures

Unlike other R:BASE commands, whose purpose is to manipulate data, control structure commands are used to control the execution of a program. R:BASE programming language has four control structures: **IF...ENDIF**, **WHILE...ENDWHILE**, **SWITCH...ENDSW**, and **GOTO** and **LABEL**. In addition, external calls using the **RUN** command may be considered a control structure.

The **IF...ENDIF** structure allows you to determine whether or not to execute a sequence of commands. The **WHILE...ENDWHILE** structure allows you to repeat a sequence of commands until a specified set of conditions is no longer true. These structures provide conditional execution of commands. **SWITCH...ENDSW** define a block of possible actions to take depending on the value of an expression. The fourth control structure, **GOTO**, passes control from one part of the program to a specific labeled part of the program.

5.1 IF...ENDIF Processing

Use an IF...ENDIF structure in a command file to cause a block of commands to be run when the specified conditions are met. IF structures can be nested within other IF and **WHILE** structures. See "Nesting Considerations" for more information.

This is the syntax of the **IF. ..ENDIF** structure:

```
IF condlist THEN
   then-block
ENDIF
```

```
IF condlist THEN
   then-block
ELSE
   else-block
ENDIF
```

The simplest form of the structure contains only the *then-block* command sequences. These commands are executed if the conditions specified in the *condlist* are true. There may be up to 10 conditions in the *condlist* with AND, OR, AND NOT, or OR NOT separating the conditions. The **IF...ENDIF** conditions are processed in the same way as **WHERE** clause conditions.
The following table lists the possible conditions that may be used in an IF structure.

Condition	**Description**
varname IS NULL	The value of the variable is null.
varname IS NOT NULL	The value of the variable is not null.
varname CONTAINS *'string'*	The variable has a TEXT data type and contains a '*string*' as a substring in the variable value.
varname NOT CONTAINS *'string'*	The variable has a TEXT data type and a *'string'* is not contained as a substring in the variable value.
varname LIKE *'string'*	The variable equals a '*string*.' A *'string'* can contain wildcards.
varname NOT LIKE *'string'*	The variable does not equal the *'string*'. A '*string'* can contain wildcards.
varname BETWEEN *value1* AND *value2*	The value of the variable is greater than or equal to *value1* and less than or equal to *value2*. The variable and the values must be the same data type.
varname NOT BETWEEN *value1* AND *value2*	The value of the variable is less than *value1* or greater than *value2*. The variable and the values must be the same data type.
varname IN *(valuelist)*	The value of the variable is in the value list.
varname NOT IN *(valuelist)*	The value of the variable is not in the value list.
item1 op *item2*	*Item1* has the specified relationship to *item2. Item1* can be a column name, value, or expression; *item2* can be a column name, value, or expression.

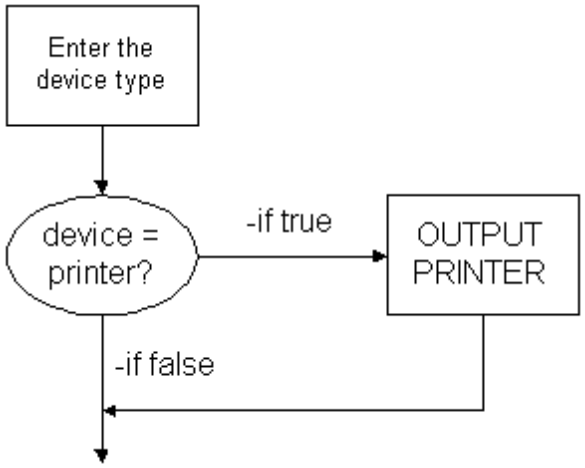
The valid operators (*op*) for the conditions in an **IF...ENDIF** structure are listed in the table below. Do not use wildcard characters with these operators.

Operator	Description
=	Equals
<=	Less than or equal to
>=	Greater than or equal to
<	Less than
>	Greater than
<>	Not equal

An example of a single condition **IF** structure follows:

```
DIALOG "Enter the output device: " vDevice vEndKey 1
IF vDevice = 'PRINTER' THEN
  OUTPUT PRINTER
ENDIF
```

The following flowchart illustrates the flow of this structure.



An example of a multiple condition IF structure follows:

```
IF vFrstName = 'Mary' OR vFrstName = 'John' AND vLstName = 'Smith' THEN
  --
  -- task
  --
ENDIF
```

Either the first and third or second and third conditions in the list of conditions must be met before the commands contained in the **IF** structure are executed. The OR provides the option of meeting only one of the first two conditions. If the total condition evaluates as false, then the **IF** structure commands are not executed. For the combined set of conditions to be true, *vFrstName* must be either Mary or John and *vLstName* must be Smith.

A simple **IF...ENDIF** structure may be extended on a single line in the program. For example:

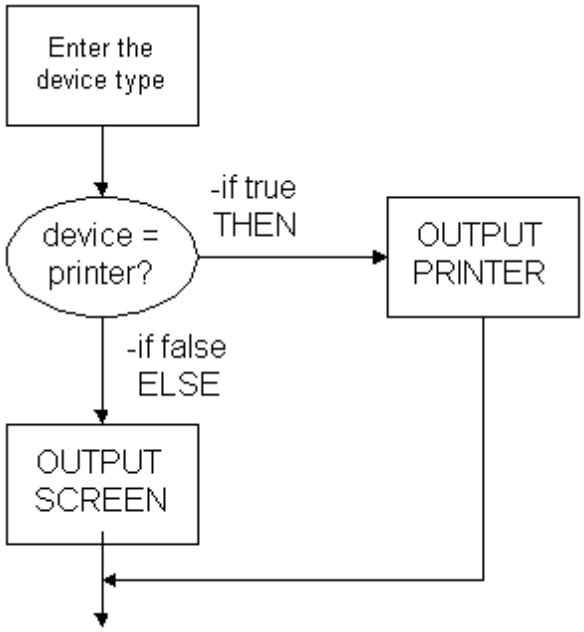
```
IF var2 IS NULL THEN; SET VAR var2 = 'Y' ; ENDIF
```

INCLUDING AN ELSE COMMAND

If an **ELSE** is included in the **IF** structure, then control is passed through either of two mutually exclusive command blocks. This allows you to retain absolute control no matter what occurs during execution. The following command syntax shows an example of an **IF** structure with an **ELSE** block:

```
DIALOG "Enter the output device: " vDevice vEndKey 1
IF vDevice = 'PRINTER' THEN
   OUTPUT PRINTER     -- if PRINTER entered
ELSE
   OUTPUT SCREEN      -- all other cases
ENDIF
```

The following flowchart illustrates the flow of this structure.



The conditions for execution of the then-block are the same as the simple **IF** structure used without an **ELSE**. The difference is that you also provide commands for execution if the conditions are false. In the above example, if the operator enters any value for device except the word **PRINTER**, the command **OUTPUT SCREEN** is executed. This construct allows you to determine a default entry value. The same technique can be used for any operator prompt.

When an **ELSE** is included in the **IF...ENDIF** structure, you must enter each part of the command on separate lines as shown in the syntax diagram.

5.2 WHILE...ENDWHILE Processing

Use **WHILE...ENDWHILE** structures when you want to repeat a group of command lines until a condition or set of conditions is no longer true. This is sometimes referred to as looping. This is the syntax of the **WHILE** structure:

```
WHILE condlist THEN
   while-block
ENDWHILE
```

Up to 10 conditions can be included in the condlist with the conditions separated by AND, OR, AND NOT, or OR NOT. These conditions are of the same type as in the **IF...ENDIF** structure.

Condition	Description
varname IS NULL	The value of the variable is null.
varname IS NOT NULL	The value of the variable is not null.
varname CONTAINS *'string'*	The variable has a TEXT data type and contains a '*string'* as a substring in the variable value.
varname NOT CONTAINS *'string'*	The variable has a TEXT data type and a *'string'* is not contained as a substring in the variable value.
varname LIKE *'string'*	The variable equals a '*string*.' A *'string'* can contain wildcards.
varname NOT LIKE *'string'*	The variable does not equal the *'string*'. A '*string'* can contain wildcards.
varname BETWEEN *value1* AND *value2*	The value of the variable is greater than or equal to *value1* and less than or equal to *value2*. The variable and the values must be the same data type.
varname NOT BETWEEN *value1* AND *value2*	The value of the variable is less than *value1* or greater than *value2*. The variable and the values must be the same data type.
item1 op *item2*	*Item1* has the specified relationship to *item2. Item1* can be a column name, value, or expression; *item2* can be a column name, value, or expression.

WHILE loops can be nested within other **WHILE** or **IF** loops. See "Nesting Considerations" for more information.

BREAKING OUT OF A WHILE LOOP

Under some circumstances, you may want to allow an early exit from the **WHILE** loop processing before the conditions which started the **WHILE** loop become false. This most commonly occurs when you use a **WHILE** loop simply as a device to speed processing since all command lines within a **WHILE** loop are retained in memory. The following command syntax shows how you can use the **WHILE** loop to optimize processing and how to break out of the **WHILE** loop processing

```
SET VAR vLoop = 0                                    1.
WHILE vLoop = 0 THEN                                 2.
--
--
  DIALOG 'Do you want to continue?' +                3.
    vEnd vKey YES
  IF vEnd = 'NO' THEN                                4.
    BREAK
  ENDIF
  --
  --
  --
ENDWHILE
```

1. Defines a variable and sets its value to 0. This variable is used to continue the **WHILE** loop processing until the loop is broken by operator intervention.
2. The commands contained within the **WHILE** loop are executed as long as the value of variable *vLoop* is 0 or the operator does not answer NO to the prompt.
3. The operator is asked whether to continue or to stop with **DIALOG** command. The answer is placed in variable *vEnd*.

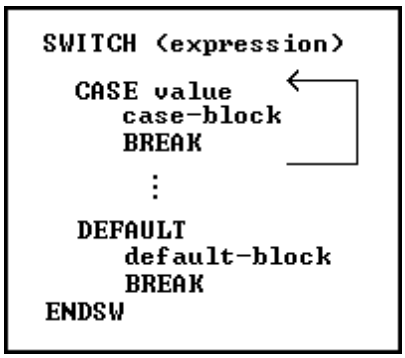
4. The value of *vEnd* is checked and, if the operator has asked to stop, the **BREAK** command exits from the **WHILE** loop.

The following commands may be used to exit from a WHILE loop:

- **BREAK**
- **QUIT**

5.3 SWITCH...ENDSW Processing

SWITCH...ENDSW structures are composed of a series of options. Use the SWITCH...ENDSW command in a program to define a block of possible actions to take depending on the value of an expression. The syntax diagram below shows the entire SWITCH...ENDSW structure, including the **SWITCH** value, **CASE** blocks, and the **DEFAULT** block.



The SWITCH Expression

SWITCH defines the expression to be compared. You can have multiple comparisons, so ENDSW defines the end of the comparisons. The SWITCH expression result must be either an INTEGER or a TEXT data type. The SWITCH expression can be a calculation, constant value, or variable. Any length of text can be compared, but only the first 30 characters are checked in each CASE block.

CASE Blocks

A CASE block consists of three parts: the CASE comparison, the commands following each comparison, and the BREAK statement.

CASE comparisons must be the same data type as the SWITCH expression result - either INTEGER or TEXT. A CASE value cannot be an expression, but must be a constant value or a variable. You can have multiple CASE comparisons to run a single set of commands. For an example of how to use multiple comparisons, see "**Examples**" below.

The commands following a CASE comparison can include any R:BASE command, including a nested SWITCH…ENDSW structure. You can nest as many SWITCH…ENDSW structures as memory allows.

Use a BREAK statement as the last command in a CASE block to exit from the SWITCH…ENDSW structure. The BREAK command stops R:BASE from checking any additional CASE comparisons.

The DEFAULT Block

You can have only one DEFAULT block for each SWITCH…ENDSW structure. A DEFAULT block provides a set of commands to run if none of the CASE comparisons is valid; make the DEFAULT block the last statement block in a SWITCH...ENDSW structure. If a CASE block follows a DEFAULT block, R:BASE generates a warning.

Example

The following SWITCH...ENDSW structure uses a date entered in a **DIALOG** command in the expression. The **TDWK** function calculates day of the week as text from the date stored in *vDay*.

```
DIALOG 'Enter a date: ' vDay vEndKey 1
SWITCH (TDWK(.vDay))
   CASE 'Saturday'
   CASE 'Sunday'
      PAUSE 2 USING 'This is a weekend day.'
      BREAK
   DEFAULT
      PAUSE 2 USING 'This is a weekday.'
      BREAK
ENDSW
```

If you entered 12/22/2007 when prompted for the date, the first CASE comparison would check whether the day of the week is the word *Saturday*. Because the word is *Saturday*, R:BASE would display the PAUSE message window stating you picked a weekend day. The BREAK command prevents R:BASE from processing the rest of the commands in the SWITCH...ENDSW structure.

If the date entered is not *Saturday* or *Sunday* - for example, 12/21/2007 - the information in the DEFAULT block would display the PAUSE message window stating you picked a weekday.

5.4 Passing Control with GOTO and LABEL

The GOTO and LABEL commands work together to transfer control from one part of the program to another. GOTO specifies a label name which is repeated in the LABEL command. The LABEL command only indicates the point to which control is passed. It has no other effect on processing. The label name is a text string 1-18 characters long. This is the syntax of the GOTO and LABEL commands:

```
GOTO lblname
LABEL lblname
```

Use a WHILE loop whenever possible either in place of or surrounding a GOTO/LABEL to speed processing. When a GOTO is executed, R:BASE starts searching for the corresponding GOTO label from the next line following the GOTO. If it reaches the bottom of the file without finding it, R:BASE continues its search at the top of the file. Preferably, your program should execute a GOTO command only once and should not use it for looping. Most commonly, GOTO is used to:

- Return to the beginning of execution
- Bypass code under specified conditions
- Pass control to the exit commands under specified conditions

The LABEL may precede or follow the GOTO command but must be in the same file as the GOTO that references it. The following command syntax shows an example of the GOTO command.

```
LABEL lbl_top
.
.
.
DIALOG 'Do you wish to quit?' vChoice VKey NO
IF vChoice = 'NO' THEN
  GOTO lbl_top
ENDIF
.
```

```
.
```

In this example, R:BASE must search down through the rest of the file and then start searching from the top of the file to find its matching LABEL. This construction, although valid, could be replaced by a **WHILE** loop structure which would process much more quickly. The following command syntax shows a preferable construction for prompting repetitively.

```
SET VAR vChoice = 'NO'
WHILE vChoice = 'NO' THEN
  .
  .
  .
  DIALOG 'Do you wish to quit?' vChoice VKey NO
  IF vChoice = 'YES' THEN
    BREAK
  ENDIF
.
.
ENDWHILE
```

5.5 Nesting Considerations

Nesting means locating a control structure within a similar control structure-for example, an **IF** structure within another **IF** structure or an IF within a **WHILE** structure. Command files, **IF** structures and **WHILE** loops can be nested.

For command files, the maximum number of nesting levels depends on your **FILES** setting in the database settings. **SET FILES** sets the maximum number of files that can be open at a time. The maximum, depending on available memory, is 255. These levels are nested **RUN** commands. Keep track of the nesting level so that the appropriate exit method is chosen.

For example, if you run another command file from within a command or procedure file, R:BASE adds one to the nesting level for the **RUN** command. If you exit from that nesting level using **QUIT**, control does not return to the command following the RUN command in the first program, where you might have wanted it, because **QUIT** clears all nesting levels. More appropriately, **RETURN** is the exit command to use. Control is returned to the next line in the calling command file and the nesting counter is decremented by one. If you are chaining files rather than nesting them, you are not limited in the number of **RUN** commands you can use.

These nesting levels increase until a command is processed that decreases the nesting level. The commands that increase and decrease the nesting level are shown in the following table.

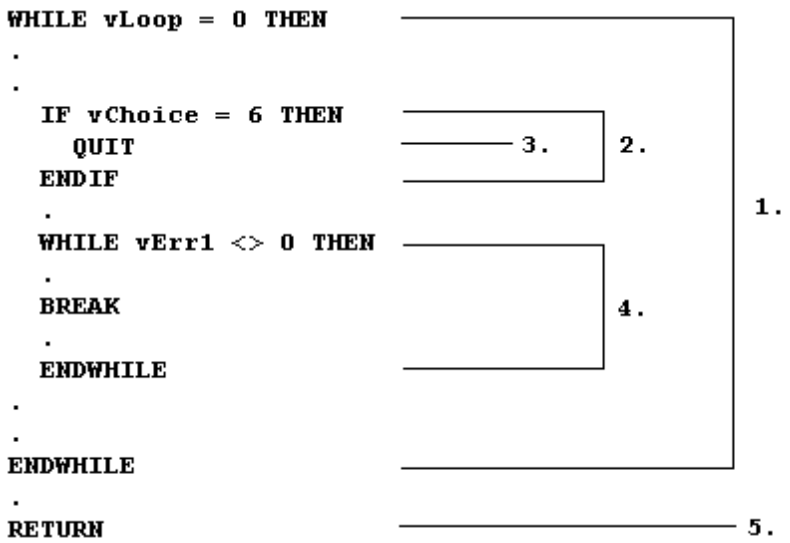
Command	**Effect on WHILE Level**	**Effect on IF Level**	Effect on Command File Level
WHILE	+1		
ENDWHILE	-1		
BREAK	-1		
IF		+1	
ENDIF		-1	
INPUT		+1	
RUN		+1	
RETURN		-1	
QUIT	-all	-all	-all
QUIT TO	-all	-all	-all
End of file	-all	-all	-1*

+1 - increase one nesting level
-1 - decrease one nesting level
-all - clear all nesting levels

***** In the executing file only

The **GOTO** command can also affect nesting levels under certain circumstances. As R:BASE searches through the file for the matching label, each **WHILE** or **IF** encountered increases the nesting level and each **ENDWHILE** or **ENDIF** encountered decreases the nesting level. This occurs even though the control structures passed over are not executed.

The following command syntax shows some nesting examples with a discussion of what occurs in each.



1. The **WHILE** nesting level increases by 1 to 1 if this is the first **WHILE**. The **ENDWHILE** decreases the level by 1. (A nesting level value of 0 means that no loops or structures are open).
2. The nesting level increases by 1 at the **IF** to 1 and decreases by 1 at the **ENDIF** back to 0.
3. **QUIT** clears all nesting levels and the file is exited.
4. The **WHILE** nesting level increases by 1 to 2. When the **BREAK** command is executed, the **WHILE** nesting level decreases to 1 and this **WHILE** loop is exited.
5. All nesting levels decrease to 0 and the file is exited.

Part

6 Accessing Rows in a Table

An especially useful feature of the R:BASE programming language is used to access and update data in your database. You will find it to be the foundation of many of the R:BASE programs you write. This feature is available using several different commands:

- **SELECT**
- **INSERT**
- **UPDATE**
- **DECLARE CURSOR**
- **LOAD**
- **SET VARIABLE**

Some of these commands can be used in combination with others. For example, DECLARE CURSOR uses SELECT, INSERT can use SELECT as one of the options, and SELECT can use itself as a sub-SELECT.

The commands INSERT and LOAD are used to specifically "add" data to an R:BASE table. The UPDATE command is used to "alter" the values of data in tables. The SELECT, and SET VARIABLE commands are used to capture data so you can perform additional tasks, although SELECT is more powerful and is more commonly used. The DECLARE CURSOR command is used to create a cursor that points to a row in a table.

6.1 SELECT

Use the **SELECT** command to display rows of data from a table or view. To display the data in the order you want, modify the SELECT command by using various clauses.

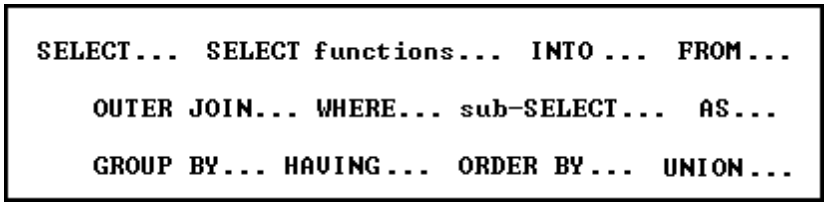
The SELECT command is a very powerful data retrieval command. By learning this command, and all of it's parts you can greatly enhance your ability to work with any other R:BASE command that uses those same portions. For example, learning to use a WHERE clause with SELECT will help you work with WHERE clauses on other commands.

You can use the SELECT command to do the following:

- Display rows of information from a table or view
- Extract information from a table or view by using a nested SELECT command (a sub-SELECT statement) in a WHERE command
- Extract information from a table or view by using a SELECT clause in another command

A SELECT command is essentially a process of elimination. A SELECT command can contain a number of clauses (two are required), each of which begins with a keyword, such as "FROM" or "WHERE."

The diagram below shows the different clauses in a SELECT command.



Each of the SELECT clauses has a specific purpose for determining what data you want. The operators are processed in the order in which they appear in the preceding diagram.

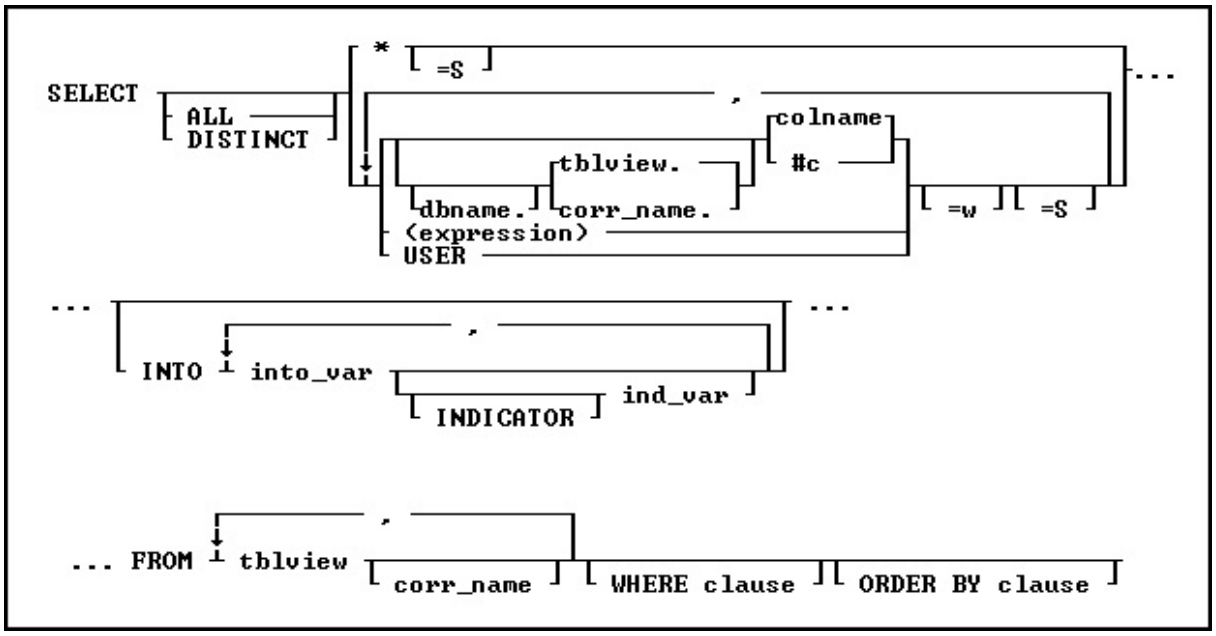
N**ote:** Many of the SELECT clauses use the same options, such as *expression* or *colname*. These common options are described only once in "SELECT Command Clause" below.

SELECT Command Clause

The required SELECT command clause specifies which columns to include. You can:

- Select all columns by entering SELECT with an asterisk.
- Name the columns you want to select.
- Use expressions and SELECT functions to perform calculations whose results will also appear as a column in the final result.
- Select the column or expression values and load them into variables.

Syntax:



Options

*****
Specifies all columns.

,
Indicates that this part of the command is repeatable.

ALL
Specifies all rows returned by the other clauses.

#c
Specifies a column, where *#c* is the column number shown in the output of the LIST TABLES command. You can enter a table or correlation name before the *#c*.

colname
Specifies a column name. In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*). You can enter *tblname.** to specify all columns in the table.

corr_name
Correlation name. A nickname or alias for a table or view name. Use *corr_name* to refer to the same table twice within the command, or to explicitly specify a column in more than one table. An example is provided below.

dbname

Currently connected database name, plus the drive and directory if the database is not on the current directory. It has the form D:\PATHNAME\DBNAME where D: is the optional drive letter, /PATHNAME is the optional directory path, and /DBNAME is the database name.

DISTINCT
Eliminates duplicate rows from the resulting data set.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

FROM
Lists the tables from which data is to be displayed.

ind_var
Specifies a variable result indicator to be used with an INTO clause in a SELECT command. This variable stores the status of the variable: non-null (0) or null (-1).

INDICATOR
Indicates the following variable is an indicator variable, which is used to indicate if a null value is retrieved.

INTO
Selects information directly from a table and puts it into variables. You must include a WHERE clause so the SELECT command finds only one row.

into_var
Specifies a variable whose value is assigned with an INTO clause in a SELECT command.

ORDER BY clause
Sorts rows of data. See "ORDER BY Clause" later in this entry.

=S
Calculates the sum of a column that has CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data type values, or the results of an expression using CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data type values.

tblview
Specifies a table or view name.

USER
Retrieves the current user as a constant.

=w
Specifies a display width.

WHERE clause
Limits rows of data. See "WHERE Clause" later in this entry.

Examples

The following command selects the company name and ID for companies in Washington state:

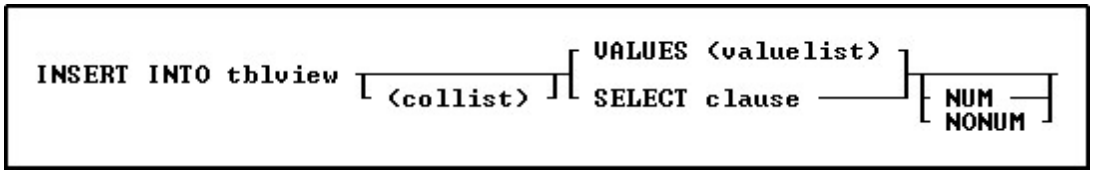
```
  SELECT custid, company FROM customer +
  WHERE custstate = 'WA' ORDER BY company

custid company
122    Data Solutions
119    Datacrafters Infosystems
130    MIS by Design
114    Softech Database Design
```

The following command selects the transaction amounts for model number by ID, and uses t1 and t2 as correlation names for the tables:

```
SELECT t1.transid, t1.netamount, t2.model +
FROM transmaster t1, transdetail t2 +
WHERE t1.transid = t2.transid
```

6.2 INSERT

Use the INSERT command to add data to a table or view without using a data-entry form.



Options

(collist)
Specifies a list of one or more column names, separated by a comma (or the current delimiter). In an SQL command, any column name in the list can be preceded by a table or correlation name and a period (*tblname.colname*).

INTO tblview
Specifies the table or view name (views must be updatable).

NUM
NONUM
NUM specifies that autonumbering columns will be numbered as they are inserted. NONUM turns off autonumbering while inserting, thereby allowing inserting of a specific value for autonumber columns. The default is NUM.

SELECT clause
Finds values in a table, tables, or view to insert into the table or view specified by the INTO *tblview* option and the columns specified by the *collist* option.

VALUES (vallist)
Specifies a list of values to insert into the table specified by the INTO *tblview* option and the columns specified by the *collist* option. Separate values with a comma or the current delimiter.

For these data types...	**Use this format for *vallist***
All data types except BIT, BITNOTE, LONG VARBIT, and VARBIT	'string' or value
BIT, BITNOTE, LONG VARBIT, LONG VARCHAR, VARBIT, and VARCHAR	['filename.ext'] or ['filename.ext', filetype, offset, length] **Note:** When you use VARCHAR, the filetype is always TXT. When you use VARBIT, BIT, and BITNOTE, filetype refers to the standard graphical file types.

About the INSERT Command

The INSERT command assigns a default value of null to any column not named in the *collist* unless a default value has been assigned to a column with the CREATE TABLE or ALTER TABLE command.

To ensure that rules are checked while adding data with the INSERT command, set RULES on before running the INSERT command.

The setting of the SET ZERO command affects the calculation of numeric computed columns. To have null values treated as zeros in expressions, set ZERO on. When ZERO is set off, if the value of a column used in a expression is null, the computed value will be null.

You cannot insert values into the table used in the SELECT clause.

To ensure that data is placed in the intended column, use the following guidelines:

- Do not embed commas within entries for CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency values; For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When a value you specify for a column is not the same data type as the column's data type, R:BASE displays an error message and you need to re-enter the entire row.
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Inserting an Autonumbered Column

When you use INSERT to add a row, INSERT assigns the next available number to autonumbered columns in the table. Therefore, omit autonumbered columns and their values from a *collist.*Also, if you use the SELECT option, omit an autonumbered column from the *collist*. If a value is included for an autonumbered column that was omitted from the column list, R:BASE does not run the command because it cannot identify which column to load.

Inserting a Computed Column

Because a computed column's value is calculated, you cannot insert a new value. Omit computed columns from a *collist* or, if you are adding data to all columns, do not use a *collist* and do not specify a value for the computed column. R:BASE will skip the computed column when the row is inserted.

Examples

In the following example, the *sales* table has three columns, *col1*, *col2*, and *col3*; and *col2* is a computed column. To insert a row, you would only specify values for *col1* and *col3*. In this example, the value for *col1* is 100, and the value for *col3*is 200.

If the expression for *col2* was (*col1 + 200*), then *col2* would have the value 300 when the row is inserted.

```
INSERT INTO sales VALUES (100, 200)
```

In the following example, a *vallist* adds a new row to the *product* table, filling the *model*, *prodname*, *proddesc*, and *listprice* columns.

```
INSERT INTO product (model, prodname, proddesc, listprice) +
   VALUES ('PB3060', 'Portable Advanced PC', 'System-Single +
   Drive w/Hard Disk-Portable', 3795)
```

The following command uses a *vallist* with global variables to insert the values from variables *v1*, *v2*, and *v3* into the *bonusrate* table.

```
SET VARIABLE v1 CURRENCY = 50000, v2 CURRENCY = 75000, +
   v3 REAL = .10
INSERT INTO bonusrate VALUES (.v1, .v2, .v3)
```

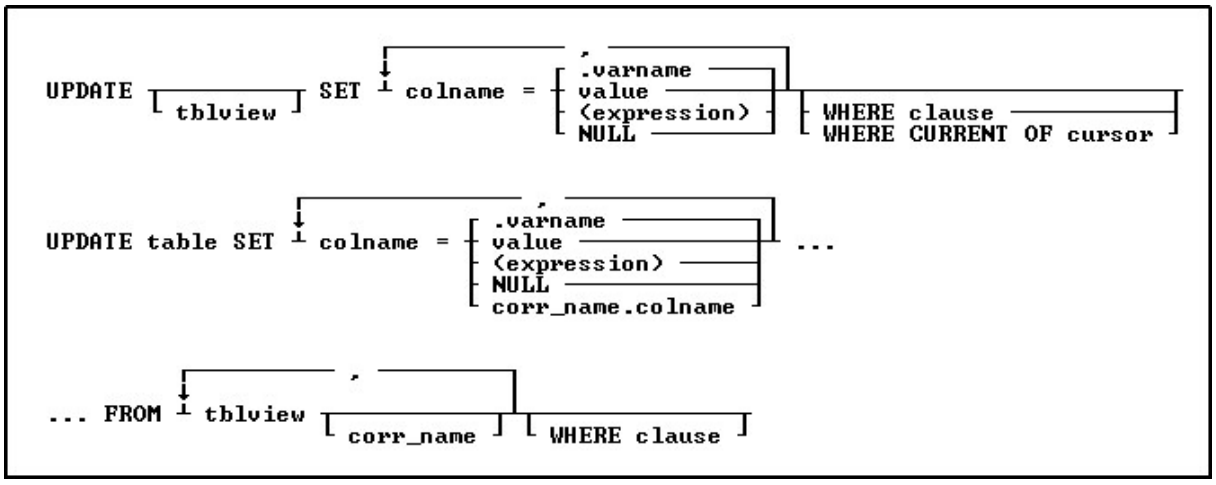
The following example adds rows to *customer* table selected from *temp* table. It adds data into the *company* and *custphone* columns. The columns taken from the *temp* table can have different column

names, the data types must be the same, and the order and number of columns in the column list of the source table (designated by SELECT) must match the column list of the destination table (designated by INTO).

```
INSERT INTO customer (company, custphone) +
   SELECT cname, phone FROM temp
```

6.3 UPDATE

Use the UPDATE command to change the data in one or more columns in a table or a view.



Options

,
Indicates that this part of the command is repeatable.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

FROM tbllist
Specifies a list of tables from which data can be retrieved and updated.

NULL
Sets the values in the column equal to null.

SET colname
Specifies the column to update.

table
Specifies a table.

tblview
Specifies a table or view. If no table or view name is included, columns will be updated in all tables containing the specified columns, according to the conditions of the WHERE clause.

value
Specifies a value to enter in the specified column.

.varname
Specifies a global variable that provides a value for a column.

WHERE clause
Limits rows of data. For more information, see the "WHERE" entry.

WHERE CURRENT OF cursor
Specifies a cursor that refers to a specific row to be affected by the UPDATE command. With this option, you must specify *tblview*.

About the UPDATE Command

The UPDATE command is useful for adjusting values in columns that require uniform changes.

The UPDATE command only modifies data in columns in one table or view. You can also update a table by referencing values from another table. You can modify a column's value by doing the following:

- Entering a new value for the column as a constant or variable
- Entering an expression that calculates a value for the column
- Entering a null value

Only users that have been granted rights to update the table(s) or column(s) can run the UPDATE command.

R:BASE complies with defined rules, even for columns not affected by the update. If an update breaks a rule, the update is not processed.

You cannot use UPDATE with computed or autonumbered columns. To change a computed column value, change the values in the columns to which the computed column refers.

Notes:

- The UPDATE command will not update data in a multi-table View (a View based on multiple tables), as the data is not editable.
- A View with a GROUP BY parameter is also not editable.

Updating Column Values

You can update a column with a specific value. The value you use must meet the requirements of the column's data type, for example, a numeric column cannot be loaded with a text value.

Use the current delimiter character (the default is a comma) to separate each column and its new value from the next column and value.

Use the following guidelines when modifying data with UPDATE:

- Do not embed commas within entries for CURRENCY, DATE, DATETIME, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency values. For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Using an Expression or Variable

Enclose expressions in parentheses. If you use global variables in an expression, dot the variable (.*varname*). If expressions contain values that have a TEXT data type, enclose the values within quotes. The default QUOTES character is the single quote (').

If you attempt to use a null value in an expression or computed column, the result of the expression is null. However, if you set ZERO to on, R:BASE treats null values as zeros and processes expressions as if the null value were zero.

Using the WHERE Clause

If an UPDATE command includes a table or view name, you do not need to specify a WHERE or WHERE CURRENT OF clause. All rows will be updated.

If you use a WHERE CURRENT OF clause, you must include a table or view name in the command.

If you omit a table or view name, you must use a WHERE clause with the UPDATE command so that you do not change values in more rows than you intended to change. The WHERE clause pinpoints the rows you want to change. If any columns exist in more than one table, all occurrences are changed if the column value meets the WHERE clause conditions. Test the WHERE clause by using the SELECT command before using the clause with UPDATE command. By using a WHERE clause with a SELECT command, you can view the rows you want to change before changing them.

R:BASE takes significantly less time to process a WHERE clause if one of the columns specified in the clause is an indexed column.

Using UPDATE with Transaction Processing

If more than one person at a time executes an UPDATE command and transaction processing is on, R:BASE might not execute the command concurrently. If you hold an UPDATE lock, you can read, modify, or delete any row in a table. R:BASE blocks any additional requests for UPDATE until other SELECT or UPDATE locks are cleared.

Examples

The following command changes values in the *company* and *custphone* columns of the *customer* table for the row where *custid* equals 100.

```
UPDATE customer SET company = 'Quality Computers', +
custphone = '617-341-3762' WHERE custid = 100
```

The following command changes the *invoicetotal* column in the *transmaster* table to the value of the expression (*invoicetotal * .9*) for rows where *transid* is greater than 5000.

```
UPDATE transmaster SET invoicetotal = ( invoicetotal * .9) +
WHERE transid > 5000
```

The following command changes the *listprice* column to the value of the expression (*1.1 * listprice*) for every row in the *prodlocation* table containing an entry in the *listprice* column.

```
UPDATE prodlocation SET listprice = (1.1 * listprice) +
WHERE listprice IS NOT NULL
```

The following command adds to the set of conditions in the above command. The command below extracts all of the selling prices from the *transdetail* table and requires that *listprice* be changed only if it matches a current selling price in the table.

```
UPDATE product SET listprice = (1.1 * listprice) +
WHERE listprice IS NOT NULL AND model = 'CX3000' +
AND listprice IN (SELECT price FROM transdetail +
WHERE model = 'CX3000')
```

The following command changes the *onhand* column in the *prodlocation* table (specified by cursor *curs1*) to the value of the expression (*onhand - 100*). The changes are made only in the row currently referenced by the cursor.

```
UPDATE prodlocation SET onhand = (onhand - 100) +
WHERE CURRENT OF curs1
```

The following example shows interactive data updating in an application file. The value of *var1* is used in the expression that is assigned to the *onhand*column of the *prodlocation* table. The UPDATE command changes values in *onhand* to the value of the expression (*onhand - .var1*) for all rows containing model numbers that begin with the letter C. The wildcard character % indicates one or more additional characters.

```
SET VARIABLE var1 TEXT
DIALOG 'Enter quantity by which to reduce inventory: '  var1 vend 1
SET VARIABLE var1 INTEGER
UPDATE prodlocation SET onhand = (onhand - .var1) +
WHERE model LIKE 'C%'
```

The following command changes the last names of two employees. This command omits the table name, thereby causing a global change to all tables that meet the WHERE clause criteria.

```
UPDATE SET emplname = 'Smith-Simpson' WHERE +
(empfname = 'Mary' AND emplname = `Simpson') OR +
(empfname = 'John' AND emplname = 'Smith')
```

The following example corrects a problem that can occur with an incorrect date sequence setting. For example, assume that you had the date sequence set to a four-digit year when you entered transactions, and you entered dates with a two-digit year (3/1/93). The dates would be stored as 3/1/0093. And, if you wanted the date to be in the 20th century, you could use the UPDATE command to modify the existing dates to 20th century dates by adding 1900 years to each date, with the ADDYR function.

The SET DATE command makes sure that you are using a four-digit year. The UPDATE command changes all *transdate* values to 20th century dates, where the current value of the column is less than 1/1/1900. The last SET DATE command returns to a two-digit date sequence and format.

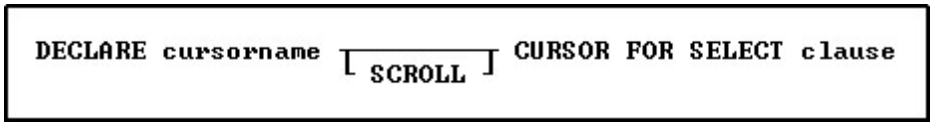
```
SET DATE MM/DD/YYYY
UPDATE transmaster SET transdate = (ADDYR(transdate,1900)) +
WHERE transdate < 1/1/1900
SET DATE MM/DD/YY
```

Assume that you wanted to update the *inventory* table with the sum of the units sold from the *orders* table. Because there are many rows in the *orders* table for each part number, you cannot do this directly with the UPDATE command. The CREATE VIEW command creates a view containing the sum of the units sold from the *orders* table. The UPDATE command updates the *inventory* table by extracting the *totalsold* value from the view named *orders_view* for each part number.

```
CREATE VIEW orders_view (partid,totalsold) AS SELECT +
partid, sum(sold) FROM orders GROUP BY partid

UPDATE inventory SET onhand = (T1.onhand - T2.totalsold) +
FROM inventory T1, orders_view T2 +
WHERE T1.partid = T2.partid
```

6.4 DECLARE CURSOR

Use the DECLARE CURSOR command to create a cursor that points to a row in a table or view.



Options

cursorname
Specifies a 1 to 18 character cursor name.

CURSOR FOR SELECT clause
Specifies the columns and rows from the table whose values you want to use. You may include the DISTINCT modifier as well as WHERE clauses and ORDER BY clauses.

SCROLL
Defines a cursor that moves forwards and backwards through a table. If this option is omitted, the cursor can only move forward.

About the DECLARE CURSOR Command

In the SELECT clause, specify the columns that contain the values you want to use from the row. Specifying the columns makes the column values accessible to the FETCH and SET VARIABLE commands. Once a cursor is declared, use the OPEN command to initialize the cursor and position it before the first row specified by DECLARE CURSOR.

Use DECLARE CURSOR to define a path through a table or view. You can move through the defined rows using the FETCH command by using either multiple FETCH commands or embedded FETCH commands within a WHILE loop. You only need to point to specific columns with DECLARE CURSOR, then FETCH can retrieve those columns by placing their values into variables. You can define a scrollable cursor, which is a cursor that moves backwards and forwards through a table.

DECLARE CURSOR defines a temporary view in memory; R:BASE does not store the view definition in the *sys_views* table. The SELECT clause defines columns, tables, rows, sort order, and potential grouping for the rows. When DECLARE CURSOR executes, it validates the syntax and names of columns and tables. The OPEN command can evaluate variables, create a copy of the cursor based on those values, then position the cursor before the first row.

Listing Cursors

Use LIST CURSOR to list all currently defined cursors and their status, open or closed.

Using Cursor Names in Commands

You can use the cursor name instead of a table name in commands. The following table provides examples of using the cursor name instead of a table name in commands.

To do this...	Use the cursor name like this...
Set a variable to a column value	FETCH *cursorname* INTO *varlist* SET VARIABLE *varname* = *colname* WHERE CURRENT OF *cursorname*
Change a column value to a constant	UPDATE *tblname* SET *colname* + = *value* WHERE CURRENT OF *cursorname*
Change a column value to a variable value	UPDATE *tblname* SET *colname* + = .*varname* WHERE CURRENT OF *cursorname*
Change a column value to an expression	UPDATE *tblname* SET *colname* + = (expression) WHERE CURRENT OF *cursorname*
Delete the pointed-to row	DELETE FROM *tblname* + WHERE CURRENT OF *cursorname*

Modifying Data Using a Cursor

If you use a cursor in commands that modify data (the UPDATE and DELETE commands), only the current row is modified. To modify all referenced rows, include FETCH in a WHILE loop to move the cursor through the rows.

Checking for End-of-Data Conditions

End-of-data conditions determine whether you have reached the end of the data declared with the DECLARE CURSOR command. The three ways to check for end-of-data conditions are:

- Use an error variable defined with the SET ERROR VARIABLE command
- Use the *sqlcode* variable
- Include a WHENEVER NOT FOUND command

Closing Cursors

The following commands close cursors.

Command Name	Description
CLOSE	Closes the open cursor but does not remove the cursor definition. However using CLOSE frees most of the memory used when a cursor is opened. CLOSE also frees any file handles used by DECLARE CURSOR.
COMMIT	Closes any open cursors
CONNECT	Removes any cursor definitions from memory
DISCONNECT	Removes any cursor definitions from memory
DROP CURSOR	Entirely removes the cursor definition. Dropping a cursor definition frees all memory used by the definition.
ROLLBACK	Closes any open cursors

Examples

The following example uses the SCROLL option with DECLARE CURSOR.

```
DECLARE c1 SCROLL CURSOR FOR SELECT empid, transid, transdate, custid, netamount FROM
transmaster
```

Checking End-of-Data Conditions Using sqlcode

The following example uses *sqlcode* to check end-of-data conditions, which is the recommended program structure for DECLARE CURSOR. The *sqlcode system variable* holds values only for specific types of status.

Type of Error	SQLCODE
Data found	0
Data not found	100

In the following example, the WHILE statement checks the value of *sqlcode*.

```
1) DECLARE cursor1 CURSOR FOR SELECT custid, netamount +
     FROM transmaster ORDER BY netamount
2) OPEN cursor1
3) FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
4) WHILE sqlcode <>100 THEN
     SHOW VARIABLE vcustid
     SHOW VARIABLE vnetamt
5)   FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
       INDICATOR vi2
   ENDWHILE
6) DROP CURSOR cursor1
```

1. DECLARE CURSOR defines the cursor path.

2. OPEN opens the cursor, evaluates variables, and positions the cursor before the first row.
3. The first FETCH command retrieves the first set of values. The indicator variables *vi1* and *vi2* capture the status values, -1 for null and 0 for a value. If you omit indicator variables in FETCH commands, R:BASE displays a message if it encounters a null value, but continues processing rows.
4. The WHILE loop processes the rows until there are no more rows. At that point, *sqlcode* is set to 100, and the WHILE loop ends. Control passes to the command after ENDWHILE. If the first FETCH retrieved no data, the WHILE loop is not entered.
5. FETCH retrieves all succeeding rows and sets *sqlcode*each time. When it does not find any more data, *sqlcode* is set to 100 and the WHILE loop ends.
6. DROP CURSOR removes the cursor definition from memory.

Using the WHENEVER Command with DECLARE CURSOR

The following example shows the use of the WHENEVER command, which checks the value of *sqlcode*. A single WHENEVER command can start a status-checking cycle that remains in operation until a command or procedure file finishes running. As in the first two examples, an indicator variable is included with each variable in FETCH. Without the indicator variables, R:BASE displays a message if it encounters a null value, but continues processing rows.

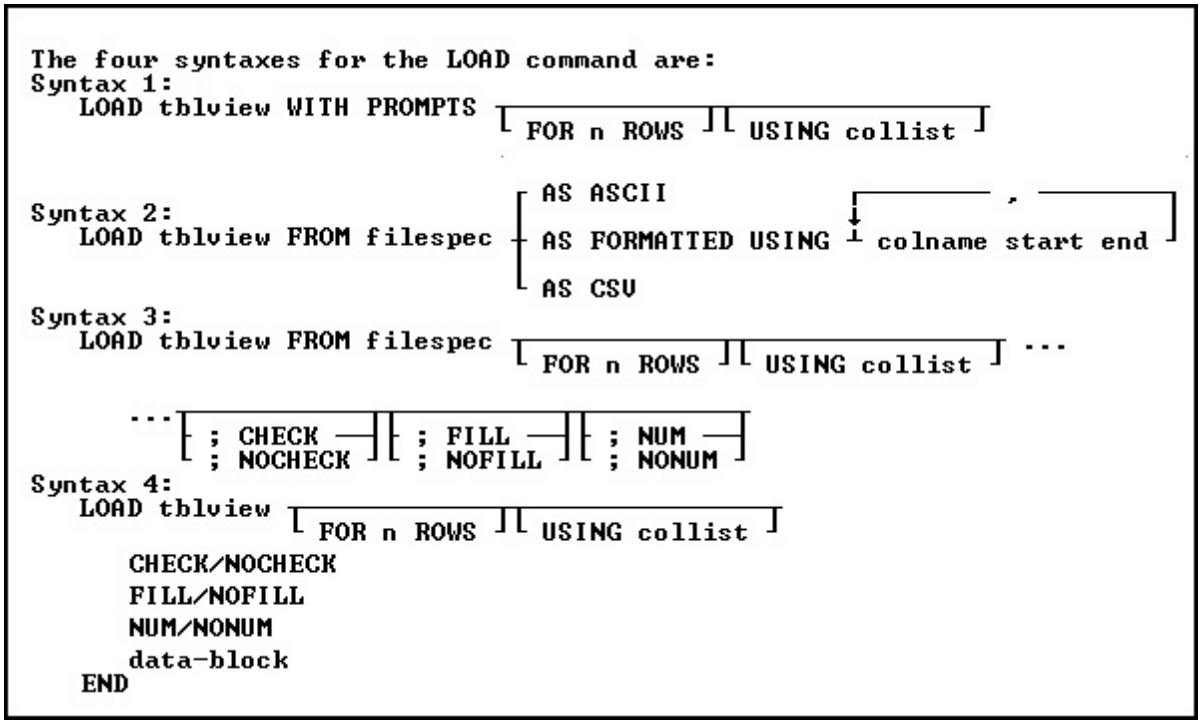
```
1) WHENEVER NOT FOUND GOTO skiploop
2) DECLARE cursor1 CURSOR FOR SELECT custid, netamount +
     FROM transmaster ORDER BY netamount
3) OPEN cursor1
4) FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
5) WHILE #DATE IS NOT NULL THEN
     SHOW VARIABLE vcustid
     SHOW VARIABLE vnetamt
     FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
   ENDWHILE
6) LABEL skiploop
7) DROP CURSOR cursor1
```

1. WHENEVER NOT FOUND tells R:BASE to execute GOTO if a command that searches for data, such as FETCH, cannot find more rows. If the first FETCH command does not find any rows, control passes to the command following LABEL *skiploop*. WHENEVER automatically checks any command that searches for data. If a data-not-found condition occurs, control passes to the command following the specified label.
2. DECLARE CURSOR defines the cursor path.
3. OPEN opens the cursor, evaluates the variables, and positions the cursor before the first row.
4. The first FETCH command retrieves the first set of values. If no rows match, control passes to LABEL *skiploop*. Indicator variables *vi1* and *vi2* capture the status values (-1 for null and 0 for a value). If you omit indicator variables in FETCH commands, R:BASE displays a message if it encounters a null value, but continues processing rows. (WHENEVER instructs R:BASE to exit the WHILE loop only when *sqlcode* is 100.)
5. The WHILE loop processes rows until WHENEVER stops execution.
6. This label defines where to pass control if a data-not-found condition occurs before the WHILE loop begins executing. WHENEVER includes this label name.
7. DROP CURSOR removes the cursor definition from memory.

Visit the From The Edge Web site to download the "R:BASE Cursors Explained" technical document.

6.5 LOAD

Use the LOAD command to add data to a table or to a single table view that can be updated.



Options

,
Indicates that this part of the command is repeatable.

AS ASCII
The LOAD AS ASCII command is designed strictly for speed of operation. LOAD AS ASCII checks rules and constraints. However, following is a list of the limitations of the LOAD AS ASCII command:

- It does not check the data types; therefore, invalid data will be loaded as null values into columns; no error messages about this conversion are displayed.
- It does not display error messages when columns must be truncated, or when excess data exists on any line.
- It does not echo data to the screen regardless of the setting for SET ECHO.

To achieve maximum speed of loading, the data must look like the data that R:BASE unloads with the UNLOAD DATA AS ASCII command. That is, the data must conform to the following:

- The carriage return/line feed characters define the end of the line for a given row; the maximum row size is 4096 characters.
- The data cannot include variables.
- The data cannot include comments.

AS FORMATTED USING
Loads data from an ASCII file when the data is formatted in fixed column locations, with the following restrictions:

- The carriage return/line feed characters define the end of the line for a given row; the maximum row size is 4096 characters.
- The data cannot include variables.
- The data cannot include comments.

- You must specify the name of each column of the table to be loaded, and the starting and ending position of its data in the line, which is specified in the USING clause of this command.

CHECK
NOCHECK
CHECK turns on rule checking. When rule checking is on, R:BASE checks input against data validation rules. NOCHECK turns off rule checking. CHECK and NOCHECK override the current setting of the SET RULES command. The default is CHECK.

colname start end
Specifies the name of a column in the table and the starting and ending position of its data in the line; this option is used with the AS FORMATTED option.

data-block
Includes lines of data to be loaded, as well as the LOAD subcommands.

For these data types...	Use this format for data-block
All data types except BIT, BITNOTE, LONG VARBIT, and VARBIT	'string' or value
BIT, BITNOTE, LONG VARBIT, LONG VARCHAR, VARBIT, and VARCHAR	['filename.ext'] or ['filename.ext', filetype, offset, length] **Note:** When you use VARCHAR, the filetype is always TXT. When you use VARBIT, BIT, and BITNOTE, filetype refers to the standard graphical file types.

FILL
NOFILL
FILL makes null any columns that have not been assigned values. All of the missing values must be at the end of the row. If a rule specifies that a column requires an entry other than null, do not use FILL. NOFILL turns off FILL and requires a value for each column. The default is NOFILL.

FOR n ROWS
Directs R:BASE to stop processing after loading *n* rows, where *n* is a positive whole number. In the fourth syntax diagram, END is not used if FOR *n* ROWS is included.

FROM filespec
Loads data into the specified table with data from an external ASCII delimited file.

NUM
NONUM
NUM specifies that autonumbering columns will be numbered as they are loaded. NONUM turns off autonumbering while loading, thereby allowing loading of a specific value for autonumber columns. The default is NUM.

tblview
Specifies a table or view name to load.

USING collist
Specifies the column(s) to use with the command.

WITH PROMPTS
Loads data into the specified table from keyboard entries. R:BASE asks for the values of each column by displaying the column name and its data type. To end the loading session, press [Esc].

About the LOAD Command

You cannot load data into a multi-table view.

Instead of using LOAD, you can also use INSERT, the Data Editor, or a form to add data to a table.

You can use the LOAD command to load data into R:BASE from a file that was not created by R:BASE. The file must be an ASCII file, either delimited or fixed.

The LOAD command will differentiate between END and 'END'; FILL and 'FILL'; NOFILL and 'NOFILL'; CHECK and 'CHECK'; NOCHECK and 'NOCHECK'; NUM and 'NUM'; NONUM and 'NONUM'. So, make sure to use the proper syntax when creating LOAD statements.

To ensure that data is placed in the intended column, use the following guidelines:

- Do not embed commas within entries for CURRENCY, DATE, DATETIME, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency values. For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When a value you specify for a column is not the same data type as the column's data type, R:BASE displays an error message and you need to re-enter the entire row.
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Loading with a USING Clause
A USING clause is helpful when you do not have all the information that is to be added to a table. The following example lets you enter some information for a product but does not require that all columns be entered. The *model* and *listprice* columns are the first and last columns in the *product* table. The *prodname* and *proddesc* columns are not included in the command and are loaded with null values. You can later edit the *product* table to enter data into the columns that have null values.

```
LOAD product USING model listprice
```

Loading with the CHECK Option
The SET RULES command does not have any effect on the CHECK option because CHECK has precedence over a RULES setting. When RULES is set off, the CHECK option still verifies data entry against existing rules.

When a user identifier has been assigned to the database owner, you must enter the owner's user identifier with the CONNECT or SET USER command before you use the CHECK or NOCHECK option. R:BASE does not accept the CHECK or NOCHECK option unless the owner's user identifier has been entered.

Loading Computed Columns
You cannot load data directly into a computed column. After you load the column values that are used to calculate the computed column, R:BASE fills the computed column with the computed value.

The setting of the SET ZERO command affects the calculation of numeric computed columns. To have null values treated as zeros in expressions, set ZERO on. When ZERO is set off, if the value of a column used in a expression is null, the computed value will be null.

Loading Negative CURRENCY values
When loading negative CURRENCY values into a table, the format must include the hyphen, i.e -$500.00. Negative CURRENCY values encased in parenthesis are not recognized, e.g. ($500.00).

Loading with Prompts
When you run the LOAD command using prompts, you load one row of data at a time into the table you specified. (See Syntax 1). For each new row you add, R:BASE displays the name and data type of the row's column as prompts. At each prompt, you enter the value that you want the column to contain. You are prompted for each column in the row beginning with the first column, unless you used a USING *collist* clause to limit the number of the columns to load, or to change the order in which the columns are loaded. Any columns not listed in the *collist* are given null values when the rows are entered.

When you load data with prompts, the default length for a text entry is 80 characters. To enter columns with a NOTE or TEXT data types that contain more than 80 characters, load the data without prompts, make a custom data-entry form, or set the WIDTH so you can enter more characters.

R:BASE does not prompt you for computed or autonumbered column values.

Loading without Prompts
Loading without prompts (See Syntax 4) is faster but requires that you remember the order of the columns in the table. When you load without using prompts and not from an ASCII file, the LOAD command provides its own distinctive prompt, which is L>. The following options can be entered at this prompt: CHECK, NOCHECK, FILL, NOFILL, NUM, and NONUM.

Loading from an ASCII File
Use the LOAD command from the R> Prompt or a command file to load data into an existing table from both delimited and fixed field ASCII files. (See Syntax 2 and 3 diagrams.) Each record in the ASCII file corresponds to one row of data in a table, and each item of data in a record corresponds to one column value in a row. Therefore, organize data in the file in the same order as the columns in the table to be loaded.

Items of data in a line of the ASCII file must be delimited to be properly placed within the columns of a row. The delimiter character must be the same as the current delimiter character specified with the SET DELIMIT command. (The default delimiter is a comma.) R:BASE also accepts a blank space as a delimiter, regardless of the setting of the SET DELIMIT command.

Data can be loaded in a fixed-field formatted ASCII file with the AS FORMATTED option. The column name and the start and end positions within the file must be specified for each value in the row of data that is to be loaded. When the start and end positions are specified, the delimiter character does not have any effect because the start and end positions for each column identify the data.

When loading from a file, be sure that the current null symbol is not a blank. If the first four characters of a field in a file are blank, R:BASE adds the column as a null column and does not read any additional characters that make up the field value.

When loading data from an ASCII file, make sure the file meets the following requirements listed in the table below.

Elements in an ASCII File	**Requirement**
INTEGER data types	Items of data to be loaded into columns with INTEGER data types cannot contain internal commas unless the item is enclosed in quotes. The default QUOTES character in R:BASE is a single quote ('); if your ASCII file uses double quotes ("), change the QUOTES setting before you load the file. If the file does not have quotes around the integer values containing commas, you must edit the ASCII file to remove any commas from the integer values, or enclose each integer value in quotes.
Embedded punctuation	Items of data containing ampersands, commas, embedded blanks, plus signs, equal signs, or semicolons must be enclosed in quotes if they are to be loaded into columns with a TEXT or NOTE data type. The default QUOTES character in R:BASE is a single quote ('); if your ASCII file uses double quotes (''), change the QUOTES setting before you load the file.
Embedded quotes	Items of data requiring quotes can also contain embedded quotes. For example, the item '*Basic*' *Keyboard* contains both a blank space and embedded quotes. Using single quotes ('), which is the default QUOTES setting, to add enclosing quotes, the item would looks like this: '''Basic'' Keyboard'
Currency	R:BASE automatically adds a currency symbol, commas, and zeros for currency units. For example, using the default currency format, R:BASE loads an entry of *1000* as *$1,000.00.*
Dates	The SET DATE sequence command sets the sequence for the date-dates in the file are loaded if the dates match the current date sequence established with the SET DATE command.
Computed columns	If the table being loaded has computed columns and the file contains values for the computed columns, R:BASE tries to load the computed column's value from the file into the column following the computed column. This results in an error because the data type of the next column might not be the correct data type, or the file will have too

	many values for the table because R:BASE does not load the computed column's value from the file.
Rules processing	Unless you run the SET RULES OFF command before loading the file, rules processing is in effect. When an incoming data item violates a rule, R:BASE does not load the row. Instead, R:BASE displays the message for the rule that has been violated. To see the data that causes a rule violation, set ECHO on when loading a table and use the [Pause] key to stop the screen from scrolling when the rule violation occurs.

Loading a Data Block
The data block shown in the Syntax 4 diagram can include lines of data and any of the options available with LOAD-CHECK/NOCHECK, FILL/NOFILL, and NUM/NONUM. You can intersperse the options with data lines, and you can enter more than one option on a line if you separate the options with semicolons. However, you cannot combine data and options on the same line.

R:BASE displays the L> prompt to accept data-block entry. LOAD adds data to a table, row by row, without using a data-entry form and without prompting for each data item.

You can enter the options for the LOAD command at the L> prompt at any time during data loading. Or you can include them on the command line, separated from the command by semicolons, as shown in the example below. (Do not use this format in command or procedure files. All options must follow the LOAD command on separate lines in command or procedure files.)

```
LOAD transdetail ; CHECK ; NUM
```

You can use global or system variables instead of constant values in the data block.

To enter values properly, use the following guidelines.

- Enter column values in the order that columns are defined in the table, and separate the values with a delimiter character. The default delimiter character is the comma.
- You can enter up to 75 characters on a single line. If a row is longer than 75 characters, continue on to the next line by typing past the end of the current line or by entering a plus (+) sign at any point on the current line. The plus sign must be the last entry on the line. The new line will begin with a +> prompt to indicate the continuation of the current line. If you are using this form of the LOAD command in a command file, you must use a + to continue on the next line; the lines will not automatically wrap.
- For other requirements on loading data, see "Loading from an ASCII File" earlier in this entry.

Examples

The following command lines allow you to load rows containing new customer information to the *customer* table. R:BASE asks with prompts for each column by column name and data type. Two columns in the table, *custid* and *custphone*, are omitted from the list. R:BASE automatically supplies a number for the *custid* column because it is an autonumbered column. R:BASE leaves the *custphone* column empty (null) when data is loaded, and does not prompt for either column.

```
LOAD customer WITH PROMPTS USING company, +
custaddress, custcity, coststate, custzip
```

After the above command is run, the WITH PROMPTS option displays the message below. If you press [Esc] before you have finished entering data in a row, the row is not added to the table. When you are prompted to exit, press [Esc].

```
Press Esc to end, Enter to continue
company (TEXT):
```

The following command loads five rows of data into the *customer* table from CUST.DAT, a delimited ASCII file. The data in the ASCII file must be in the same order as the columns in the *customer* table. Only the first five lines from the file will be loaded:

```
LOAD customer FROM cust.dat FOR 5 ROWS
```

In the following example, the first command line starts the loading process. When loading begins, an L> prompt is displayed for each new line. The second command line, which contains the FILL option, tells R:BASE to give any additional columns at the end of the row null values. The third command line loads the values from three global variables into a new row. The values fill the first three columns of the new row, while the remaining columns (if any) are given null values. The last command line tells R:BASE to end the loading session:

```
R>LOAD bonusrate
L>FILL
L>.v1, .v2, .v3
L>END
```

In the following example, the first command line tells R:BASE to start loading data for the *customer* table. The second command line tells R:BASE to number any defined autonumbered columns in the table. A value for the autonumbered column is not designated because the row might not be loaded if a value is included.

```
R>LOAD customer
L>NUM
```

After the command line in the following example is run, R:BASE expects the next five lines entered at the L> prompt to contain data to be loaded into the *customer* table. After the fifth line of data is entered, the R> Prompt returns and loading ends. To end loading before five rows of data are entered, enter END.

```
R>LOAD customer FOR 5 ROWS
```

The following command lines show you how to load data into the *company*, *custaddress, custcity, coststate*, and *custzip* columns of the *customer* table. The *custid* and *custphone* columns in the *customer* table will not have data loaded and will be given null values.

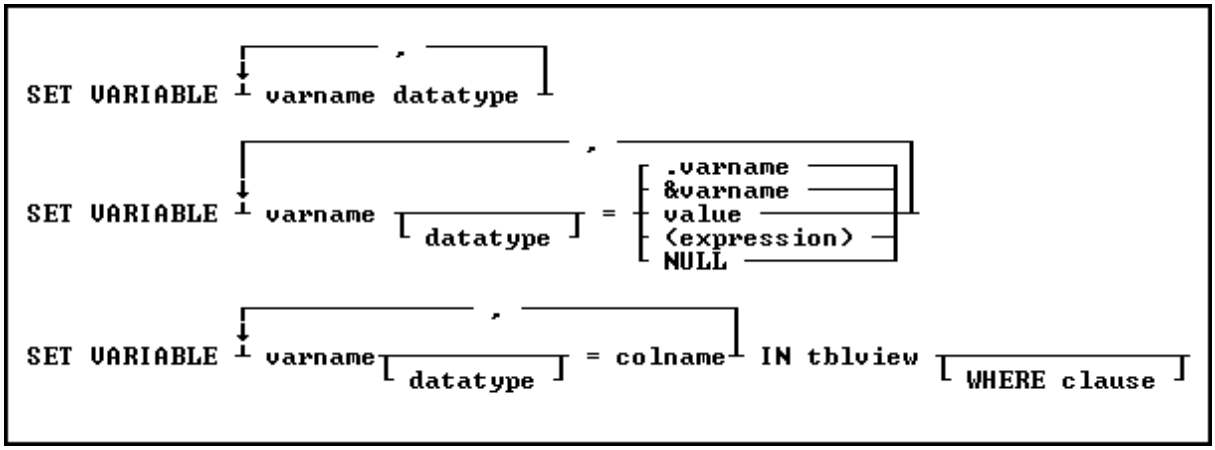
```
LOAD customer FROM customer.fix AS FORMATTED +
USING company 11 50, custaddress 51 80, +
custcity 81 100, coststate 101 102, custzip 103 112
```

When you use the LOAD command, you must omit values for computed or autonumbered columns. Instead, enter the value for the next column in the data list. In the following example, to add a row to the *transdetail* table, which has a computed column, you would only enter data for the first five columns; the sixth column is a computed column based on the fourth and fifth columns. The columns entered are *transid*, *detailnum*, *model*, *units*, and *price*. The computed column is *extprice* and has the expression (*units* * *price*).

```
R>LOAD transdetail
L>6000,1,'CX3000',100,$1900
L>END
```

6.6 SET VARIABLE

Use the SET VARIABLE command to define or redefine a variable value and/or data type.



Options

,
Indicates that this part of the command is repeatable.

colname
Specifies a column name. In R:BASE X.5, the column name is limited to 18 characters. In R:BASE X.5 Enterprise, the column name is limited to 128 characters. In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

datatype
Specifies an R:BASE data type for the variable.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

IN tblview
Specifies a table or view.

NULL
Sets the variable equal to NULL.

value
Sets the variable equal to a specified value. A value is a constant amount, text string, date, or time, or the value assigned to *varname*.

varname
Specifies a 1 to 18 character variable name.

&varname
Sets the first variable equal to the exact contents of a second variable; the ampersand tells R:BASE to evaluate the contents of the variable first.

For example, if *varname* is the text string (2+3), then &*varname* is the value 5.

.varname
Sets the first variable equal to the exact contents of a second variable.

For example, if *varname* is the text string (2+3), then .*varname* is (2+3).

WHERE clause
Limits rows of data. For more information, see the "WHERE" entry.

About the SET VARIABLE Command

Variables identify a changeable value. R:BASE provides three kinds of variables: global, error, and system. The SET VARIABLE command defines global variables, which are temporary variables that exist within R:BASE, but are not part of any database. Global variables remain in memory until you clear them or exit from R:BASE. R:BASE sets error and system variables internally.

Global variables have several uses: they can provide a temporary value in a command, hold the result of a calculation, act as a counter, or capture keyboard entries for use with menus or screens. The most common method of defining variables is to assign the variable value with the SET VARIABLE command. For information about defining global variables, see the "Variables" entry.

Variable names have the following restrictions:

- The variable name is limited to 18 characters in R:BASE X.5 and 128 characters in X.5 Enterprise.
- The variable name is not an R:BASE reserved word.
- The variable name begins with a letter, contains only letters, numbers, and the following special characters: #, $, _ , and %.

It is good programming practice to always define the data type for the variable before assigning it a value, unless you are setting a variable to a column value or using the variable in the CHOOSE command.

When defining an variable as a text string, enclose the text string in single quote marks (or the current QUOTES character); otherwise, it might be interpreted as an arithmetic expression.

Assigning a Data Type to a Variable

The *datatype* option refers to one of the valid R:BASE data types: BIT, BITNOTE, CURRENCY, DATE, DATETIME, DOUBLE, INTEGER, LONG VARBIT, LONG VARCHAR, NOTE, NUMERIC, REAL, TIME, VARBIT, OR VARCHAR. You can define a variable to have a NOTE data type, but R:BASE treats it as TEXT for most uses. You can also specify the precision and scale for NUMERIC data types.

The *datatype* option creates a variable, determines its data type, and sets its value to null. Use this option to define a variable's data type before assigning a value to the variable. To set multiple variables in the same command, separate the variables by a comma or the current delimiter.

For an existing variable, you can use the *datatype* option to change the data type, but it is recommended to use one of the conversion functions. If you change the data type, the new data type must be compatible with the current variable value; if the variable is not compatible, R:BASE displays an error message and leaves the value and data type unchanged. If you change a variable with a TEXT data type to a non-compatible data type, R:BASE changes the value to null.

Assigning a Value to a Variable

The *value* option is a data value or constant, such as 10, TOM, 3.1416, or $17.23. If the variable already exists, any new value must have a data type that is compatible with that variable. If the variable does not exist, R:BASE defines the variable's data type.

You can also define the variable's data type in this command before assigning it a value.

Setting the Value of a Variable to Another Variable

When you set the variable to the value of another variable, the second variable must be a dot variable (.) or an ampersand (&) variable.

When you precede a variable with a dot (.), R:BASE uses the value stored in the variable as if it were a constant.

When you precede a variable with an ampersand (&), R:BASE first evaluates the value contained in the ampersand variable. For example, consider the following uses of the command:

```
SET VARIABLE v1 TEXT = '(A + B)'
SET VARIABLE v2 = .v1
SET VARIABLE v3 = &v1
```

When the first command line runs, variable *v1* will contain (A + B). When the second command line runs, variable *v2* will also contain (A + B) because the dot tells R:BASE to set the value as an exact match to the contents of variable *v1*. When the third command line runs, variable *v3* will contain AB (the concatenation of A and B) because the ampersand tells R:BASE to compute the contents of variable *v1*.

As shown in the example above, an ampersand variable can contain one command or part of one command, such as an expression. The first variable is set to the computed value of the ampersand variable. Below is an example:

```
1.  SET VARIABLE v1 TEXT
2.  SET VARIABLE v2 INTEGER
3.  SET VARIABLE v1 = '((50 + 100) / 2)'
4.  SET VARIABLE v2 = &v1
5.  SHOW VARIABLE
```

- Sets the data types for variables *v1* and *v2* to TEXT and INTEGER, respectively.
- Sets variable *v1* to a text value that is a valid arithmetic expression.
- Sets variable *v2* to &*v1*.
- Displays the value of all variables, including the system variables.

R:BASE computes the expression contained in *v1* and assigns the calculated value to *v2*. When R:BASE sees a variable name preceded by ampersand, it treats the contents of the variable as if it was entered from the keyboard. The SHOW VARIABLE display would like the following:

Variable	= Value	Type
#date	= 04/12/94	DATE
#time	= 22:52:52	TIME
#pi	= 3.14159265358979	DOUBLE
sqlcode	= 0	INTEGER
v1	= ((50 + 100) / 2)	TEXT
v2	= 75	INTEGER

Setting a Variable to an Expression

An (*expression*) can be either an arithmetic operation that combines two or more items in an arithmetic computation, or a string expression that concatenates two or more text items, or uses a TEXT function. The items can be values or the values contained in variables.

If you do not predefine the data type of a variable, the original data type of each item determines the data type of the result. For example, if you add a variable that has an INTEGER data type to a variable that has a REAL data type, the resulting variable has a REAL data type unless you define the result to be an INTEGER data type.

If any item in an arithmetic expression is null, the result will be null unless you specify SET ZERO ON.

Assigning Column Values in a Table or View

If you specify a table or view in a SET VARIABLE command, you can include an optional WHERE clause to indicate which row to use. If you do not include the WHERE clause, R:BASE uses the value for the column in the first row.

You must have SELECT privileges on the table to use this form of SET VARIABLE.

You must use an ampersand variable in place of a column or table name, for example:

```
CHOOSE vtab FROM #TABLES
```

```
CHOOSE vcol FROM #COLUMNS IN &vtab
SET VARIABLE vnewpr = &vcol IN &vtab
```

Enter the table and column names into the *vtab* and *vcol* variables first. You can do this by using the CHOOSE #TABLES and CHOOSE #COLUMNS commands, as shown in the above example. The CHOOSE command displays a menu of tables or columns from which to choose. By using ampersand variables to hold the table and column names, you can use the same SET VARIABLE command to get values from different columns in a table or from different tables. Each time SET VARIABLE requests a column, it retrieves information from the first row in the table stored in *&vtab*.

You can define multiple variables with a single SET VARIABLE command when you set the value of the variables to the value of columns in a table. However, when capturing column data into variables, it is better to use the SELECT command; specifically, SELECT INTO. SELECT INTO is the SQL compliant command when capturing table data into variables.

Examples

The following table provides examples of the SET VARIABLE command.

SET VARIABLE Examples

Examples	Description
SET VARIABLE *vtext* TEXT	Defines the *vtext* variable to have a TEXT data type.
SET VARIABLE *vreal* REAL = 100.9	Defines *vreal* variable to have a REAL data type, and assigns it the value 100.9.
SET VARIABLE *vnumer* NUMERIC (9,3)	Defines the *vnumer* variable to have a NUMERIC data type having a precision of 9 and scale of 3.
SET VARIABLE *vnum* = 14322	Assigns the integer value 14322 to the *vnum* variable.
SET VARIABLE *VTWO* =.VNUM	Assigns the value of the *vnum* variable to the *vtwo* variable.
SET VARIABLE *V3* = &*V4*	Assigns the computed value of *v4* to the *v3* variable.
SET VARIABLE *vltdate* = ('12/25/93' + 90)	Assigns the value 03/25/94 to the *vltdate* variable.
SET VARIABLE *vfulln* = + (.*VFIRSTN* & .*VLASTN*)	Assigns to the *vfulln* variable the value of the full name formed by concatenating the values in the *vfirstn* and *vlastn* variables The ampersand inserts a space between the two values.
SET VARIABLE *v1* = *col1*, *v2* = *col2*, *v3*= *col3* IN *tbl1* WHERE *col1* = 'Smith' **OR SQL compliant variation:** SELECT *col1, col2, col3* INTO *v1* INDI *iv1, v2* INDI *iv2, v3* INDI *iv3* FROM *tbl1* WHERE *col1* = 'Smith'	Assigns Smith to the variable *v1*; the value of column *col2* to *v2*; and the value of column *col3* in *tbl1*, from the row where *col1* contains Smith, to variable *v3*.

Part VII

7 Executing External Programs

The LAUNCH command allows you to execute (or LAUNCH), from within R:BASE or an R:BASE program, any associated files along with required programs.

LAUNCH allows you to add the features of just about any other program to R:BASE. For example, you can access a word processor without exiting from R:BASE. This is the syntax for the LAUNCH command:

```
LAUNCH filename.ext|<parameters>|W
```

The LAUNCH command follows the corresponding file association registered within the Windows operating system. This means that the file name specified can be set to "any" file that has a corresponding association. For example, launching the file **MyText.TXT** will display the contents of the file with Windows NotePad, Wordpad, or some other text editor, based upon the file associated with the .TXT file extension. If a full path is not specified, the current search path is used. The LAUNCHed application will start in the same directory as the executable file or in the directory referenced, if specified.

The LAUNCH command also allows you to specify additional "parameters" based upon the file you are launching. When listing program parameters, the selected program will start when the LAUNCH command is used.

A "Wait" parameter **|W** is also availble within the LAUNCH commnad, which allows you to specify whether or not you want R:BASE to sleep while the launched program executes. This basically, tells R:BASE to "wait until finished". If the Wait parameter is not specified, the launched process is executed in its own thread, and the R:BASE will continue to execute.

Examples

Example 01: (LAUNCH command with Parameters and Wait option)

```
LAUNCH C:\WordDocuments\TestFile.DOC|/MacroName|W
```

This will invoke the MS Word with the *TestFile.DOC* file using the *MacroName* as a main window while R:BASE sleeps in the background. You will have to Exit MS Word in order to give control back to R:BASE.

Example 02: (To automate emails via R:BASE)

If you know the email tags of your default email program, use the LAUNCH command to easily achieve email automation.

```
SET VAR vMailTo TEXT = NULL
SET VAR vCustID INTEGER = 2001
SELECT ('mailto:'+EMailAddress) INTO vMailTo INDIC IvMailTo +
FROM Contacts WHERE CustId = .vCustID
LAUNCH .vMailTo
```

This will launch default e-mail programs with filled in To:
You could follow the same scenario for other e-mail tags, such as:
Subject:
Cc:
Bcc:
Attached:
Message:

Example 03: (To automatically print a PDF file using Adobe Reader 7.0)

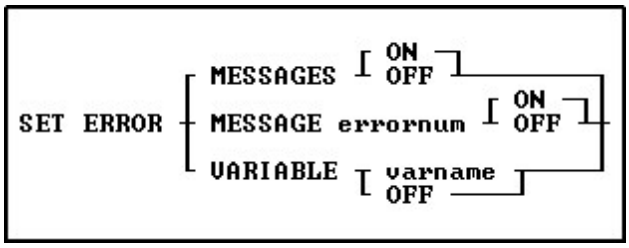
```
SET VAR vPDFFilename TEXT = '800023.pdf'
SET VAR vQuotes TEXT = (CVAL('QUOTES'))
SET VAR vPrintString = +
(.vQuotes+'C:\Program Files\Adobe\Acrobat 7.0\Reader\AcroRd32.exe|/p /h'
&.vPDFFilename+.vQuotes)
LAUNCH &vPrintString
RETURN
```

Part

8 Error Handling

8.1 Setting and Using Error Variables

SET ERROR VARIABLE defines an error variable to hold error message numbers. The variable name (*varname*) defines the variable R:BASE uses to hold R:BASE error codes. If set to off (the default), error variable processing is removed. The syntax for the SET ERROR VARIABLE command looks like this:



When an error occurs in a command file, R:BASE normally displays a system error message. **SET ERROR** enables a programmer to anticipate errors in command and procedure files and program the file to keep running even when an error occurs. You must always set ERROR VARIABLE off, rather than clearing it with the **CLEAR VARIABLES** command.

R:BASE resets the error variable to zero as each command is successfully run. If an error occurs, the error variable is set to the error number value. To determine the error condition for any line, you must immediately check the value of the error variable or capture the error value in a global variable for later examination.

By checking the error variable for a non-zero value, you can detect (or trap) many errors and run a sequence of error-handling commands such as an error-recovery procedure. Once the error number is captured in an error variable, you can write error-handling command files to control a program's flow based on these errors (error values).

The error variable value is set for each command that is run, not each line in a command file. If you have placed multiple commands on a line, the last command's error value is placed in the error variable. A similar situation occurs for multi-line commands such as the subcommands you can use when loading a data block with the **LOAD** command. For example, a data block loaded with the LOAD command leaves the error variable with a value of zero because the **END** command runs successfully, whether or not the data is actually loaded.

Rule violations do not set the error variable to a non-zero value; they are not the same as errors recognized by R:BASE.

The command below defines *errvar* as the current error variable:

```
SET ERROR VARIABLE errvar
```

When a command is run, R:BASE sets the error variable *errvar* to the error code before anything else happens. The following command lines illustrate how to use *errvar* in a command file.

```
LABEL tryagain
DIALOG 'Enter the database name: ' vdbname vEnd 1
CONNECT .vdbname
IF errvar <> 0 THEN
   PAUSE 2 USING 'Database not found.'
      GOTO tryagain
ENDIF
```

The first command establishes a label to return to, the second requests that the user enter the name of a database, and the third opens the specified database using the global variable defined by the DIALOG command.

The IF...ENDIF structure checks the error variable value. If the value is not zero (that is, if the database was not opened successfully), then it sends a message to the screen and passes control to the label *tryagain* so that the user is asked to enter the database name again.

You can also write a separate command file specifically designed to handle a variety of errors. In this case, the above code might look like this:

```
DIALOG 'Enter the database name: ' vdbname vEnd 1
CONNECT .vdbname
SET VARIABLE verr1 = .errvar
IF verr1 <> 0 THEN
       RUN errhandl.cmd USING .verr1
ENDIF
```

This series of commands captures the error value in the global variable *verr1* so that it can be passed through the USING clause of the RUN command to an error-handling routine. The routine itself determines the nature of the error and how to take care of the problem.

You can use the WHENEVER command to run status-checking routines for SQL commands. WHENEVER uses the special R:BASE variable SQLCODE.

8.2 Displaying an Error Message

You can display an error variable's message value by using the PAUSE 2 command. If you want to display *ErrVar* as used in the above program, enter the following command:

```
PAUSE 2 USING .ErrVar
```

This command allows you to display the actual error message text rather than writing your own as shown in the earlier program example.

If you have the **WINBEEP** set ON when executing applications, you may want to use the **BEEP** command to sound the bell when you display error messages with **SHOW ERROR**. To do this, you enter the single word **BEEP** on the line before you display the error message.

Part

9 Creating and Using Menus

As described in the introduction page of this document, R:BASE provides many different methods to store your series of R:BASE commands, which can be used to create your program. The same applies to how and where you wish to store your application menus.

- Command Files
- Application Files
- Forms/Variable Forms/External Form Files
- All, or some, of the above

Put it Down on Paper

Regardless of the above method(s) that you will use to contain your application, you have likely started to plan what your application will be based upon and what your requirements consist of. Before creating any menu system, you should write down a diagram of the menu structure of the application, and its associated actions, so that the whole process can be done in a logical manner.

1. Start by identifying the tasks that your application will perform. Consider the tables located within your database and include any tasks required to maintain the data within them. Also, include any data reports or mailing labels you will need to print.

2. Next consider how you can best organize these tasks into menus. You can organize your menus by any one, or a combination, of the ways identified here.

    - Do the tasks have to be performed in a certain order? If so, organize them by function.
    - Are some tasks performed more often than others? If so, organize them by their frequency of use.
    - Are the tasks related to the structure of the database? If so, organize them by table.

3. Using the menu titles and the tasks as the options under each menu, turn your list of tasks into a menu tree. A menu tree shows the order and hierarchy of menus in your application.

4. Be sure to add a menu action to exit the application.

Once you are ready to start building your application menus, review the available methods to see which suits your application requirements.

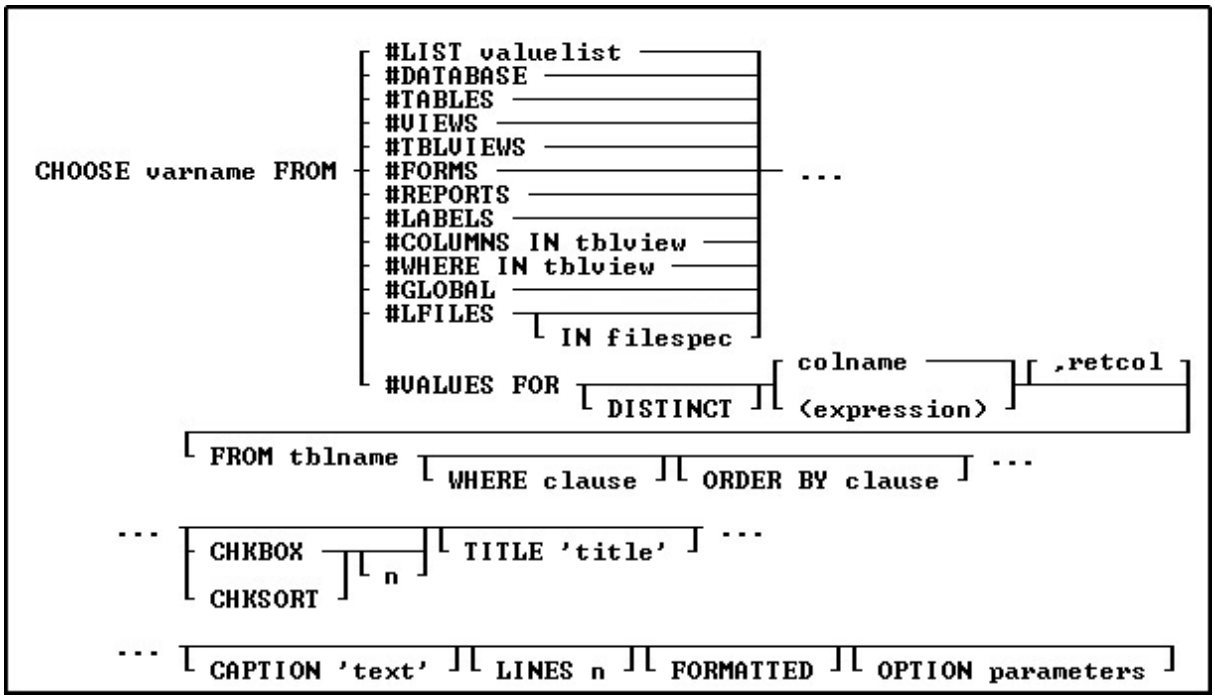
9.1 Command Files

Using only the R:BASE command syntax, specifically **CHOOSE**, an application can be entirely written in one or many RMD, APP, CMD, and/or DAT files. This method would place all of your work into ASCII text files, that any other text editor would also be allowed to open. If you anticipate that your application will require security measures to prevent users from tampering with your command syntax, then either "Application Files" or "Forms/External Form Files" would be better suited for your needs.

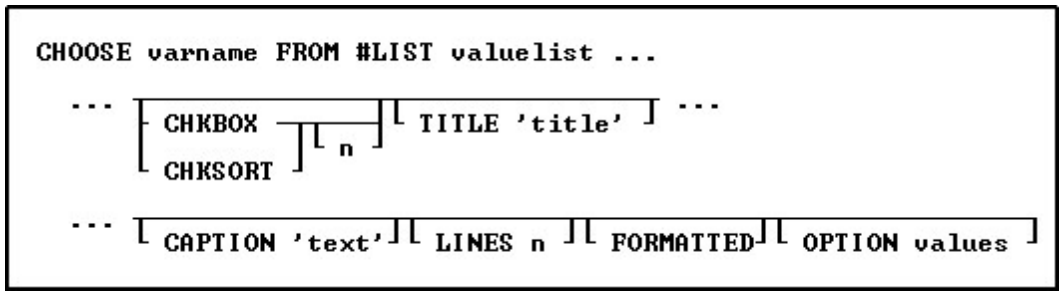
In legacy R:BASE versions, application files also used the CHOOSE command, only selecting from a menuname in a procedure file, which is no longer supported.

You will display your menus using the CHOOSE command. The CHOOSE command displays a formatted menu and enters the operator's selection into the variable specified in the command. That resulting variable will allow you to apply the user's selection to a Control Structure where your application can begin taking different directions.

This is the "complete" available syntax for the CHOOSE command:



With the intent of using the CHOOSE command to display your menu option list, you will be specifying the **#LIST** parameter, and the fewer following options would require your direct attention.



varname
Specifies the variable name that will hold the selected value from the menu.

#LIST valuelist
Allows you to specify a list of values in a comma delimited format. You can also use a variable that contains comma delimited values. The group of values MUST be encapsulated in quotes UNLESS a variable is used.

CHKBOX
Displays a check_box menu

CHKSORT
Displays a sort check_box menu

n
n is an optional positive integer specifying the maximum number of options on that menu that can be checked. If n is zero or is greater than the number of menu options, all options can be checked. If n is unspecified, the default value is zero. The maximum value of n is 9999.

TITLE 'title'
Displays a title in the window box

CAPTION 'text'
Displays text in the window box caption

LINES n
Determines the number of lines, n, to display in the listbox. The default is 10. If the number of items listed is greater than the values for LINES, a scroll bar will appear.

FORMATTED
Displays the CHOOSE box using a monospace font.

OPTION parameters
These parameters can be used to change a "look and feel" of CHOOSE window. All parameters for values must be separated by "|" (pipe) symbol.

Notes:

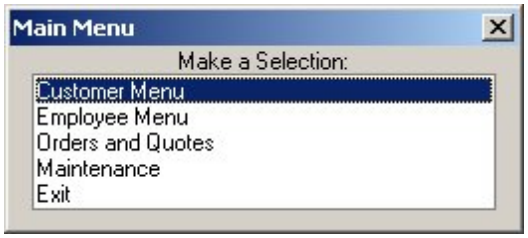
- CHOOSE will always be displayed at the center of your screen, unless specified using the **TOP *nn* LEFT *nn*** parameters.
- Do not pre-assign the data type for the CHOOSE variable, because the resulting variable will always be TEXT.
- The resulting variable will be left justified without leading spaces.
- If a user selects the "x" button or presses the "[Esc]" key, the CHOOSE variable value will be ***[Esc]***. Checking the CHOOSE variable value after issuing the CHOOSE command will be useful for defining exits or help screens in your control structure.

Displaying the CHOOSE

When R:BASE executes a CHOOSE command, it draws the menu on the screen. The following shows an example of how you might use a menu in a program for the higher positions in your application hierarchy.

```
CHOOSE vPick FROM #LIST +
'Customer Menu,Employee Menu,Orders and Quotes,Maintenance,Exit' +
TITLE 'Make a Selection:' CAPTION 'Main Menu' LINES 5
```

The window displayed would look like this:



After selecting the first option, the result value of *vPick* would be "Customer Menu".

A menu like the above will suit the needs of initial testing environment, but be sure to explore all of the additional **OPTION** parameters to make more appealing menus. The following CHOOSE command uses several OPTION parameters to enhance the menu display:

```
CHOOSE vPick FROM #LIST +
'Customer Menu,Employee Menu,Orders and Quotes,Maintenance,Exit' +
TITLE 'Make a Selection:' CAPTION 'Main Menu' LINES 5 +
OPTION List_Font_Color NAVY|List_Back_Color WHITE +
|List_Font_Size 12|LIST_BOLD ON +
|Title_Font_Color NAVY |Title_Back_Color LIGHT GRAY +
|Window_Back_Color LIGHT GRAY|Title_Font_Size 12 +
|Title_Font_Name ARIAL
```

The window displayed would look like this:

9.1.1 OPTION parameters

These parameters can be used to change the "look and feel" of a CHOOSE window. In addition to the parameters below, you can also alter the CHOOSE Title, List, and Buttons.

Parameter	Value	Description
WINDOW_CAPTION	OFF SMALL	Customizes the window caption. The OFF value makes window caption invisible. The SMALL value makes window caption small (tool window).
WINDOW_BACK_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes the color of the window background area. User can specify the integer value or type one of the predefined values.
TOP	value	Moves the CHOOSE box location, in pixels, from the top of the screen down.
LEFT	value	Moves the CHOOSE box location, in pixels, from the left of the screen right.
WHERE_CLAUSE	value	Allows you to pre-load a WHERE Clause into the WHERE Builder window when using "#WHERE IN tblview".
SHOW_ALL_BUTTON	ON OFF	Displays a "Select All" button when using the CHKBOX option.
SINGLE_CLICK	ON OFF	Limits the user to select a single option from the displayed list, and then immediately close. This option works only when user can select single item from a list. For multiple item selection this feature will be ignored.
THEMENAME	value	Specifies one of 86 pre-defined Themes, or a custom Theme loaded into R:BASE.

		NOTE: All previously defined CHOOSE commands which are redefined to use Themes should be thoroughly checked prior to putting into production as objects and text may look substantially different.

9.1.1.1 Title

These parameters are specific to changing how the CHOOSE "title" is displayed.

Parameter	Value	Description
TITLE_BACK_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes the background color of the title area. User can specify the integer value or use one of the predefined values.
TITLE_FONT_NAME	value	Specifies a font name for the title area. Values would consist of the fonts available on the computer.
TITLE_FONT_SIZE	value	Specifies a font size in title area.
TITLE_FONT_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes the font color of the title area. User can specify the integer value or type one of the predefined names.
TITLE_BOLD	ON OFF	Makes font in title area bold style.
TITLE_ITALIC	ON OFF	Changes the font in title area italic style.
TITLE_UNDERLINE	ON OFF	Changes the font in title area underlined.
TITLE_STRIKEOUT	ON	Makes the font in title area strikeout.

	OFF	

9.1.1.2 List

These parameters are specific to changing how the CHOOSE "list" is displayed.

Parameter	Value	Description
LIST_BACK_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes the background color of the list area. User can specify the integer value or type one of the predefined values.
LIST_FONT_NAME	value	Specifies a font name in the list area. Values would consist of the fonts available on the computer.
LIST_FONT_SIZE	value	Specifies a font size in the list area.
LIST_FONT_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes font color of the list area. User can specify the integer value or type one of the predefined values.
LIST_BOLD	ON OFF	Makes the font in the list area bold style.
LIST_ITALIC	ON OFF	Makes the font in the list area italic style.
LIST_UNDERLINE	ON OFF	Makes the font in the list area underlined.
LIST_STRIKEOUT	ON OFF	Makes the font in the list area strikeout

9.1.1.3 Buttons

These parameters are specific to changing how the CHOOSE "buttons" is displayed.

Parameter	Value	Description
BUTTON_OK_CAPTION	value	Changes the caption for the "OK" button.
BUTTON_CANCEL_CAPTION	value	Changes the caption for the "Cancel" button.
BUTTONS_SHOW_GLYPH	ON OFF	Places images on the OK and Cancel buttons.
BUTTONS_BACK_COLOR	AQUA BLACK BLUE CREAM DARK GRAY FUCHSIA GRAY GREEN LIME LIGHT GRAY MAROON MEDIUM GRAY MINT GREEN NAVY OLIVE PURPLE RED SILVER SKY BLUE TEAL WHITE YELLOW	Changes the color of the button area. User can specify the integer value or type one of the predefined values.

9.2 Application Files

Using the Application Builder interface, your menus can be stored in an .RBA file, separate from the database files. Within this interface you can store your menu structure and the underlining command files that are associated with each menu. Application files can be password protected, which offers additional security over using command files.

If you have completed the Tutorial, you will recall that you can add an Action to your application and instruct the Action to run a command file. In the Tutorial, the command file, MyProg.rmd, was added to the "Calculate Total for Date Range" Action.

In addition to assigning an Action to run an outside command file, you can also store your command file contents within the Application File itself by using the "Custom Action" option.

For additional information to fully understand setting up an application menu structure in the Application Builder, refer to Application Help and the Tutorial documentation.

9.3 Forms/Variable Forms/External Form Files

A menu structure can be stored in Forms, Variable Forms, and/or External Form Files. Using the many various controls available in the R:BASE designers, you can create an appealing hierarchy for your application.

Before going into detail regarding the available controls, review the following differences between Forms, Variable Forms, and External Form Files.

Functionality	Form	Variable Form	External Form File
Stored within the database files	Yes	Yes	No
Has a Database Explorer Menu Option	Yes	Yes (Forms)	Yes

Has its own designer interface	Yes	Yes (Form Designer)	Yes
Can contain table objects	Yes	No	No
Can contain variable objects	Yes	Yes	No
Can be launched when disconnected from a database	No	No	Yes

The difference between a Form and Variable Form is that when you create a new Form, you omit specifying a table name. This automatically creates a Variable Form. When you open the Variable Form in the Form Designer, the "Database Controls" tool bar that contains table controls will be disabled with other table driven options on the Menu Bar. When you open an External Form File in the External Form Designer, the "Database Controls" tool bar is also disabled.

The more commonly used objects used for displaying a menu in Forms, Variable Forms, and External Form Files include the following objects:

- Group Bar
- Tree View
- Buttons
- Drop-Down Menu Buttons
- Design Menu Bar

Each of these controls allow you to display an object with descriptive text and then place some method of underlying command syntax behind it to perform a task when the end users selects it. Like in the Application Designer, you can run a command file that is stored in the same directory as the database, or you can store your command syntax in the Form/Externals Form File. The underlying command syntax will be referred to as an EEP.

What is an EEP?

The letters E-E-P represent the term Entry and Exit Procedure. An EEP is basically a command file that can be run within forms (Forms/Variable Forms/External Form Files). An EEP can represent a command file that uses the .EEP file extension, or you can store the command syntax in the form, which is then referred to as a "Custom EEP". The only difference between an EEP and a Custom EEP is that the Custom EEP is stored in the form, whereas a regular EEP must be stored in a command file with the .EEP file extension.

Custom EEPs were introduced in an effort to decrease the amount of command files stored in a database folder, and to increase portability of a database application. In fact, if you locate all of your EEPs in custom EEPs, you can limit your database folder to just one command file as the startup file.

If you have taken the time to review the sample applications, which is highly recommended, you will notice that nearly all of the command syntax is stored in Custom EEPs.

There is no right or wrong way to store your command syntax, but storing your EEPs in the database directory may leave the syntax open to tampering by users. A Custom EEP will hide your code. Additional security is even available for you to password protect your forms.

To place your command syntax from a command file into a Custom EEP, simply select all ([Ctrl]+[A]) of your command file text in your file and use the "Copy & Paste" functionality of your operating system to drop the code into the Custom EEP. Whenever code is placed into a Custom EEP, you will notice that the "Custom EEP" button will turn yellow.

For additional information on the Form Designer and External Form Designer, refer to the Help files that are specific to each of these modules. To launch the Forms Help, select [Shift]+[F1] from within the Form Designer, or launch the Forms.chm file from within the R:BASE program directory. To launch the External Form File Help, select [Shift]+[F1] from within the External Form Designer, or launch the EForms.chm file from within the R:BASE program directory.

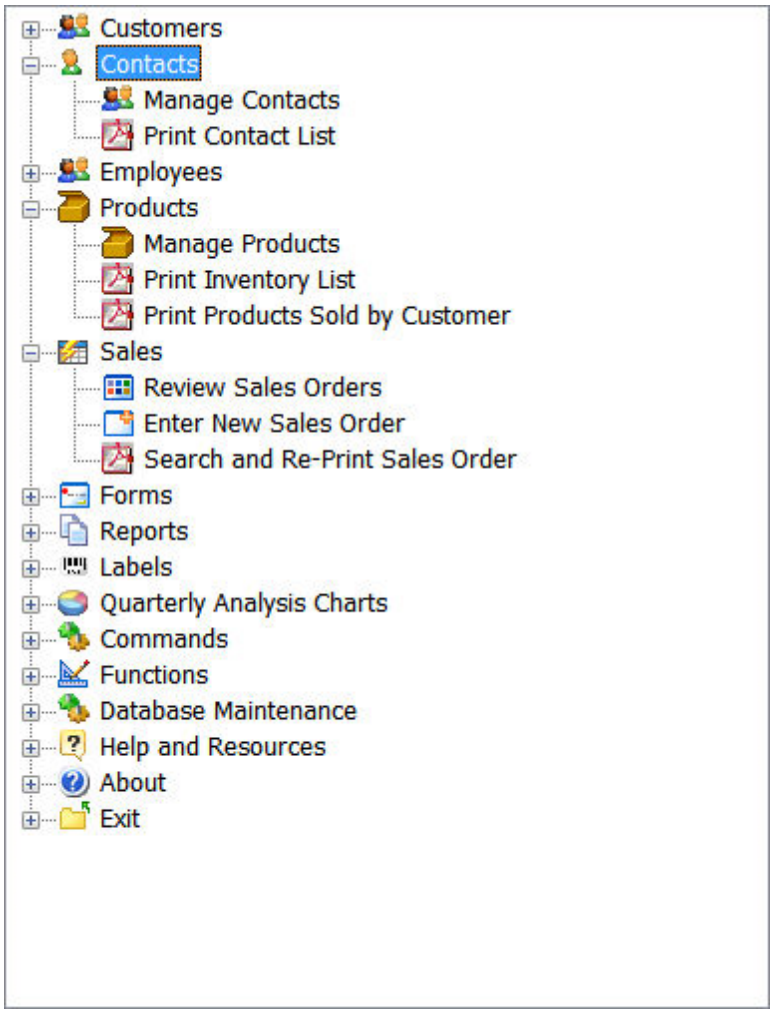
9.3.1 Group Bar

Use the Group Bar control to create a collection of user interface command groups. A command group typically contains a list of clickable items that invoke commands. These clickable items will run your EEPs. The Group Bar is a single control that supports all of these types of display styles. In particular, the Style property can be set to Category View, Task List, and Outlook. The default is set to Category View.

Typical examples of group bars include the navigational bar (e.g. Outlook Bar) in Microsoft Outlook and the category view in Windows XP's Control Panel (also called the Explorer Bar). If you have launched the RRBYW14 sample application, the Main Menu form displays an R:BASE Group Bar control listed on the left side.

9.3.2 Tree View

The Tree View control displays an object containing a hierarchical list of static items (nodes), such as the headings in a document, the entries in an index, or the files and directories on a disk. Each node consists of a label and an optional image, where each node can have a list of subitems associated with it. By selecting the tree structure, the user can expand or collapse the associated list of subitems. R:BASE command syntax can be stored or referenced within the item properties to accomplish a task.



9.3.3 Buttons

Form Buttons are another available option that can be used to perform tasks when selected. Buttons can be set to execute either Pre-Defined Actions, external command files (EEPs), embedded Custom EEPs, or Stored Procedures. The list of Pre-Defined Actions includes:

- Add Row
- Add Row and Exit
- Delete Row
- Discard Row
- Discard Row and Exit
- Duplicate Row
- Exit
- Next Row
- Next Table
- Previous Row
- Previous Table
- Save Row
- Last Row
- First Row
- Save Row and Exit
- Refresh Current Table

Some of the available buttons available in forms include:

- **Bit Button** - a push button object, which can load image files to be displayed on the button
- **Speed Button** - a push button object, which can load image files, but does not become part of the field tab order
- **Office Button** - a push button object that has a unique visually interactive behavior
- **Enhanced Speed Button** - a push button object, which can load image files and includes additional visual effects, but does not become part of the field tab order

Here are some Bit Buttons in a group:

9.3.4 Drop-Down Menu Buttons

The Drop-Down Menu Button is a push button object which presents a drop-down list of static values (menu items) that can execute either external command files (EEPs) or embedded Custom EEPs. This control is ideal for designing menu-driven applications on your forms. The Drop-Down Menu Button also allows you to load image files to be displayed on the button and listed menu items.

9.3.5 Design Menu Bar

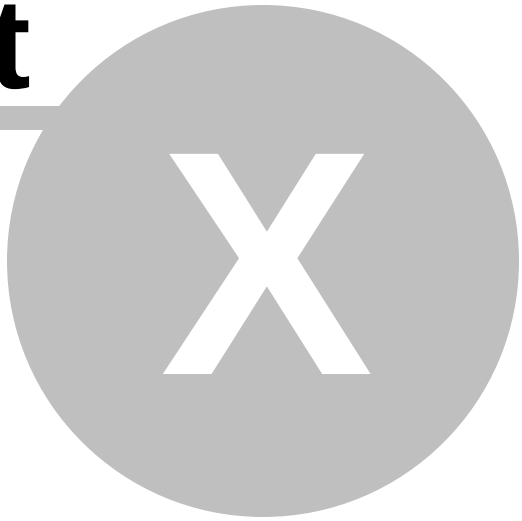
The Design Menu Bar is an interface that allows you to build a "Menu Bar" for your R:BASE form. The interface presents you with the Menu Designer dialog for designing an embedded menu bar for the form. The menu items added can execute either external command files (EEPs), or embedded Custom EEPs.

To open the "Menu Designer", select "Layout" > "Design Menu Bar" from within the Form Designer.

What is a Menu Bar?

A Menu Bar is commonly provided in nearly all computer software programs as a means to offer "word text" menu selections to users. The Menu Bar is usually the tool bar that is displayed in the top portion of an application. The Menu Bar in R:BASE starts with the word "Database" and ends with "Help".

Part



10 Debugging Your Program

The debugging phase of programming is usually the most time-consuming part of developing an application. This section describes some techniques you can use when testing and debugging your application programs.

If you plan on using CODELOCK, always test your program as an ASCII command file first. By thoroughly testing a command file in ASCII, you can save a great amount of time spent converting and reconverting your programs.

Either omit the SET MESSAGES OFF and SET ERROR MESSAGES OFF commands you would normally place at the beginning of your program, or enter the commands but format them as comments like this:

```
-- SET MESSAGES OFF
-- SET ERROR MESSAGES OFF
```

Leave them as comments until you have fully tested all of your command files.

10.1 Trace Debugger

The best way to debug your command files is to use the interactive R:BASE Trace Debugger. The Trace Debugger is a way for your to view your command files and scripts as your application runs, allowing you to find programming errors and problems. The Trace Debugger pinpoints exactly where errors occur by allowing you to do the following:

- View the results of commands
- Find problems in application logic or command syntax
- Process individual commands, one at a time
- Process blocks of commands
- Display variables and monitor their values
- Change variable values
- Save debugging configurations for the next debugging session

As opposed to the RUN command, which just executes the instructions in a command file, the Trace Debugger opens a command file and lets you observe the commands and variables as the instructions are processed. After the command(s) have executed or when the debugger pauses, the Trace Debugger updates variables you are monitoring and displays the command file code. Error messages appear when applicable.

The interactive Trace Debugger works with ASCII, command, or application files encrypted with CodeLock. The source files must be located in the same directory as the CodeLocked files. If you are going to Trace a CodeLocked application, then you must also follow the same R:BASE standard of (.APP) for the source files, and (.APX) for the CodeLocked files.

You can also debug Entry/Exit Procedures (EEPs) and ASCII files with any name.

Certain commands used in applications require keyboard input as the Trace Debugger processes them. You cannot return to the Trace Debugger when the system is processing the input. You must first exit the form, menu, dialog, or edit screen.

Here is a list of commands that require user input:

- BROWSE
- CHOOSE
- DIALOG
- EDIT
- EDIT USING
- ENTER
- FILLIN
- GATEWAY
- LBLPRINT
- PAUSE

- PRINT
- QUERY
- RBEDIT

The Trace Debugger is interrupted if an error or warning occurs while processing commands. This halt in debug processing occurs and the error or warning message is displayed irregardless of the ERROR MESSAGES and MESSAGES settings. **The error or warning message displayed relates to the last command processed. This is the command on the highlighted line in the Trace Debugger window.**

The Trace Debugger exits when the command file finishes. To exit sooner, press the exit icon at any time.

10.1.1 Using the Trace Debugger

Using the Trace Debugger is very simple in R:BASE. In the Database Explorer, simply click the Command Files button on the Toolbar to the left and, locate your file in the right panel, and press the Trace option.

You can also initialize the Trace debugger from the R> Prompt by using the TRACE command.

10.1.2 Breakpoints

You can interrupt the Trace Debugger at specific points as the command file is processed by setting up breakpoints. A breakpoint is a command in the command file that interrupts processing.

To execute the file between the present position and the next breakpoint, press the [F5] key.

Breakpoints are useful in the following situations:

- To advance to a specific part of a command file, such as an area that needs more debugging, without stepping through the whole file.
- To watch the results of each iteration through a WHILE loop--set a breakpoint near the end of the loop.
- To watch the results of an IF...ENDIF statement--place a breakpoint inside the IF...ENDIF block and processing stops when the condition(s) are true.

Follow these guidelines when using breakpoints:

- You can have a maximum of 25 breakpoints per session.
- If a command spans more than one line, you must set the breakpoint on the last line of the multi-line command.
- You cannot set a breakpoint on a comment. The Trace Debugger does not recognize breakpoints on non-executable lines.
- Breakpoints set on commands that start with DEBUG work only when DEBUG is set to ON.

NOTE: If you set a breakpoint inside a WHILE loop, the Trace Debugger will only stop on the breakpoint if the WHILE loop condition(s) are met. The same is true for IF...ENDIF and SWITCH...CASE blocks.

Setting Breakpoints

To set a breakpoint in the Trace Debugger, scroll to the place in the command file being debugged where you want the debug process to break, place your cursor on that line by clicking on it, and press the [F9] key or  icon from the toolbar. The line will highlight in red when the breakpoint is set.

Clearing Breakpoints

To clear a particular breakpoint, place the focus on the line by clicking on it, and press the [SHIFT] + [F11] keys or the  icon on the toolbar.

You can also clear all the breakpoints by pressing the [SHIFT] + [F7] keys or the  icon on the toolbar.

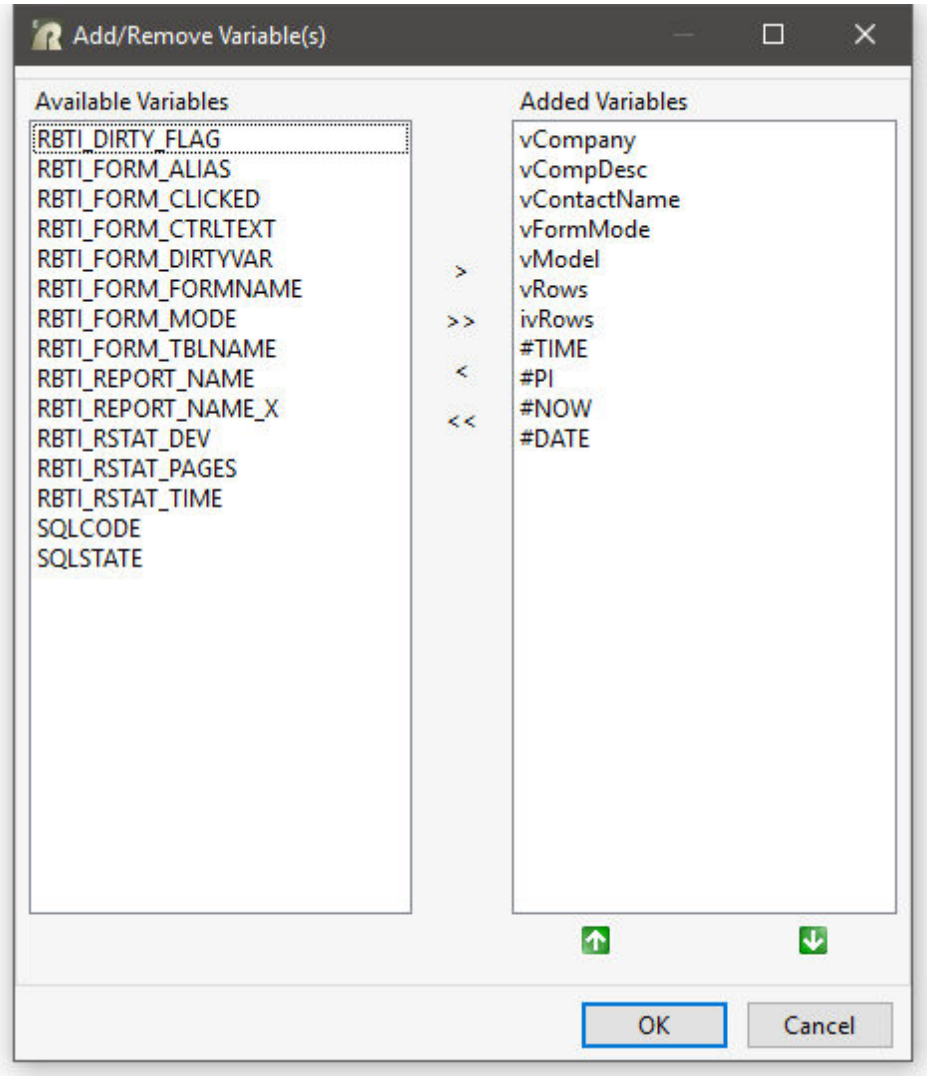
10.1.3 Watch Variables

A Watch Variable is a variable that you can monitor as the command file is processed in the Trace Debugger. Use Watch Variables to provide debugging information. After you add Watch Variables to your Watch Variable list, they appear in list fashion within a panel on the right side of the Trace Debugger window. Every time control returns to the Trace Debugger, the Watch Variable list refreshes.

10.1.3.1 Adding/Removing Watch Variables

Using the Add/Remove Variable(s) dialog, you can add, remove, and reorder your Watch Variables.

To access the Add/Remove Variable(s) dialog, press the [F6] key or the  icon on the toolbar.



The single arrow buttons displayed in the dialog permit a selected variable to be added to or removed from the watch variable list. The double arrow buttons will add or remove all available variables.

To reorder your Added Variables, select the variable whose position you wish to change, and press the green arrow button to raise or lower its position.

10.1.3.2 Modifying Watch Variable Values

Users can modify the value of any watch variable. To do so, several methods are available once the specific variable is selected in the watch variable list:

- press the [Ctrl] + [F8] key combination
- select the "Modify Variable Values" button  on the Trace Toolbar
- select "Watch" > "Modify Variable Values" from the Menu Bar

Once performing on of the above, you will be presented with a dialog displaying the current value of the variable which you can edit.

NOTE: Pay attention to the variable's data type because you will be able to type any alpha-numeric characters in the dialog, but only values matching the variable's current data type will be accepted.

10.1.3.3 Clearing Watch Variables

Users can clear all the currently selected variables from the watch variable list. To do so, several methods are available:

- press the [Ctrl] + [F7] hot key combination
- select the "Modify Variable Values" button on the Trace Toolbar
- select "Watch" > "Modify Variable Values" from the Menu Bar

WARNING: By clearing the watch variables, you are not only removing the variables from the Watch Variable toolbar list, but you are also clearing the variables from the instance of R:BASE. Therefore, if the variables are needed at a later point in the processing, they will need to be redefined.

10.1.3.4 Saving Watch Variables

Users can save the watch variable list to an outside file for use at a later time. This allows the ability to access your list without needing to reselect the variables again.

To save your watch variable list, select the "Save Watch Variable(s)" button on the Trace Toolbar. You will be prompted to save the file name to store the list, with the .DBG file extension. By default, the filename will be the same as the file being traced, only with a (.DBG) extension.

10.1.3.5 Loading Watch Variables

After saving a watch variable list, the list can be loading into an instance of the Trace Debugger.

To load watch variables from a saved list, press the "Load Watch Variables" button on the Trace Toolbar. Then, when prompted, and browse to the (.DBG) file that you wish to load and select the "Open" button. All variables in the saved list, which are not currently in the watch variable list, will be added.

10.1.4 Changing Variable Values

Changing variable values can produce unexpected results. When using statements such as IF, WHILE, and SWITCH, changing the value of the variable will not cause R:BASE to re-evaluate the conditions. Changing a variable does not update previous uses of the variable. For instance, if a DECLARE CURSOR command uses a variable, changing the value after the DECLARE command processes does not cause the CURSOR definition to change.

In certain R:BASE commands, variables are produced at the pre-parsing stage, while R:BASE is searching for the next command in the file. Since the Trace Debugger also uses the pre-parsing function to locate the next command, the value of some variables might already be in the command. In this case, R:BASE does not use the new value for the variable in the processing of the current command.

Also, the Trace Debugger does not allow you to change the data type of a variable. To change a data type, you must exit the Trace Debugger and insert the appropriate command in the command file.

10.1.5 Debugging CodeLocked Files and Blocks

It legacy R:BASE versions, it was supported to use $COMMAND, $MENU, and $SCREEN blocks in a single file and use CodeLock to produce a single application file. This process decreases the number of files to maintain and introduces code security to the application. Using CodeLock to encode an application prevents unauthorized access to the source code. The Trace Debugger can process applications developed in this manner only if the ASCII version (.APP) of the command file is present. Otherwise, it runs that portion of the application (.APX), rather than displaying the source code.

Unlike $COMMAND blocks, the Trace Debugger ignores $MENU and $SCREEN blocks. They are parsed internally by the CHOOSE and DISPLAY commands. When using the Trace Debugger on applications that use either of these commands, an ASCII version of $MENU or $SCREEN blocks is not required. R:BASE reads the CodeLocked files as needed.

10.1.6 Debugging EEP's

To execute your application until the point of a particular EEP, insert the string DEBUG TRACE as the first line of the EEP and make sure that DEBUG is set to ON (SET DEBUG ON). This will cause the Trace Debugger to initialize when it hits the DEBUG TRACE line in the EEP code.

10.1.7 Debugging Nested RUN Commands

You can use a RUN command in a command file when you want to control the program flow, to modularize application development, or to improve application speed. The Trace Debugger can debug these command files each time it finds a RUN command.

Using RUN commands affects the number of files open at one time in R:BASE. If you have your FILES setting at 5, once a sixth RUN command processes, the first file opened will not be accessible. The Trace Debugger does not change the number of RUN commands that can be called. You can, however, change the number of RUN commands that can be called with the SET FILES command.

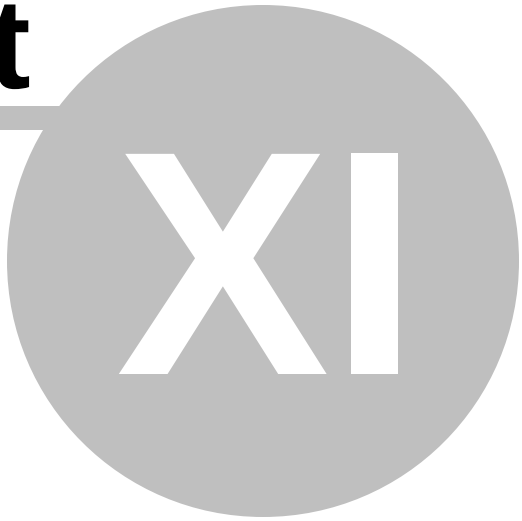
Each time the RUN command processes, R:BASE opens an additional file. Each file is displayed as a separate tab in the Trace Debugger.

When an ASCII command file is called by a RUN command, R:BASE displays it automatically in the Trace Debugger. When you develop applications using CodeLocked files (.APX), R:BASE only displays a command file called by a RUN command if the ASCII version is present (.APP).

If the RUN command executes a $COMMAND block within a CodeLocked file, the Trace Debugger highlights the first executable line of the command block. If R:BASE does not find the ASCII version of the CodeLocked file, R:BASE runs the command block from the CodeLocked file and returns to the Trace Debugger when the RUN command finishes.

Combining ASCII and CodeLocked files during development will not affect the Trace Debugger. Any time the ASCII file is available, the Trace Debugger screen appears. ASCII files run from CodeLocked files appear in the Trace Debugger even if the ASCII version of the CodeLocked file is not available.

Part



11 Using Optimizing Techniques

Use the following techniques in your command files and EEPs to speed up processing:

- **Group similar commands**

  R:BASE loads into memory the information needed to process each command. If the command is already in memory, R:BASE does not need to read the disk again. For example, try grouping separate SET VARIABLE commands into one SET VARIABLE whenever possible.

- **Minimize disk access**

  Minimize disk access by using Custom EEPs, which run from the computer's memory, and by combining small command files into command blocks within a procedure file.

- **Use WHILE loops**

  Use WHILE loops instead of IF structures or GOTO processing whenever possible. All the commands contained within a WHILE loop are completely read and are retained in memory as long as the WHILE loop is processing. Use BREAK or change the condition for a natural exit rather than using GOTO to exit from a WHILE loop.

- **Use forward GOTO searches**

  Use forward GOTO searches whenever possible. When R:BASE encounters a GOTO, it performs a forward search for the matching label more efficiently than by seeking it in memory, even though the labels are retained internally.

- **Predefine variables**

  Make sure that the data type of each variable is unambiguous by explicitly assigning the data type when the variable is defined.

- **Reduce redundancy**

  Eliminate duplication of command sections where possible. If you have a set of commands duplicated in several places within a form, use a "Custom Form Action" which can be called upon at any place in the form. If you have a set of commands duplicated in several places within the entire application, use "Stored Procedures", which can be called upon from almost any place within the application.

  For multiple commands duplicated in a file, put those commands in a separate command block in the procedure file and use RUN to execute the commands where needed.

- **Eliminate rules checking**

  Set RULES OFF when not needed for data verification. This saves time by preventing R:BASE from checking the data for rule violations.

- **Experiment with Manual Table-Order Optimization**

  By default, R:BASE uses its own internal algorithm optimizer that determines the best order for joining tables. You can turn off the automatic optimizer using the MANOPT setting. MANOPT (default:OFF) disables the automatic table-order optimization that R:BASE performs when running queries. This gives maximum control over the order in which columns and tables are assembled in response to a query. With MANOPT set to ON, R:BASE uses the order of the tables in the FROM clause and the order of the columns in the column list of the SELECT clause to construct the query.

Part

12 Useful Resources

. R:BASE Home Page:	https://www.rbase.com
. Up-to-Date R:BASE Updates:	https://www.rbaseupdates.com
. Current Product Details and Documentation:	https://www.rbase.com/rbgx5
. Support Home Page:	https://www.rbase.com/support
. Product Registration:	https://www.rbase.com/register
. Official R:BASE Facebook Page:	https://www.facebook.com/rbase
. Sample Applications:	https://www.razzak.com/sampleapplications
. Technical Documents (From the Edge):	https://www.razzak.com/fte
. Education and Training:	https://www.rbase.com/training
. Product News:	https://www.rbase.com/news
. Upcoming Events:	https://www.rbase.com/events
. R:BASE Online Help Manual:	https://www.rbase.com/support/rsyntax
. Form Properties Documentation:	https://www.rbase.com/support/FormProperties.pdf
. R:BASE Beginners Tutorial:	https://www.rbase.com/support/rtutorial
. R:BASE Solutions (Vertical Market Applications):	https://www.rbase.com/products/rbasesolutions

Part

13 Feedback

Suggestions and Enhancement Requests:

From time to time, everyone comes up with an idea for something they'd like a software product to do differently.

If you come across an idea that you think might make a nice enhancement, your input is always welcome.

Please submit your suggestion and/or enhancement request to the R:BASE Developers' Corner Crew (R:DCC) and describe what you think might make an ideal enhancement. In R:BASE, the R:DCC Client is fully integrated to communicate with the R:BASE development team. From the main menu bar, choose "Help" > "R:DCC Client". If you do not have a login profile, select "New User" to create one.

If you have a sample you wish to provide, have the files prepared within a zip archive prior to initiating the request. You will be prompted to upload any attachments during the submission process.

Unless additional information is needed, you will not receive a direct response. You can periodically check the status of your submitted enhancement request.

If you are experiencing any difficulties with the R:DCC Client, please send an e-mail to rdcc@rbase.com.

Reporting Bugs:

If you experience something you think might be a bug, please report it to the R:BASE Developers' Corner Crew. In R:BASE, the R:DCC Client is fully integrated to communicate with the R:BASE development team. From the main menu bar, choose "Help" > "R:DCC Client". If you do not have a login profile, select "New User" to create one.

You will need to describe:

- What you did, what happened, and what you expected to happen
- The product version and build
- Any error message displayed
- The operating system in use
- Anything else you think might be relevant

If you have a sample you wish to provide, have the files prepared within a zip archive prior to initiating the bug report. You will be prompted to upload any attachments during the submission process.

Unless additional information is needed, you will not receive a direct response. You can periodically check the status of your submitted bug.

If you are experiencing any difficulties with the R:DCC Client, please send an e-mail to rdcc@rbase.com.

Index

- # -

- & -

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- I -

- L -

- M -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

Notes