mvBase mvBASIC User Reference Guide Overview

The mvBASIC User Reference Guide documents the mvBase version of DATA/BASIC, and contains these major sections:

|  |  |
| --- | --- |
| Syntax Notations | Describes the syntax conventions that are used throughout this document. |
| Introduction to mvBASIC | Introduces mvBASIC, summarizes some of its features and enhancements, and recommends a sequence for reading the sections of this guide. |
| Creating mvBASIC Programs | Describes how programs are created. It goes into more detail than the average beginner requires, so most readers may skim it for the general concepts and return to it for details as necessary. |
| Format, Data and Expressions | Describes program format, data types, and the syntax for expressions. It should be read carefully. |
| Overview of mvBASIC Statements and Functions | Provides a tour of the mvBASIC language. Every statement and function is covered by topic and should be read very carefully. It is recommended that readers stop periodically, perhaps after every subsection of "Overview of Statements and Functions" to try the material just covered. "Statement and Function Reference" can be used for reference at this stage. |
| Using the mvBASIC Debugger | Provides a quick summary of Debugger commands, and several task-oriented topics related to using the Debugger. |
| Statement and Function Reference | Provides a comprehensive listing of mvBASIC functions and statements, their syntax parameters, and several examples of their use. |
| O/S Interoperability Commands | Provides a comprehensive listing of O/S interoperability commands, their syntax parameters, and several examples of their use. |
| Appendix A: Error Messages | Provides a comprehensive listing of mvBASIC error messages. |
| Appendix B: List of ASCII Codes | Provides a table which summarizes ASCII codes and compares them with decimal, hex, and character values. |
| Appendix C: mvBASIC Program Examples | Provides diverse examples of actual mvBASIC programs in order to show a variety of programming techniques. |

Contents

[mvBase mvBASIC User Reference Guide Overview 1](#_Toc449701857)

[Syntax Notations 9](#_Toc449701858)

[mvBASIC Syntax Notations 10](#_Toc449701859)

[Introduction to mvBASIC 11](#_Toc449701860)

[mvBASIC Enhancements 11](#_Toc449701861)

[Creating mvBASIC Programs 14](#_Toc449701862)

[Creating the Program File 15](#_Toc449701863)

[Editing and Listing the Source Code 16](#_Toc449701864)

[Listing the Program with BLIST 17](#_Toc449701865)

[Compiling the Program 19](#_Toc449701866)

[Option for Debugging a Program (X) 20](#_Toc449701867)

[Options for Cataloging (C, S) 21](#_Toc449701868)

[Printing Compiler Output (P) 21](#_Toc449701869)

[Specifying a Version of the Operating System 22](#_Toc449701870)

[Obtaining the Compilation Time and Date 22](#_Toc449701871)

[Running the Program 24](#_Toc449701872)

[Cataloging the Program 25](#_Toc449701873)

[A Sample Program 27](#_Toc449701874)

[Variables 29](#_Toc449701875)

[Assigning and Using Variables 29](#_Toc449701876)

[Data Typing in mvBASIC 29](#_Toc449701877)

[Advanced Variables 29](#_Toc449701878)

[Array Variables 30](#_Toc449701879)

[Dimensioned Arrays 31](#_Toc449701880)

[File Variables 31](#_Toc449701881)

[Select-list Variables 32](#_Toc449701882)

[Reading and Writing Tapes or Floppy Disks 33](#_Toc449701883)

[READ Statement 33](#_Toc449701884)

[UREADLINE Function 34](#_Toc449701885)

[UREAD Command 36](#_Toc449701886)

[Compiler Directives 38](#_Toc449701887)

[Comments in the Object Code 38](#_Toc449701888)

[Reading External Source Code ($CHAIN, $INCLUDE, $INSERT) 38](#_Toc449701889)

[Reading and Updating File Items 39](#_Toc449701890)

[File Variables (OPEN) 39](#_Toc449701891)

[Reading and Writing a File Item (READ, WRITE, etc.) 39](#_Toc449701892)

[File Item Locks (READU, WRITEU, RELEASE, etc.) 40](#_Toc449701893)

[The LOCKED Clause 40](#_Toc449701894)

[Select-lists (SELECT, READNEXT) 40](#_Toc449701895)

[Deleting File Items (DELETE, CLEARFILE) 41](#_Toc449701896)

[READT Statement 41](#_Toc449701897)

[READNEXT Statement 43](#_Toc449701898)

[READTX Statement 44](#_Toc449701899)

[CHAIN Statement 45](#_Toc449701900)

[READV Statement 45](#_Toc449701901)

[READB Statement 47](#_Toc449701902)

[READF Statement 47](#_Toc449701903)

[Reading and Writing Tapes or Floppy Disks 48](#_Toc449701904)

[Case Construct 49](#_Toc449701905)

[IF Construct 51](#_Toc449701906)

[MATREAD Statement 53](#_Toc449701907)

[OPEN Statement 55](#_Toc449701908)

[The Print and CRT Statements 57](#_Toc449701909)

[Sending Output to the Screen and Printer 57](#_Toc449701910)

[Output Devices (PRINT, CRT, DISPLAY) 57](#_Toc449701911)

[Sending Output to the Printer (PRINTER) 57](#_Toc449701912)

[Print Units 57](#_Toc449701913)

[Formatting and Positioning Output 58](#_Toc449701914)

[Tabulation and Carriage Return Suppression 58](#_Toc449701915)

[Formatted Screens (@) 58](#_Toc449701916)

[Masking Data (FMT) 59](#_Toc449701917)

[Headings and Footings 59](#_Toc449701918)

[The PRINTERR Statement 60](#_Toc449701919)

[PRINT Statement 61](#_Toc449701920)

[CRT (or DISPLAY – not used) Command 63](#_Toc449701921)

[CRT Statement 63](#_Toc449701922)

[Using the mvBASIC Debugger 65](#_Toc449701923)

[Debugger Commands: Quick Reference 66](#_Toc449701924)

[Fixing a Bug 67](#_Toc449701925)

[A Sample Program 68](#_Toc449701926)

[Printing Source Code 68](#_Toc449701927)

[Using Breakpoints and Trace Variables 70](#_Toc449701928)

[Displaying and Changing a Variable 71](#_Toc449701929)

[Using Execution Steps 71](#_Toc449701930)

[Assigning New Values for Testing 72](#_Toc449701931)

[Entering the Debugger 74](#_Toc449701932)

[Exiting the Debugger 75](#_Toc449701933)

[Displaying and Changing a Variable 76](#_Toc449701934)

[Displaying All Variables 76](#_Toc449701935)

[Displaying and Changing Simple Variables 76](#_Toc449701936)

[Displaying and Changing Dimensioned Array Elements 77](#_Toc449701937)

[String Windows ([) 78](#_Toc449701938)

[Accessing Source Code 79](#_Toc449701939)

[Identifying Source Code (Z) 79](#_Toc449701940)

[Displaying Source Code (L, $, ?) 79](#_Toc449701941)

[Using Breakpoints and Tracing 81](#_Toc449701942)

[Establishing a Breakpoint (B) 81](#_Toc449701943)

[Deleting a Breakpoint (K) 82](#_Toc449701944)

[Defining Trace Variables (T) 82](#_Toc449701945)

[Deleting a Trace Variable (U) 83](#_Toc449701946)

[Displaying Breakpoints and Trace Variables (D) 83](#_Toc449701947)

[Using Execution Control 84](#_Toc449701948)

[Continue Execution (G) 84](#_Toc449701949)

[Setting an Execution Step (E) 84](#_Toc449701950)

[Ignoring Breakpoints (N) 85](#_Toc449701951)

[Printing Output 86](#_Toc449701952)

[Toggling Program Output (P) 86](#_Toc449701953)

[Toggling Line Printing (LP) 86](#_Toc449701954)

[Close the Printer (PC) 86](#_Toc449701955)

[Using the Return Stack 87](#_Toc449701956)

[Displaying the Return Stack (S) 87](#_Toc449701957)

[Popping the Return Stack (R) 87](#_Toc449701958)

[Modal Traps (M) 87](#_Toc449701959)

[Additional mvBase specific commands 88](#_Toc449701960)

[Format, Data and Expressions 89](#_Toc449701961)

[Program Format 90](#_Toc449701962)

[Types of Statements 90](#_Toc449701963)

[Statement Labels 91](#_Toc449701964)

[Writing Readable Code 91](#_Toc449701965)

[Using Remarks 92](#_Toc449701966)

[Constants, Variables, and Data Types 93](#_Toc449701967)

[Assigning and Using Constants 93](#_Toc449701968)

[Assigning and Using Variables 93](#_Toc449701969)

[Data Typing in mvBASIC 93](#_Toc449701970)

[Building Expressions 95](#_Toc449701971)

[Simple Assignment 95](#_Toc449701972)

[Using Operators and Functions 96](#_Toc449701973)

[Numeric Expressions 97](#_Toc449701974)

[Arithmetic Operators 97](#_Toc449701975)

[Parentheses in Expressions 97](#_Toc449701976)

[Character Strings in Arithmetic Expressions 98](#_Toc449701977)

[Intrinsic Mathematical Functions 98](#_Toc449701978)

[String Expressions 99](#_Toc449701979)

[Logical Data (Booleans) 100](#_Toc449701980)

[Relational Operators 100](#_Toc449701981)

[Logical Operators 101](#_Toc449701982)

[The MATCH Operator 102](#_Toc449701983)

[Logical Functions 102](#_Toc449701984)

[Overview of mvBASIC Statements and Functions 104](#_Toc449701985)

[Assignment Statements 105](#_Toc449701986)

[Intrinsic Functions 107](#_Toc449701987)

[Internal Program Control 109](#_Toc449701988)

[The IF Conditional 109](#_Toc449701989)

[The THEN and ELSE Clauses 109](#_Toc449701990)

[CASE Constructs 110](#_Toc449701991)

[The LOOP Construct 110](#_Toc449701992)

[FOR Loops 111](#_Toc449701993)

[Stopping a Program (STOP, ABORT, END) 111](#_Toc449701994)

[The END Statement 111](#_Toc449701995)

[Internal Subroutines (GOSUB, RETURN) 112](#_Toc449701996)

[External Program Control 113](#_Toc449701997)

[External Subroutines (CALL, SUBROUTINE, RETURN) 113](#_Toc449701998)

[Passing Parameters (COMMON) 113](#_Toc449701999)

[Executing a TCL Command (EXECUTE, CHAIN, DATA) 114](#_Toc449702000)

[The DATA Statement 114](#_Toc449702001)

[Using EXECUTE with Select-lists 114](#_Toc449702002)

[Executing Another mvBASIC Program (ENTER) 115](#_Toc449702003)

[CAP-HUSH-ON and CAP-HUSH-OFF Commands 115](#_Toc449702004)

[Executing a Windows Command Line Command 115](#_Toc449702005)

[Sending Output to the Screen and Printer 116](#_Toc449702006)

[Output Devices (PRINT, CRT, DISPLAY) 116](#_Toc449702007)

[Sending Output to the Printer (PRINTER) 116](#_Toc449702008)

[Print Units 116](#_Toc449702009)

[Formatting and Positioning Output 116](#_Toc449702010)

[Tabulation and Carriage Return Suppression 117](#_Toc449702011)

[Formatted Screens (@) 117](#_Toc449702012)

[Masking Data (FMT) 118](#_Toc449702013)

[Headings and Footings 118](#_Toc449702014)

[The PRINTERR Statement 119](#_Toc449702015)

[Terminal Input 120](#_Toc449702016)

[The INPUT Statement 120](#_Toc449702017)

[Variations on INPUT 120](#_Toc449702018)

[Input from the Type-ahead Buffer (INPUTIF) 120](#_Toc449702019)

[Masked Input Statements (INPUT @) 121](#_Toc449702020)

[INPUTTRAP, INPUTNULL and INPUTERR 121](#_Toc449702021)

[INPUT and the Data Stack 122](#_Toc449702022)

[Dynamic Array Processing 123](#_Toc449702023)

[File Items and Dynamic Arrays 123](#_Toc449702024)

[Dynamic Array Functions 123](#_Toc449702025)

[The LOCATE Statement 124](#_Toc449702026)

[Alternate Forms for Dynamic Array Processors 124](#_Toc449702027)

[Counting Delimiters and Substrings 125](#_Toc449702028)

[Generalized String Processing 126](#_Toc449702029)

[Substring Assignment 126](#_Toc449702030)

[The FIELD Function 126](#_Toc449702031)

[The COL1, COL2, and LEN Functions 127](#_Toc449702032)

[The INDEX Function 127](#_Toc449702033)

[Trimming Spaces 127](#_Toc449702034)

[Converting Characters 127](#_Toc449702035)

[Dimensioned Arrays 128](#_Toc449702036)

[Assigning Dimensioned Array Variables (DIM) 128](#_Toc449702037)

[Converting Strings to Dimensioned Arrays 128](#_Toc449702038)

[MATREAD and MATWRITE 129](#_Toc449702039)

[The MAT Statement 129](#_Toc449702040)

[Reading and Updating File Items 131](#_Toc449702041)

[File Variables (OPEN) 131](#_Toc449702042)

[Reading and Writing a File Item (READ, WRITE, etc.) 131](#_Toc449702043)

[File Item Locks (READU, WRITEU, RELEASE, etc.) 131](#_Toc449702044)

[The LOCKED Clause 132](#_Toc449702045)

[Select-lists (SELECT, READNEXT) 132](#_Toc449702046)

[Deleting File Items (DELETE, CLEARFILE) 132](#_Toc449702047)

[Reading and Writing Tapes or Floppy Disks 133](#_Toc449702048)

[Communications 134](#_Toc449702049)

[Unlinking and Attaching a Line 134](#_Toc449702050)

[Sending Data to a Line (SEND, SENDX, SENDBREAK) 134](#_Toc449702051)

[Sending a BREAK 134](#_Toc449702052)

[Receiving Data from a Line 134](#_Toc449702053)

[The Type-ahead Buffer 134](#_Toc449702054)

[Input from an Attached Line (GET, GETX) 135](#_Toc449702055)

[Execution Locks 136](#_Toc449702056)

[The THEN and ELSE Clauses to LOCK 136](#_Toc449702057)

[Compiler Directives 137](#_Toc449702058)

[Comments in the Object Code 137](#_Toc449702059)

[Reading External Source Code ($CHAIN, $INCLUDE, $INSERT) 137](#_Toc449702060)

[Miscellaneous Statements and Functions 138](#_Toc449702061)

[Conversion Codes (ICONV, OCONV) 138](#_Toc449702062)

[The SENTENCE ( ) Function 138](#_Toc449702063)

[The SYSTEM Function 138](#_Toc449702064)

[Entering the Debugger 139](#_Toc449702065)

[The Error Message Processor 140](#_Toc449702066)

[Statement and Function Reference 143](#_Toc449702067)

[$\* Statement 146](#_Toc449702068)

[$CHAIN Statement 147](#_Toc449702069)

[$INCLUDE/$INSERT Statement 149](#_Toc449702070)

[= Statement 151](#_Toc449702071)

[[ ]= Statement 153](#_Toc449702072)

[@ Function 156](#_Toc449702073)

[ABORT Statement 163](#_Toc449702074)

[ABS Function 164](#_Toc449702075)

[ALPHA Function 165](#_Toc449702076)

[ASCII Function 167](#_Toc449702077)

[ASSIGNED/UNASSIGNED Function 168](#_Toc449702078)

[ATTACH TAPE DEVICES/ CHANGE BLOCK-SIZE Statement 169](#_Toc449702079)

[AUX ON/AUX OFF Statement 170](#_Toc449702080)

[BLOCK/UNBLOCK Statement 171](#_Toc449702081)

[BREAK Statement 173](#_Toc449702082)

[CALL Statement 175](#_Toc449702083)

[Passing Arrays 175](#_Toc449702084)

[CASE Construct 177](#_Toc449702085)

[CHAIN Statement 179](#_Toc449702086)

[CHANGE Function 180](#_Toc449702087)

[CHAR Function 181](#_Toc449702088)

[CLEAR Statement 182](#_Toc449702089)

[CLEARCOMMON Statement 183](#_Toc449702090)

[CLEARDATA Statement 184](#_Toc449702091)

[CLEARFILE Statement 185](#_Toc449702092)

[CLEARSELECT Statement 186](#_Toc449702093)

[COL1 Function 187](#_Toc449702094)

[COL2 Function 189](#_Toc449702095)

[COMMON Statement 190](#_Toc449702096)

[COMPARE Statement 192](#_Toc449702097)

[CONNECT/ DISCONNECT Statement 193](#_Toc449702098)

[CONSOLE Statement 194](#_Toc449702099)

[CONVERT Function 195](#_Toc449702100)

[CONVERT Statement 196](#_Toc449702101)

[COS Function 197](#_Toc449702102)

[COUNT Function 198](#_Toc449702103)

[CRT Statement 199](#_Toc449702104)

[CRT ON Statement 200](#_Toc449702105)

Syntax Notations

The following conventions are used throughout mvBase documentation for indicating command line syntax:

|  |  |
| --- | --- |
| ***italics*** | Anything shown in italics is variable information for which the user provides a specific value. |
| ***options*** | If two or more options are specified, separate the options with a space. |
| **( )** | Parentheses must be typed. It is usually sufficient to type only the first parenthesis; the second is normally optional. |
| **[ ]** | Anything shown enclosed within square brackets is optional. The square brackets themselves are not typed. |
| **|** | A vertical bar that separates two or more elements indicates that any one of the elements can be typed. |
| **{ }** | If two or more element are enclosed within curly braces and separated by a vertical bar, one of the elements must be typed. |

All punctuation marks that are included in syntax format lines (e.g., commas, parentheses, angle brackets, underscores, hyphens) are required in the syntax unless indicated otherwise. Square brackets are not typed.

The following command line is an example that incorporates these syntax notations.

|  |
| --- |
| LIST [DICT] *filenam*e [WITH [EVERY | EACH] *attribute-name* *value-list*] [(P)] |

The only two elements of the line that must be entered are **LIST** and ***filename***. **LIST** must be entered exactly as shown. ***filename*** is a variable; the user can enter the name of any accessible file. ***attribute-name*** and ***value-list*** are also variables that the user supplies. The vertical bar indicates that either **EVERY** or **EACH** can be entered; the brackets indicate that both of these components are optional. If the **P** option is entered, it must be enclosed within parentheses.

When variables that the user supplies are two or more words long, hyphens are used instead of blank spaces to separate the words in order to show that only one element is required. This command line is an example.

|  |
| --- |
| LIST *filename* *item-list* |

The word ***filename*** indicates a single element, and the words ***item*** and ***list*** joined by a hyphen likewise indicate a single element.

mvBASIC Syntax Notations

Syntax notations for mvBASIC are very similar to the standard syntax notations, but they do differ in minor ways. This topic summarizes syntax notations specific to mvBASIC.

|  |  |
| --- | --- |
| ***italics*** | Anything shown in italics is variable information for which the user provides a specific value. |
| **[ ]** | Anything shown enclosed within square brackets is optional unless indicated otherwise. The square brackets themselves are not typed unless they are shown in bold. |
| **|** | A vertical bar that separates two or more elements indicates that any one of the elements can be typed. |
| **< >** | Bold angle brackets are part of the syntax, and must be typed unless indicated otherwise. |
| **{ }** | Bold parentheses are a part of the syntax. both parentheses must be typed unless indicated otherwise. |

All punctuation marks that are included in syntax format lines (e.g., commas, parentheses, angle brackets, underscores, hyphens) are required in the syntax unless indicated otherwise.

The following command line is an example that incorporates these syntax notations.

|  |
| --- |
| OPEN [‘DICT’] *filename* TO *filevar* THEN | ELSE *statements* |

The keywords **OPEN** and **TO** must be specified. Either the **THEN** or **ELSE** clause must be specified, but both are not necessary. The user must supply appropriate values for ***filename***, ***filevar***, and ***statements***. The keyword **DICT** is optional, but if it is included, it must be enclosed within parentheses.

When variables that are supplied by the user are two or more words long, hyphens are used instead of blank spaces to separate the words in order to show that only one element is required. For example, in the following statement, the word ***count-var*** indicates a single element.

|  |
| --- |
| GET *var* [,*length*] SETTING *count-var*... |

# Introduction to mvBASIC

The Development of mvBASIC

The mvBASIC Language is an extended version of Dartmouth BASIC, specifically designed for data base management. Developed at Dartmouth College in 1963, Dartmouth BASIC is a language especially easy for the beginning programmer to master.

Consequently, practically every programmer today knows at least a little BASIC; therefore, because of its similarity to BASIC, mvBASIC is instantly familiar to almost every programmer. In addition, mvBASIC it is infinitely more flexible and more powerful than Dartmouth BASIC.

## mvBASIC Enhancements

Some of mvBASIC’s enhancements, which represent improvements over Dartmouth BASIC, are listed here to suggest what mvBASIC can accomplish.

Program Format

|  |  |
| --- | --- |
| Statement Labels | In Dartmouth BASIC, numeric statement labels are mandatory for each line of source code. The statements are stored according to their statement labels, so they are not executed in linear sequence (that is, according to where they appear in the source text), but in numeric sequence.  In mvBASIC, statements are executed in the order in which they appear. Statement labels are not mandatory for each line. Furthermore, statement labels do not have to be numeric: alphabetic and alphanumeric labels are also supported, with the provision that the label must end with a colon (**:**). A particularly useful feature is that there is no limit to the length of a statement label, as long as the file item does not exceed 248K. |
| Multiple Statements | mvBASIC allows several statements to be written on the same line of source code, as long as they are separated by a semicolon (**;**). |
| Variable Names | Variables can have any name of any length, as long as the file item does not exceed 248K. |
| Fixed Point Arithmetic | Computations are done with fixed-point arithmetic, with 19-digit precision and up to 9 decimal digits. |

New Features

|  |  |
| --- | --- |
| O/S Interoperability Commands | This is one major and unique feature of mvBASIC. O/S interoperability commands are executable from within mvBASIC, and enable the user to execute Windows programs and/or to access Windows files. |
| Dynamic Arrays and String Handling | Since string manipulation is primary to the mvBase system, string functions are key to the structure of mvBASIC. File items are read as dynamic array strings in an mvBASIC program, with each line of text separated by an attribute mark. String functions range from locating a substring or specifying a range of characters, to the powerful dynamic array functions for extracting, replacing, deleting, or inserting a specified field in the array. |
| Screen Manipulation | The **@** function in mvBASIC provides a wide range of terminal control sequences. Using these sequences, full formatted screen programs can be produced. |
| Communications | mvBASIC includes several statements designed for communication programs on the mvBase system. Data can be sent to or taken from any remote line. |
| External Subroutines | External subroutines can be executed in mvBASIC with the **CALL** statement. In addition, any TCL command can be executed with the **EXECUTE** statement, and its output and error messages can be captured for use in the program. |
| Dimensioned Array | In mvBASIC, dynamic arrays can be converted into dimensioned arrays, and vice versa. Programmers therefore have the freedom to choose the data form that is most efficient for their applications. |
| Item and Execution Locking | File item locks and execution locks in mvBASIC can prevent multiple users from accessing the same data or executing the same subroutine at the same time. |
| Tape or Floppy Disk I/O | mvBASIC provides statements for directly reading and writing magnetic tapes or floppy disks within the program. This includes Virtual tape drives, and where security permissions allow, Mapped Drives within and external to the Windows environment. |

This guide aims to teach the mvBASIC programming language to a beginning user, assuming that the user is familiar with some programming concepts and techniques and also with the structure of the mvBase system. The reader does not have to be an experienced programmer in order to learn mvBASIC from this guide.

The following reading sequence is recommended:

|  |  |
| --- | --- |
| Creating mvBASIC Programs | Describes how programs are created. It goes into more detail than the average beginner requires, so most readers may skim it for the general concepts and return to it for details as necessary. |
| Format, Data and Expressions | Describes program format, data types, and the syntax for expressions. It should be read carefully. |
| Overview of mvBASIC Statements and Functions | Provides a tour of the mvBASIC language. Every statement and function is covered by topic and should be read very carefully. It is recommended that readers stop periodically, perhaps after every subsection of [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) to try the material just covered. [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) can be used for reference at this stage. |

After studying the sections on Creating mvBASIC Programs,  [Format, Data, and Expressions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/format,_data_and_expressions.htm), and Overview of mvBASIC Statement and Functions, readers should be ready to start writing programs, using the section Statement and Function Reference as a reference.

Readers should refer to [Using the mvBASIC Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_the_mvbasic_debugger.htm) to learn how to debug their programs. This section contains a tutorial and a reference. There are few Debugger commands, so each may be studied and experimented with as the reader proceeds.

|  |  |
| --- | --- |
| Appendix A: Error Messages | Lists error messages which may be encountered while creating programs. |
| Appendix B: List Of ASCII Codes | Provides a list of ASCII codes, which are often necessary for using the CHAR or SEQ functions. |
| Appendix C: mvBASIC Program Examples | Provides several example applications for users to study or copy. |

# Creating mvBASIC Programs

The process of writing an mvBASIC program is relatively simple. The programmer edits the program source code and then compiles the program. If the program compiles successfully, it then executes. If the program runs correctly, it is ready to be cataloged; otherwise, it is debugged (optionally, using the interactive debugger) and the sequence is repeated.

The following topics are presented in this section:

* Creating the Program File
* Editing and Listing the Source Code
* Compiling the Program
* Running the Program
* Cataloging the Program
* A Sample Program

## Creating the Program File

You must store mvBASIC programs in a special file called a program file. Create program files using the **CREATE-BFILE** command for compatibility.

For example, to create a program file called **BP** with a modulo of 3 for the file dictionary and a modulo of 5 for the data file, type:

|  |
| --- |
| >**CREATE-BFILE BP 3 5** |

**CREATE-BFILE** prepares a file to be used for mvBASIC programs by creating a File Definition item in the Master Dictionary with a definition code of **DC** in Attribute 1. **CREATE-BFILE** is otherwise identical to **CREATE-FILE**.

See *User Account Verbs* for further information about the options to **CREATE-BFILE**.

## Editing and Listing the Source Code

The source code of the program is written and edited by the programmer as an item in the program file.

The only restriction to the name of the program is that it not be the same as the name of the file (e.g., item BP in file BP). Such a program will not compile: if it did, the data file pointer in the file dictionary would be overwritten when the program was compiled, and all source code would be lost.

The source code can be created and edited with the Editor or with DocuMentor, a full screen editor. See the Editor User Reference Guide section for more information about the Editor.

### Listing the Program with BLIST

The **BLIST** command produces a formatted listing of the source code item.

Format

|  |
| --- |
| BLIST *filename* *progname-list* [(*options*)] |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of a file that contains mvBASIC programs |
| ***progname-list*** | Contains the item-IDs of the programs to be printed. |

Description

An asterisk (**\***) specifies all programs in the file. **BLIST**, unlike most other commands, sends output to the printer by default. Use the **T** option to **BLIST** to force output to the terminal screen. To send a program **MENU** to the screen, type:

|  |
| --- |
| >**BLIST BP MENU (T)** |

The screen is cleared and the text of **MENU** is printed:

|  |
| --- |
| DD MMM YYYY       BASIC PROGRAM NAME:   MENU                   PAGE    1  0001      LOOP  0002        PRINT "Q FOR QUIT, C FOR CREATE A NEW ENTRY, "  0003        PRINT "E FOR EDIT A ENTRY, D FOR DELETE AN ENTRY"  0004          PRINT  0005          PRINT "ENTER A CHARACTER (Q,C,E OR D)":  0006          INPUT ANSWER,1\_  0007        ON INDEX("QCED",ANSWER,1) GOSUB 100,200,300,400  0008     REPEAT  0009     STOP  0010     SUBROUTINES  0011 100 \*\*\* SUBROUTINE FOR QUITTING \*\*\*  0012     PRINT "--EOJ"  0013     STOP  0014     RETURN  0015 200 \*\*\* SUBROUTINE FOR CREATING A NEW ENTRY \*\*\*  0016     PRINT "CREATING A NEW ENTRY..."              .              .                . |

The program line numbers are shown, and the text prints with an indent of 5 spaces. The text between **THEN** or **ELSE** clauses and corresponding **END** statements are indented another 3 spaces, as is the text between program loops. Note that alphanumeric statement labels are also indented, although numeric statement labels are not.

The **D** option to **BLIST** specifies that the lines should be double-spaced. In addition, a range of lines in the program can be specified.

The default indents used by **BLIST** are changed by editing line 4 of the Verb Definition item in the Master Dictionary.

## Compiling the Program

Source code must be compiled before the program can be executed. The source code can be written and edited by the programmer, but since it cannot be directly interpreted by the mvBase system until it is translated into object code, the compiler translates source code into object code and places a pointer to the object code in the file dictionary.

The compiler can therefore be thought of as a translator from your language (or more accurately, the language of mvBASIC) into the machine’s language.

Two (synonymous) commands can be used to compile a program, **BASIC** and **COMPILE**.

Format

|  |
| --- |
| BASIC *filename* *progname-list* [(*version#*,*options*)] |

Format

|  |
| --- |
| COMPILE *filename* *progname-list* [(*version#*,*options*)] |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the mvBASIC program file |
| ***progname-list*** | Contains the item-IDs of the programs to be compiled. An asterisk (**\***) specifies all programs in the file. |

For example, to compile the program **ADDNUMS** in the file **BP**, type:

|  |
| --- |
| >**COMPILE BP ADDNUMS** |

or:

|  |
| --- |
| >**BASIC BP ADDNUMS** |

If the compile is successful, the user sees something similar to:

|  |
| --- |
| >**BASIC BP ADDNUMS**  \*\*\*\*\*\*\*\*\*  SUCCESSFUL COMPILE!  1 FRAMES USED.    > |

The asterisks (**\***) each represent a source line successfully compiled into object code. If an error occurs in compilation, the error code is printed with a message. See [Appendix A: Error Messages](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/appendix_a_error_messages.htm) for a list of error messages generated by the **COMPILE** command.

The following topics are presented in this section:

* Option for Debugging a Program (X)
* Options for Cataloging (C, S)
* Listing the Source (L)
* Printing Compiler Output (P)
* Specifying a Version of the Operating System
* Obtaining the Compilation Time and Date

### Option for Debugging a Program (X)

The **X** option supplies information that the programmer may use to debug a program.

#### Creating the Cross-reference (X)

The **X** option to **COMPILE** creates a cross-reference of all variables and labels used in the program and places it in the BSYM file in the current account. For example:

|  |
| --- |
| >**COMPILE BP ADDNUMS (X)**  \*\*\*\*\*\*\*\*  SUCCESSFUL COMPILE!   1 FRAMES USED.    > |

The BSYM file now has 3 items with item-IDs: **SUM**, **NUM1**, and **NUM2**. Each item has a single attribute, with value marks separating the line numbers at which the variable was accessed. For example, item **SUM** contains in Attribute 1:

|  |
| --- |
| 005\*]006 |

From this we can tell that the **SUM** variable is accessed in lines 5 and 6 of the program. The asterisk (**\***) after 005 signifies that the variable is assigned a value on line 5 of the program. Similarly, item **NUM1** of the file BSYM contains:

|  |
| --- |
| 002\*]005]006 |

The data section of the BSYM file is initialized each time the **X** option to **COMPILE** is used, regardless of whether the same program is compiled or what program file the program resides in. If the BSYM file does not already exist, an error message results and the cross-reference is not created.

### Options for Cataloging (C, S)

When a programmer has determined that no more changes will be made to the program, the **C** or **S** options compact the object code or suppress the symbol table.

#### Suppressing the EOLs

As seen in the example of output from **COMPILE** with the A option, the last opcode on each line is EOL, for End-Of-Line. When a program is fully debugged, a programmer might choose to compile it with the **C** option to suppress the EOL opcodes, since they are necessary only for debugging purposes. Thus the programmer can reduce the size of the object code by 1 byte for each line of source code. In the 8-line program ADDNUMS, only 8 bytes would be saved, but the **C** option could make a significant difference for a larger program.

The EOL opcodes, however, are used to count lines for error messages and for the interactive debugger. By using the **C** option, further debugging of the program becomes exceedingly difficult. If errors do occur in a program compiled with the **C** option, all errors will report on line 1, and the interactive debugger will be largely inoperable.

#### Suppressing Messages and the Symbol Table (S)

The compiler creates a symbol table along with the object code. The symbol table, along with any messages generated by the compiler, can be suppressed with the **S** option to **COMPILE**.

The symbol table is necessary for the mvBASIC interactive debugger, which is largely inoperable without it. However, a programmer can choose to suppress the symbol table when the program is fully debugged and ready for use.

See [Using the mvBASIC Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_the_mvbasic_debugger.htm) for further discussion on this subject.

### Printing Compiler Output (P)

The **P** option sends all compiler output to the printer. This option is particularly useful with the **L** option.

### Specifying a Version of the Operating System

The **COMPILE** command includes an option for specifying what version of the operating system the program is meant to run on. It can be one of the following numbers:

|  |  |
| --- | --- |
| 22 | Specifies Mentor O/S release 2.2. |
| 24 | Specifies Mentor O/S and PC/OS release 2.4. |
| 25 | Specifies Mentor O/S release 2.5. |
| 26 | Specifies Mentor O/S release 2.6. |
| 27 | Specifies Mentor O/S release 2.7. |
| 28 | Specifies Mentor O/S and O/E release 2.8. |
| 30 | Specifies Mentor O/S, PC/OS, and O/E release 3.0. |

Although the program compiles for a different version of the operating system, it will not run unless it matches the current operating system release. For example, the **ADDNUMS** program can be compiled for version 2.7 of the operating system with the following command:

|  |
| --- |
| >**COMPILE BP ADDNUMS (27)**  \*\*\*\*\*\*\*\*  SUCCESSFUL COMPILE!   1 FRAMES USED.    > |

However, if a user attempts to run the program on a different version of the operating system, the program does not run and an error message is printed:

|  |
| --- |
| >**RUN BP ADDNUMS**    [B97] VERSION NUMBER OF COMPILED PROGRAM 'ADDNUMS' IS DIFFERENT FROM THAT OF OPERATING SYSTEM.    > |

### Obtaining the Compilation Time and Date

The time and date an mvBASIC program was compiled may be obtained using the following:

Format

|  |
| --- |
| LIST [DICT] *filename* [*item-list*] [*selection*]  *COMP.TIME* [*output*] [*print*] [*modifiers*] [(*options*)] |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the mvBASIC program file. |
| ***item-list*** | Item-ID of the program for which you want to obtain the time and date of compilation. If ***item-list*** is not specified, the time and date will be listed for all programs in the file. |

For example, the time and date **TEST1** was compiled may be obtained using the following:

|  |
| --- |
| >**LIST DICT BP TEST1 COMP.TIME**  BP...........................TIME OF COMPILATION  TEST1   08:11:12  02 MAR 1992  1 ITEM LISTED.  > |

|  |  |
| --- | --- |
| **NOTE** | **COMP.TIME** is a B-correlative in NEWAC which is copied to the master dictionary of new accounts. |

## Running the Program

The **RUN** command executes a compiled mvBASIC program.

Format

|  |
| --- |
| RUN *filename* *progname* [(*options*)] |

Parameter(s)

|  |  |  |
| --- | --- | --- |
| ***filename*** | Name of the mvBASIC program file | |
| ***progname*** | Item-ID of the program that you wish to run. (Multiple programs cannot be listed on the syntax line for **RUN**.) | |
| ***options*** | The following options are supported: | |
| **A** | Prevents a fatal error from invoking the Debugger by forcing an abort of the program instead. However, the BREAK key can still be used if the **A** option is specified. |
| **D** | The mvBASIC interactive Debugger is entered during execution by pressing the BREAK key. Alternatively, the program can be run with the **D** option, which forces the program to enter the Debugger before executing line 1. |
| **E** | The Debugger is entered when a fatal run-time error is encountered. If RUN is used with the **E** option, non-fatal errors will invoke the Debugger as well. See [Appendix A: Error Messages](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/appendix_a_error_messages.htm) for a list of error messages generated by the **RUN** command. |
| **I** | Inhibits the initialization of the data area when a new program is called with the **CHAIN** statement. When the **EXECUTE** statement was incorporated into mvBASIC, however, the **CHAIN** statement became largely obsolete. The I option is included for compatibility with older code, but its use is not recommended and will create problems if the workspace area is at all corrupted. |
| **N** | If a **HEADING** statement has been specified in a program, the program waits for a carriage return by the user after each page of output to the terminal. The **N** option suppresses this feature. This is equivalent to specifying the N option to the **HEADING** statement within the program source. |
| **P** | Sends output generated by **PRINT**, **HEADING**, and **FOOTING** statements to the printer instead of the screen, and is equivalent to placing a **PRINTER ON** statement in the beginning of the program. |
| **Q** | Eliminates checking the Master Dictionary for a cataloged subroutine, and assumes the object code of the subroutine resides in the same file as the object code of the main program. |
| **S** | The **S** option suppresses error messages generated by the **RUN** command. |

For example, to execute the compiled program **MENU** in file **BP**, type:

|  |
| --- |
| >**RUN BP MENU** |

## Cataloging the Program

The **CATALOG** command creates a Verb Definition item in the Master Dictionary of the user’s account by creating a direct pointer to the object code. By using **CATALOG**, a program can be executed directly through TCL, without using the **RUN** command. Once cataloged, the program can be recompiled without having to be recataloged.

Format

|  |
| --- |
| CATALOG *filename* *progname-list* |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the mvBASIC program file. |
| ***progname-list*** | List of the item-IDs of the programs to be cataloged. An asterisk (**\***) specifies all programs in the file. |

For example, to catalog the program **ADDNUMS** in the file **BP**, use the following command:

|  |
| --- |
| >**CATALOG BP ADDNUMS**  [244] 'ADDNUMS' CATALOGED!    > |

The program can then be accessed as if it were a command:

|  |
| --- |
| >**ADDNUMS**  ENTER ONE NUMBER?**3**  ENTER ANOTHER NUMBER?**4**  THE SUM OF 3 AND 4 IS 7    > |

The program may fail to be cataloged because the object code cannot be found or because there already exists an item in the Master Dictionary with the same name.

The **DECATALOG** command deletes the Verb Definition item from the user’s account.

Format

|  |
| --- |
| DECATALOG *filename* *progname-list* |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the mvBASIC program file. |
| ***progname-list*** | List of the item-IDs of the programs to be decataloged. An asterisk (**\***) specifies all programs in the file. |

Example

|  |
| --- |
| >**DECATALOG BP ADDNUMS**  [242] ’ADDNUMS’ DECATALOGED.    > |

The **DECATALOG** command not only deletes the Verb Definition item, it also deletes the pointer to the object code in the file dictionary. The program must be recompiled after it has been decataloged, before it can be executed again with **RUN**.

## A Sample Program

As an example, you can create a simple program called **ADDNUMS**; it returns the sum of two numbers.

Before you write the program, you need a program file. The source code for an mvBASIC program is entered as an item in the program file. By convention, the program file for an account is called **BP** (for Basic Programs). The program file is created with the **CREATE-BFILE** command.

|  |
| --- |
| >**CREATE-BFILE BP 1 3**  [417] FILE 'BP' CREATED; BASE = 10999, MODULO = 1.  [417] FILE 'BP' CREATED; BASE = 11000, MODULO = 3.  > |

The command in the preceding example creates a program file, with a modulo of 1 for the file dictionary and a modulo of 3 for the data file. The new source code can now be placed in item **ADDNUMS** in the program file BP. For writing the program, use the Editor:

|  |
| --- |
| >**ED BP ADDNUMS**  NEW ITEM  TOP  .I  001+**PRINT "ENTER ONE NUMBER":**  002+**INPUT NUM1**  003+**PRINT "ENTER ANOTHER NUMBER":**  004+**INPUT NUM2**  005+**2INT\_SUM = NUM1 + NUM2**  006+**PRINT "THE SUM OF ":NUM1 : " AND ":NUM2:" IS ":2INT\_SUM**  007+**STOP**  008+**END**  009+ **<RETURN>**  TOP  .FI  'ADDNUMS' FILED.    > |

Compile the program with the **COMPILE** command, translating the program’s source code into object code.

|  |
| --- |
| >**COMPILE BP ADDNUMS**  \*\*\*\*\*\*\*\*  SUCCESSFUL COMPILE!   1 FRAMES USED.    > |

The program compiles without error. Once the program is compiled, execute it with the **RUN** command.

|  |
| --- |
| >**RUN BP ADDNUMS**  ENTER ONE NUMBER?**4**  ENTER ANOTHER NUMBER?**9**  THE SUM OF 4 AND 9 IS 13    > |

The program runs successfully on the first attempt. You can now catalog it, so that you can use it as if it were a command:

|  |
| --- |
| >**CATALOG BP ADDNUMS**  [244] 'ADDNUMS' CATALOGED!    >ADDNUMS  ENTER ONE NUMBER?**5**  ENTER ANOTHER NUMBER?**3**  THE SUM OF 5 AND 3 IS 8    > |

You have created an mvBASIC program. More complex programs seldom compile the first time, and once they compile they don’t always run without error. However, the process remains unchanged: edit, compile, and run the program until it runs successfully.

As the training progresses, the following items are highlighted for initial importance:

Definition and examples of Variables

READ statement

CASE statement

IF statement

MATREAD command

OPEN Statement

These items are covered in detail on the following pages

## Variables

### Assigning and Using Variables

Variables are symbolic names that represent stored data values and can change in value during program execution. The value can be explicitly assigned by the programmer, can be read as input, or can be the result of operations performed by the program during execution.

At the start of program execution, all variables are set to an unassigned state. Any attempt to use a variable in the unassigned state produces an error message, **and a value of 0** is assumed.

Names for both variables and constants must begin with **an initial alphabetic character**. They can also include one or more digits, letters, periods, or dollar signs. (Note that hyphens and underscores are not valid in a variable name.) Uppercase and lowercase are interpreted differently. A variable name can be any length, but it cannot be the same as any reserved word.

### Data Typing in mvBASIC

In many other programming languages, such as Pascal and PL/I, a distinction is made among types of data. In these languages, all constants, variables, and their data types (integer, real, string, character, etc.) have to be declared at the beginning of the program so that the compiler will know how to store the data. Furthermore, the size of the variable often has to be declared so that the compiler will know how much space to set aside.

In mvBASIC, on the other hand, **no data typing is made by the compiler**: all data typing is made at run time, by context. A variable can therefore alternate between numeric and string values within the program. The only thing to be careful of is that when string values are assigned in the program text, they must be delimited by single quotes (**'**), double quotes (**"**), or backslashes (**\**). Otherwise, they are assumed to be variable names.

There is, of course, a difference between the way a numeric value and a string value can be treated: it is unreasonable to expect a program to take the square root of the string **CARL**. In such a situation, however, a fatal error will not occur—when a string value is applied to a numeric function, a value of 0 is assumed, a warning message is printed, and the program continues from there. String operations, on the other hand, can be executed on numeric values without conflict.

The advantage to no data typing is obvious; less work for the programmer and more flexibility for the program. The disadvantage is that errors which one might expect the compiler to detect are not caught. For example, if a variable name is misspelled, the compiler will simply assume that it is a new variable, and the program will successfully compile without an error or warning. Similarly, if a string variable containing **CARL** were accidentally used in the **SQRT** function, the programmer would not find out until the program was executed.

## Advanced Variables

Thus far we have discussed simple numeric and string data only. There are other types of data in mvBASIC, however, which are assigned with special syntax.

The following topics are presented in this section:

[Array Variables](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/array_variables.htm)

[Dimensioned Arrays](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dimensioned_arrays.htm)

[File Variables](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/file_variables.htm)

[Select-list Variables](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/select-list_variables.htm)

### Array Variables

An array variable is a variable that represents more than one data value. There are two types of array; dynamic arrays and dimensioned arrays.

Dynamic Arrays

A dynamic array is a mapping of the structure of file items to string data. Any string, however, can be considered a dynamic array.

A dynamic array is a string containing substrings that are separated by special delimiter characters. At the highest level, these elements are called attributes, and are separated by attribute marks (CTRL+^). Each attribute can contain values separated by value marks (CTRL+]). Each value can contain subvalues separated by subvalue marks (CTRL+\). Thus, an example of a dynamic array is as follows:

PETER THOMPSON]333-8989\232-8665^JOEFRIDAY]872-1789\865-0096

In this dynamic array string, there are two attributes:

PETER THOMPSON]333-8989\232-8665

JOE FRIDAY]872-1789\865-0096

there are four values:

PETER THOMPSON

333-8989\232-8665

JOE FRIDAY

872-1789\865-0096

and there are four subvalues:

333-8989

232-8665

872-1789

865-0096

The primary use of dynamic arrays is to store data that is either read from or written to a file item. Each line in a file item corresponds to a separate attribute. However, mvBASIC includes facilities for manipulating dynamic array elements that make dynamic arrays a powerful data type for processing information independently of file items.

Dynamic arrays are called arrays because they can be referenced by array functions using 3 subscripts, and they are called dynamic because elements can be added or deleted without having to recompile the program. Null attributes, values, and subvalues are represented by two consecutive attribute marks, value marks, or subvalue marks, respectively.

See [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) for more information on processing dynamic arrays.

### Dimensioned Arrays

Dimensioned arrays (also called standard arrays) are one- or two- dimensioned structures. Each value in a standard array is called an element of the array.

A one-dimensioned array (also called a vector) has its elements arranged in sequence. An element of a vector is specified by the variable name, followed by the index of the element enclosed in parentheses. The index of the first element is (1).

A two-dimensioned array (also called a matrix) has the elements of the first row arranged sequentially in memory, followed by the elements of the second row, and so on. An element of a matrix is specified by the variable name, followed by two indexes enclosed in parentheses, representing the row and column position of the element. The indexes of the first element are (1,1).

The indexes used to specify the elements of a matrix that has four columns and three rows are illustrated by the following:

COST:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Col1 | Col2 | Col3 | Col4 |
| Row1 | **COST(1,1)** | **COST(1,2)** | **COST(1,3)** | **COST(1,4)** |
| Row2 | **COST(2,1)** | **COST(2,2)** | **COST(2,3)** | **COST(2,4)** |
| Row3 | **COST(3,1)** | **COST(3,2)** | **COST(3,3)** | **COST(3,4)** |

Note that vectors, or one-dimensioned arrays, are treated as matrixes with a second dimension of 1. **COST(3)** and **COST(3,1)** are equivalent specifications and can be used interchangeably.

Indexes can be written as constants or as expressions.

Before a dimensioned array can be used in an mvBASIC program, a **DIM** or **COMMON** statement must be used to declare the maximum number of elements it will store throughout the program. See the reference pages on **DIM** and **COMMON** for more information.

### File Variables

A file variable is created by a form of the **OPEN** statement. Once opened, a file variable is used in I/O statements to access the file.

See [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) for more information on assigning and using file variables.

### Select-list Variables

A select-list is a set of item-IDs or attributes created by the **SELECT** statement or by TCL select-list generators, to be used in a **READNEXT** statement. There are three types of select-list:

|  |  |
| --- | --- |
| External file select-lists | External file select-lists are created by TCL list generators such as **SELECT**, **SSELECT**, and **QSELECT**, external to the mvBASIC program. |
| Internal file select-lists | Internal File select-lists are created by using the **SELECT** statement on an mvBASIC file variable. |
| Dynamic array select-lists | Dynamic array select-lists are created by using the **SELECT** statement on an mvBASIC string variable. Such strings are stored in a dynamic array. |

Refer to the section titled [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) for more information on select-list variables.

## Reading and Writing Tapes or Floppy Disks

mvBASIC includes several statements for tape and floppy disk processing. For the purpose of this discussion a floppy disk functions as a tape and therefore, is included in any reference to a tape. For reading and writing strings on tape, there are the **READT** and **WRITET** statements. As expected, the **READT** statement reads the next record off the attached tape device, and the **WRITET** statement writes a record onto the tape. The **READTX** statement is designed for tapes which might contain segment marks. The **READTX** statement is identical to the **READT** statement, except that the data from the tape is translated into ASCII hexadecimal format before it is assigned to the string. The **ICONV** function can then be used to translate the string back into readable format. **READTX** is designed for reading segment marks (**CHAR(255)**) from a tape.

In addition, there are statements to simulate the **T-WEOF** and **T-REW** commands. The **WEOF** statement writes an End-Of-File mark at the current position of the tape, and the **REWIND** statement rewinds the tape to the beginning.

Each of the tape I/O statements includes **THEN** and **ELSE** clauses to specify action according to whether the tape statement was successful. The **ELS** clause is often used to produce a meaningful error message by calling the **SYSTEM(0)** function. The **SYSTEM(0)** function returns a number from 0 to 4, reflecting whether the latest tape I/O statement worked, and if it didn’t, what the problem was. See [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) for more information.

## READ Statement

The **READ** statement assigns the string value of a file item to a variable.

Format

|  |
| --- |
| READ *var* FROM [*filevar*,] *item-ID* [THEN *statements1*] [ELSE *statements2*] |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Assigns ***var*** to the string value of the file item, in dynamic array form. |
| **FROM [*filevar*,] *item-ID*** | ***Item-ID*** is an expression evaluating to an item-ID. Assign ***var*** the string value of ***item-ID*** in the file which was opened as ***filevar***. If ***filevar*** is not specified, the default file variable is used, which is the file most recently opened without a file variable assignment. If the specified ***item-ID*** does not exist, ***var*** is assigned the value of the null string (**""**). |
| **THEN *statements1*** | Executes ***statements1*** if ***item-ID*** is read successfully. |
| **ELSE *statements2*** | Executes ***statements2*** if ***item-ID*** cannot be read. |

Description

Before a file can be accessed in a **READ** statement, it must be opened with an **OPEN** statement or an error will occur at runtime. See [OPEN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/open_statement.htm) for more information.

In mvBASIC there are also **READU**, **READV**, and **READVU** statements available. The **READU** statement sets an item update lock on the file item before reading it, the **READV** statement reads a single attribute from a given file item, and the **READVU** statement sets a item update lock and then performs a **READV**. See [READU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readu_statement.htm), [READV Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readv_statement.htm) and [READVU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readvu_statement.htm) for more information.

Example

In this application, the **OPEN** statement is used to open a reservation file and the operator is asked to enter the customer's last name, to be used as an item-ID. If the reservation file is not found, the program aborts. A **READ** statement is then used to find the file item. If the item is found, any current reservations are shown; if it is not found, a new reservation may be entered.

|  |
| --- |
| OPEN "RESERVATIONS" TO RES.FILE ELSE     ABORT 201,"RESERVATIONS"  END     .     .     .  LOOP     PRINT "LAST NAME :  " :     INPUT ITEM.ID     READ RECORD FROM RES.FILE,ITEM.ID THEN        PRINT ITEM.ID : " ON FILE."        GOSUB SHOW.RES     END ELSE        PRINT ITEM.ID : " NOT ON FILE"        GOSUB ENTER.RES     END  UNTIL LAST.NAME = "" DO REPEAT |

## UREADLINE Function

The mvBASIC command clause, **UREADLINE**, reads data from a previously opened file on the host system. A file on the host system is represented as an unformatted string of bytes without internal delimiters or markers. Read and write commands provide sequential access to files by advancing a pointer within the file. Subsequent read or write commands advance this internal pointer from the current position in the file. The user may specify the exact location within a file to be read from by positioning the file pointer before executing the command.

Format

|  |
| --- |
| UREADLINE *var* FROM *FileHandle* {UNTIL *delimiter*} THEN *statement(s)* ELSE *statement(s)* |

Description

**UREADLINE** reads the file currently opened to the file variable, ***FileHandle***, starting at the current file pointer position up to but not including the delimiter or until an EOF mark is reached. The delimiter can be any single character and if not specified, the character line-feed (0x0a) is used. The resulting string value is assigned to the variable ***var***. If an error occurs, the **ELSE** clause executes and **UERROR()** returns the appropriate error code. The file pointer is advanced by the number of bytes read.

|  |  |
| --- | --- |
| **NOTE** | The file must be opened to be read. |

If the **UREADLINE** command is used before opening the file for reading, an error results. The **ELSE** clause executes, the ***var*** is not changed and **UERROR()** returns the appropriate error code.

If any data is successfully read, then the **THEN** clause is taken. The ***var*** contains the data read and the value returned by **UERROR()** is the number of bytes read.

If no data is successfully read due to an EOF being encountered immediately, the **ELSE** clause is taken. The ***var*** is set to **NULL** and **UERROR()** returns the Windows error message 38 (**ERROR\_HANDLE\_EOF**).

If errors other than EOF occur during a **UREADLINE** the **ELSE** clause is taken, the ***var*** contains the data read up to the error and the **UERROR()** returns an appropriate error code.

For example, the statement below generates the behavior as shown in the table. The File Contents describes the content of the host file for the example. EOF indicates the actual End of File, ***var*** is the value returned. **THEN/ELSE** indicates which of the **THEN** or **ELSE** clauses is taken and **UERROR()** indicates what the value returned by an immediate call to the **UERROR()** function would return.

|  |
| --- |
| UREADLINE *var* FROM *filehandle* UNTIL ';' THEN/ELSE clause |

|  |  |  |  |
| --- | --- | --- | --- |
| **HOST FILE CONTENTS** | **var** | **THEN/ELSE** | **UERROR( )** |
| EOF | "" | ELSE | 38 |
| ";EOF" | ""  "" | THEN  ELSE | 0  38 |
| ";;EOF" | ""  ""  "" | THEN  THEN  ELSE | 0  0  38 |
| "ABCEOF" | "ABC"  "" | THEN  ELSE | 3  38 |
| "ABC;EOF" | "ABC"  "" | THEN  ELSE | 3  38 |
| "ABC;;EOF" | "ABC"  ""  "" | THEN  THEN  ELSE | 3  0  38 |
| "ABC;XYZEOF" | "ABC"  "XYZ"  "" | THEN  THEN  ELSE | 3  3  38 |
| "ABC;XYZ;;EOF" | ABC"  "XYZ"  "" | THEN  THEN  ELSE | 3  3  38 |
| ";ABC;;XYZ;;EOF" | ""  "ABC"  ""  "XYZ"  ""  "" | THEN  THEN  THEN  THEN  THEN  ELSE | 0  3  0  3  0  38 |

This example prints the first line (delimited by a line-feed) of the file \books\chap5.txt. A null string is returned if the file pointer is positioned at the end of the file.

|  |
| --- |
| FILENAME="c:\books\chap5.txt"  LF = CHAR(10)  UOPEN FILENAME FOR READ TO FILEDES ELSE     PRINT "Unable to open ":FILENAME     STOP  END  UREADLINE VAR1 FROM FILEDES UNTIL LF ELSE GOTO   EOJ:  END  PRINT VAR1 |

## UREAD Command

The **UREAD** command reads data from a previously opened file on the host system. A file on the host system is represented as an unformatted string of bytes without internal subdividers or markers. Read and write commands provide sequential access to files by advancing a pointer within the file. Subsequent read or write commands advance this internal pointer from the current position in the file. The user may specify the exact location within a file to be read from by positioning the file pointer before executing the command. See [ULSEEK Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ulseek_function.htm) for additional information.

Format

|  |
| --- |
| UREAD *var* FROM *FileHandle* FOR *n* THEN *statement(s)* ELSE *statement(s)* |

Description

**UREAD** reads the file currently opened to the file variable ***FileHandle*** for the number of bytes specified by n or until an EOF mark is reached. ***n*** indicates the number of contiguous bytes from the current file pointer position to be read. The resulting string value is assigned to the variable ***var***. If an error occurs, the **ELSE** clause executes and **UERROR( )** returns the appropriate error code. The file pointer is advanced by the number of bytes read.

|  |  |
| --- | --- |
| **NOTE** | The file must be opened to be read. |

* If the **UREAD** command is used before opening the file for reading, an error results, the **ELSE** clause executes, and **UERROR( )** returns the appropriate error code.
* If any data is successfully read, the **ELSE** clause is not taken. In that case, **UERROR( )** returns the number of bytes read.
* If an EOF is encountered, the value returned by **UERROR( )** is less than the requested amount.
* If no EOF is encountered, the value returned by **UERROR( )** is equal to the requested amount.
* If no data is successfully read due to an EOF being encountered immediately, the **ELSE** clause is taken, no data is returned and **UERROR( )** returns the Windows error message 38: **ERROR\_HANDLE\_EOF**.
* If errors other than EOF occur during a **UREAD** the **ELSE** clause is taken, no data is returned, **UERROR( )** returns the appropriate error code.

Examples

|  |
| --- |
| UOPEN "C:\TEST.TXT" TO HANDLE ELSE STOP  10 UREAD ITEM FROM HANDLE FOR 500 THEN       PRINT UERROR( )    END ELSE       PRINT "ERROR CODE = ": UERROR( )       GOTO EOJ:    END    GOTO 10  If TEST.TXT is 943 bytes the output is:    500    443    ERROR CODE = 38 |

The next example prints the first 50 bytes of the file \books\chap5.txt. If less than 50 bytes are present, the bytes available are printed. A null string is returned if the file pointer is positioned at the end of the file. Line feed characters embedded in the file are also printed.

|  |
| --- |
| FILENAME="c:\books\chap5.txt"    UOPEN FILENAME FOR READ TO FILEDES ELSE       PRINT "Unable to open ":FILENAME       STOP    END    UREAD VAR1 FROM FILEDES FOR 50 ELSE GOTO EOJ:    PRINT VAR1 |

After execution of the following example, the string **TESTDATA2** is equal to **TESTDATA1**. \books\chap6.txt was created using the **UCREATE** statement which opened the file for writing only.

|  |
| --- |
| TESTDATA1 = "THIS IS TEST DATA"  FILENAME="c:\books\chap6.txt"    UCREATE FILENAME TO FILEDES2 ELSE       PRINT "Unable to create and open ":FILENAME       STOP    END    \* Write out DATA and CLOSE    UWRITE TESTDATA1 ON FILEDES2 ELSE       PRINT "Write to ":FILENAME:" failed"    END    UCLOSE FILEDES2 ELSE       PRINT "UCLOSE failed on ":FILENAME    END    \*    UOPEN FILENAME TO FILEDES3 ELSE STOP    UREAD TESTDATA2 FROM FILEDES3 FOR 17 ELSE GO TO EOJ: |

## Compiler Directives

There are four compiler directive statements. Each of these statements begins with a dollar sign (**$**).

### Comments in the Object Code

The **$\*** statement places a comment directly in the object code of a program when it is compiled. It is most useful for entering version numbers or copyright information before software is distributed.

### Reading External Source Code ($CHAIN, $INCLUDE, $INSERT)

Three statements tell the compiler to read source code from another file item: **$INCLUDE**, **$INSERT**, and **$CHAIN**.

**$INCLUDE** and **$INSERT** are identical statements. Either statement results in the program being compiled as if the external source code were written at the point where the **$INCLUDE** or **$INSERT** statement had been entered. Compilation will then continue at the line after the **$INCLUDE** or **$INSERT** statement. **$INCLUDE** and **$INSERT** are also useful for any code that might be used several different programs. An example of such code might be a file item containing **COMMON** statements.

The **$CHAIN** statement is different from **$INCLUDE** and **$INSERT** in that the compilation will not return to the original program. The **$CHAIN** statement is not intended for code which might be shared by several programs, but for programs which may have source code longer than 32K bytes. The **$CHAIN** statement allows several different file items containing source code to be **CHAIN**ed together.

The only restriction to **$INCLUDE**, **$INSERT**, and **$CHAIN** is that the number of bytes in the resulting object code cannot exceed 248K.

## Reading and Updating File Items

Before an item in a file can be accessed, it must be assigned a symbolic name, called a file variable. The file variable is necessary to provide a pointer to the file that will be used by the program each time the file is accessed.

### File Variables (OPEN)

The **OPEN** statement assigns a file variable to a file, so that the program can read, write, select, or delete items in the file. All subsequent access of the file must reference the file variable and not the file name itself.

If a file is opened without a file variable specified, it uses the default file variable. Any subsequent file access statements that do not specify a file variable will use the default file variable. Only one file can be assigned to the default file variable at a single time.

### Reading and Writing a File Item (READ, WRITE, etc.)

Once the file is opened, any item can be directly accessed. The **READ** statement assigns the string value of a file item to a dynamic array variable. The fields of the array can then be accessed by the dynamic array processing functions **EXTRACT**, **REPLACE**, **INSERT**, and **DELETE**. The **WRITE** statement writes a new or updated dynamic array string into a file item. There are several variations to **READ** and **WRITE** provided by mvBASIC. The **READV** and **WRITEV** statements read and write only a single attribute of an item, as a shortcut for programs which are concerned only with a single attribute. In addition, the **MATREAD** and **MATWRITE** statements read and write items as dimensioned arrays, with each attribute corresponding to an element of the array.

The file-reading statements are each equipped with **THEN** and **ELSE** clauses. If the item cannot be found, the v statements are executed. If it can be found, the **THEN** statements are executed. See [Internal Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/internal_program_control.htm) for more information on the syntax of **THEN** and **ELSE** clauses.

### File Item Locks (READU, WRITEU, RELEASE, etc.)

Each of the statements for reading a file item have corresponding statements that place a lock on the file item as it is read. These statements are the **READU**, **READVU**, and **MATREADU** statements. (The **U** suffix stands for Update, declaring that the file item might be changed and rewritten.) The item lock is lifted either when the item is released with a **RELEASE** statement, deleted with a **DELETE** statement, written with a **WRITE**, **WRITEV**, or **MATWRITE** statement, or when the program is terminated. Until the lock is lifted, no other users will be able to access the same file item with a **READU**, **READVU**, or **MATREADU** statement.

File item locks only affect other **READU**, **READVU**, and **MATREADU** statements. While an item is locked, programs can access the file item with a normal **READ**, **READV**, or **MATREAD** statement, or they can even write it with any of the file writing statements. The success of a file item lock depends on its being respected by all other programs that access the same file.

If an item is to be written but the programmer does not want the lock removed, the **WRITEU**, **WRITEVU**, and **MATWRITEU** statements should be used in place of **WRITE**, **WRITEV** or **MATWRITE**. These statements will write the file item but retain the item lock for subsequent update. (Again, the U suffix stands for Update, declaring that further update might occur.)

### The LOCKED Clause

The item-locking statements **READU**, **READVU**, and **MATREADU** are each equipped with an optional **LOCKED** clause. Normally, when a program attempts to read and lock an item which is already locked, the program waits for the item to be released before continuing with execution. However, if the **LOCKED** clause is included, the program simply executes the **LOCKED** statements and continues with execution immediately. The **LOCKED** statements follow the syntax of **THEN** and **ELSE** clauses in mvBASIC.

The **LOCKED** clause helps to avoid the situation called a deadly embrace. A deadly embrace happens when two users both lock items, and before releasing their locks, each user then tries to read and lock the other item. Without the **LOCKED** clause, both users will be indefinitely stuck since neither is free to unlock its item. If the **LOCKED** clause is used, however, the deadly embrace cannot occur.

### Select-lists (SELECT, READNEXT)

Select-list variables can be created through the mvBASIC **SELECT** statement, or by using the **EXECUTE** statement to call one of the INFO/ACCESS select-list generators. The **SELECT** statement does not accept the selection expressions accepted by the INFO/ACCESS commands; however, the **SELECT** statement does allow a select-list to be created from the attributes of a dynamic array string. See [External Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/external_program_control.htm) for more information on using **EXECUTE** for generating select-lists.

A select-list can also be created external to the program by executing one of the INFO/ACCESS select-list generators and then immediately running the program. If the program is designed this way, the **SYSTEM(11)** function is recommended to test if there is an external select-list.

Once the select-list is created, it can be read with the **READNEXT** statement. **READNEXT** reads the next item- ID in the select-list. After selecting a file, the **READNEXT** statement is generally used in a loop to perform a procedure on all selected items.

### Deleting File Items (DELETE, CLEARFILE)

The **DELETE** statement is a statement that deletes a specific file item from an opened file. It should not be confused with the **DELETE** function or the **DEL** statement, which both delete a field from a dynamic array.

The **CLEARFILE** statement deletes all items in the data file.

## READT Statement

The **READT** statement reads the next record (block) on the magnetic tape or floppy disk unit, assigning its value to the specified variable.

Format

|  |
| --- |
| READT *var* [RETURNING *var*]  [THEN  *statements*  END] [ELSE  *statements*  END] |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Variable into which the next record is read, in dynamic array form. |
| **RETURNING *var*** | If a tape or floppy disk error occurs while using the **RETURNING** clause, the tape or floppy disk error is assigned to the returning variable and program control continues through the **ELSE** clause. |
| **THEN *statements*** | Executes ***statements*** if record is successfully read. |
| **ELSE *statements*** | Executes ***statements*** if record cannot be read. |

Description

The **READT** statement may be used to read a record from an attached tape or floppy disk unit. If a record is read, its value is assigned to the specified variable and the **THEN** statements are executed. If the record cannot be read, the **ELSE** statements are executed, and the value of ***var1*** does not change.

A record might not be read because the tape or floppy disk has not been attached, or because an End-Of-File mark was encountered. To determine why a tape or floppy disk could not be read, the **SYSTEM** function is often used in the **ELSE** portion of a **READT** statement. See [SYSTEM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/system_function.htm) for more information.

**READT** also has an optional **UNLABELED** clause to allow 1/2-inch tapes to read a tape without reading a label.

Format

|  |
| --- |
| READT *var* [UNLABELED] [RETURNING *var*]  [THEN  *statements*  END] ELSE  Statements  END |

To read tapes or floppy disks in ASCII hexadecimal format, use the **READTX** statement.

See [READTX Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readtx_statement.htm) for more information.

Example

The program segment in this example reads data off a tape and prints them in a readable format. The item-IDs are printed and then each attribute is printed on a separate line, preceded by the attribute number.

|  |
| --- |
| LOOP     READT NEWRECORD ELSE        IF SYSTEM(0) = 2 THEN           END.OF.TAPE = TRUE        END ELSE           PRINT "SYSTEM ERROR --"           GOSUB EXIT        END     END     REC.NUM += 1  UNTIL END.OF.TAPE DO     PRINT     PRINT "PRESS ANY KEY TO READ RECORD " : REC.NUM :" : " :     INPUT CHAR,1     PRINT     NO.OF.ATTRS = DCOUNT(NEWRECORD , AM)     FOR I = 1 TO NO.OF.ATTRS        PRINT I, NEWRECORD< I >     NEXT I  REPEAT |

## READNEXT Statement

The **READNEXT** statement reads the next sequential value in a select-list.

Format

|  |
| --- |
| READNEXT *var1* [, *var2*] [FROM *select-var*] [THEN  *statements1*] [ELSE *statements2*] |

Parameter(s)

|  |  |
| --- | --- |
| ***var1*** | Reads the next value in the select-list, and assign it to ***var1***. |
| ***var2*** | Assigns the value mark count to ***var2***. This option is applicable only to external select-lists constructed through the TCL **SSELECT** command. |
| **FROM *select-var*** | Reads values from the named select-list variable ***select-var***. If ***select-var*** is not specified, the default select-list variable is used. |
| **THEN *statements1*** | Executes ***statements1*** unless at the end of the list. |
| **ELSE *statements2*** | Executes ***statements2*** if at the end of the list. |

Description

The **READNEXT** statement assigns the next value from an active select-list to the specified variable. If it is a select-list of item-IDs, the variable may then be used in a **READ** statement to read the file item. (The **SELECT** statement might also create a select-list of attributes from a dynamic array.)

The select-list can be either an internal select-list or an external select-list. See [SELECT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/select_statement.htm) for more information on creating select-lists. If a value is successfully read from the select-list, the variable ***var*** is assigned to the value, and the **THEN** statements are executed. When the end of the select-list is reached, ***var*** is set to the null string and the **ELSE** statements are executed.

Example

This application creates an alphabetical list of each item-ID in the file **CUSTOMERS**. The file is selected to the select variable **LIST**, and each item-ID is read from **LIST** with **READNEXT**. The **END.OF.LIST** variable is set to true when the **READNEXT** statement fails to read any more item-IDs, and is used to complete the loop. The actual alphabetizing is accomplished with the **LOCATE** and **INS** statements.

|  |
| --- |
| EQUATE TRUE TO 1, FALSE TO 0  OPEN "CUSTOMERS" TO CUSTFILE ELSE     ABORT 201, "CUSTOMERS"  END  SELECT CUSTFILE TO LIST  ALPH.LIST = ""  END.OF.LIST = FALSE  LOOP     READNEXT ID FROM LIST ELSE        END.OF.LIST = TRUE     END  UNTIL END.OF.LIST DO     LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION  THEN        PRINT ID : " DUPLICATE ENTRY!  POSSIBLE FILE CORRUPTION"        ABORT     END ELSE        INS ID BEFORE ALPH.LIST<POSITION>     END  REPEAT |

## READTX Statement

The **READTX** statement reads the next record (block) on the magnetic tape or floppy disk unit in hexadecimal format, assigning its value to the specified variable.

Format

|  |
| --- |
| READTX *var* [RETURNING *var*]  [THEN  *statements*  END] [ELSE  *statements*  END] |

Description

The **READTX** statement may be used to read a record from an attached magnetic tape or floppy disk unit, converting each character into its ASCII hexadecimal equivalent. See [READT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readt_statement.htm) for more information on the syntax for **READTX**.

By converting to hexadecimal, tapes or floppy disks containing segment marks (**CHAR(255)**) may be read by mvBASIC. **READTX** is most useful for reading tapes or floppy disks which were not written on a Mentor-based system and are therefore not in Mentor format.

After reading a tape or floppy disk in hexadecimal format via **READTX**, the **ICONV** function may be used with conversion code MX to convert the text back to readable text. To read a tape or floppy disk in its natural format, use the **READT** statement. See [READT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readt_statement.htm) for more information, and see [Appendix B: List Of ASCII Codes](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/appendix_b_list_of_ascii_codes.htm) for a list of ASCII codes.

Example

In this application, the **READTX** statement reads the data off the tape or floppy disk in hexadecimal format, and the **ICONV** function is used to convert the data back into ASCII characters.

|  |
| --- |
| READTX RECORD ELSE     PRINT "SYSTEM ERROR --"  END  RECORD = ICONV(RECORD,"MX") |

## CHAIN Statement

The **CHAIN** statement terminates execution of a program and executes a TCL command.

Format

|  |
| --- |
| CHAIN *command-expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***command-expr*** | Any command to be passed to TCL. |

Description

Like the **EXECUTE** statement, the **CHAIN** executes a TCL command. The **CHAIN** statement differs from the **EXECUTE** statement, however, in that it does not support any of **EXECUTE**’s features (such as capturing output or error messages), and it does not return to the program, but returns directly to the environment which called the program.

If the **CHAIN** statement is used to execute another program, parameters cannot be directly passed to the second program. However, if the I option (which suppresses initialization of all values) is used with the **RUN** command, the **COMMON** area may be used to pass parameters from one program to the next. See [COMMON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/common_statement.htm) for more information.

The data stack may be used to supply input which the TCL command might request. See [DATA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/data_statement.htm) for more information.

Example

To end a program by running another program, **WRAPUP**, the code might read:

|  |
| --- |
| CHAIN "RUN BP WRAPUP" |

## READV Statement

The **READV** statement permits a single attribute of a file item to be read and placed into a dynamic array variable.

Format

|  |
| --- |
| READV *var* FROM [*filevar*,] *item-ID*, *attr#*  [THEN *statements1*] [ELSE *statements2*] |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Assigns to ***var*** the string value of the attribute, in dynamic array form. |
| **FROM *[filevar,] item-ID, attr#*** | Assigns ***var*** the string value of attribute ***attr#*** in ***item-ID*** in the file which was opened as ***filevar***.  ***item-ID*** is an expression evaluating to an item-ID.  ***attr#*** is a valid expression which must evaluate to an attribute number in the specified item. If the attribute number is zero then it returns the number of attributes in the item being read. This is useful when processing items of different sizes, where a different algorithm is required for very large items.  If ***filevar*** is not specified, the default file variable is used, which is the file most recently opened without a file variable assignment. If the specified item-ID or attribute number does not exist, ***var*** is assigned the value of the null string (**""**). |
| **THEN *statements1*** | Executes ***statements1*** if the attribute is read successfully. |
| **ELSE *statements2*** | Executes ***statements2*** if the item cannot be found or if the attribute cannot be read. |

Description

Before a file may be accessed in a **READV** statement, it must be opened with an **OPEN** statement or an error will occur at run-time. See [OPEN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/open_statement.htm) for more information.

In mvBASIC, there are also **READ**, **READU**, and **READVU** statements available. The **READVU** statement sets an item update lock on the item before reading the attribute. The **READ** statement reads the entire file item, and the **READU** statement sets an item update lock on the file item before reading it.

See [READU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readu_statement.htm), [READV Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readv_statement.htm) and [READV Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readv_statement.htm) for more information.

Example

In this application, the file represented by **CUSTFILE** contains the name, address, and phone number of each customer. The phone number is kept in Attribute 6 of each item. To retrieve only the customer's phone number, the **READV** statement is used to capture Attribute 6.

|  |
| --- |
| PRINT "ENTER CUSTOMER ID : " :  INPUT ID  READV PHONE FROM CUSTFILE,ID,6 ELSE     PRINT "ERROR!"     STOP  END  PRINT ID , PHONE |

## READB Statement

The **READB** statement reads the specified number of characters from a binary item.

Format

|  |
| --- |
| READB *str* FROM *file.var*,*item.id*,*start.pos*, *no.of.bytes* THEN/ELSE |

Description

mvBASIC allows the user to read a specified number of characters from a binary item stored in a disk file. The format is very similar to **READV**, but rather than specifying the attribute to be read, the user specifies the starting position, and number of bytes to read. **READBU** behaves like **READVU**, and sets and obeys item locks.

If a start position of 0 is specified, the number of bytes (***no.of.bytes***) in the item is returned. If a start position beyond the end of the item is specified, null is returned. If too many bytes are specified, as many as are available are returned.

Binary items are stored in 2000 byte increments, so there may be extra data following the last meaningful data byte.

## READF Statement

The **READF** statement reads the specified number of characters from a normal item.

Format

|  |
| --- |
| READF *str* FROM *file.var*,*item.id*,*start.pos*, *no.of.bytes* THEN/ELSE |

Description

mvBASIC allows the user to read a specified number of characters from any normal item stored in a disk file. The format is very similar to **READV**, but rather than specifying the attribute to be read, the user specifies the starting position and number of bytes to read. **READFU** behaves like **READVU** and sets and obeys item locks.

If a starting position of 0 is specified, the number of bytes in the item is returned. If a starting position beyond the end of the item is specified, null is returned. If too many bytes are specified, as many as are available are returned.

## Reading and Writing Tapes or Floppy Disks

mvBASIC includes several statements for tape and floppy disk processing. For the purpose of this discussion a floppy disk functions as a tape and therefore, is included in any reference to a tape. For reading and writing strings on tape, there are the **READT** and **WRITET** statements. As expected, the **READT** statement reads the next record off the attached tape device, and the **WRITET** statement writes a record onto the tape. The **READTX** statement is designed for tapes which might contain segment marks. The **READTX** statement is identical to the **READT** statement, except that the data from the tape is translated into ASCII hexadecimal format before it is assigned to the string. The **ICONV** function can then be used to translate the string back into readable format. **READTX** is designed for reading segment marks (**CHAR(255)**) from a tape.

In addition, there are statements to simulate the **T-WEOF** and **T-REW** commands. The **WEOF** statement writes an End-Of-File mark at the current position of the tape, and the **REWIND** statement rewinds the tape to the beginning.

Each of the tape I/O statements includes **THEN** and **ELSE** clauses to specify action according to whether the tape statement was successful. The **ELS** clause is often used to produce a meaningful error message by calling the **SYSTEM(0)** function. The **SYSTEM(0)** function returns a number from 0 to 4, reflecting whether the latest tape I/O statement worked, and if it didn’t, what the problem was. See [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) for more information.

## Case Construct

The **CASE** construct performs a conditional selection of a sequence of statements.

Format

|  |
| --- |
| BEGIN CASE     CASE *expr*  *statements*     CASE *expr*  *statements*        .        .        .  END CASE |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | An expression to be evaluated for its logical value. |
| ***statement*** | Statements to be executed if the previous ***expr*** had been tested to be logically true. |

Description

A **CASE** construct must begin with a **BEGIN CASE** statement and end with an **END CASE** statement. The **CASE** construct evaluates a series of conditions until one is true and executes a set of statements accordingly. The expressions in the **CASE** statements are evaluated sequentially for their logical value until a value of true is encountered. When an expression evaluates to true, the statements between the **CASE** statement and the next **CASE** statement are executed, and all subsequent **CASE** statements are skipped. Execution continues with the next sequential statement following the **END CASE** statement.

If none of the expressions evaluate to true, no action is performed, and program execution continues with the statement after the **END CASE** statement.

The **CASE** statement can usually replace multiple nested IF constructs: it is much more readable and easier to implement.

Example

To test a variable **NUMBER** for positive or negative value, the source code might read:

|  |
| --- |
| BEGIN CASE     CASE NUMBER > 0     PRINT "POSITIVE"     CASE NUMBER < 0     PRINT "NEGATIVE"     CASE 1     PRINT "ZERO"  END CASE |

Note that the third and last condition reads **CASE 1** instead of **CASE NUMBER = 0**. In this situation the two conditions are equivalent since the last condition would only be tested if the first two failed. **CASE 1** is often used as the last condition of a **CASE** statement, as a catch-all condition.

## IF Construct

The **IF** construct allows execution of a statement or series of statements if the calculated expression is true, or of a separate set of statements if it is false.

Format

|  |
| --- |
| IF *expr* THEN  *statements*  END [ELSE  *statements*  END] |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | Any mvBASIC expression to be calculated for its logical value. |
| ***statements*** | Statement or set of statements to be executed conditionally. |

The **IF** construct calculates the given expression for its logical values. The expression is false if it evaluates to 0 or the null string; it is true if it evaluates to anything else. If the expression is true, it then allows the statements following **THEN** to be executed; if the expression is false, it allows the statements following the **ELSE** to be executed, or if there is no **ELSE** portion, it allows program execution to continue with the next executable statement.

Both the **THEN** clause and the **ELSE** clause are optional; however, one or the other must be included.

**IF** constructs may be nested. However, it is recommended to use a **CASE** construct instead if possible.

Statement Syntax

Although the logistics of the **IF** construct are relatively simple, the syntax is very exact. These restrictions apply:

* Neither **THEN** nor **ELSE** can begin a program line. That is, this construct:

|  |
| --- |
| IF ANSWER="Y"        THEN... |

results in an error message at compile time.

* When the statements following the **THEN** or **ELSE** are kept on a single line, they must be separated by a semicolon (**;**). That is, this construct is correct:

|  |
| --- |
| IF PROFIT THEN GOSUB 100; PRINT PROFIT ELSE     GOSUB 200;     PRINT LOSS |

* When the statements following the **THEN** or **ELSE** are written on more than one line, the **THEN** or **ELSE** must be the last word on its line and an **END** statement must end the set of statements. For example, the above example may be written:

|  |
| --- |
| IF PROFIT THEN        GOSUB 100        PRINT PROFIT     END ELSE        GOSUB 200        PRINT LOSS     END |

|  |  |
| --- | --- |
| **NOTE** | The second variation is much easier to read than the first. |

Example

In this application, **IF** constructs are nested to calculate the winner in a game of blackjack. It is sometimes difficult to determine which **END** statement belongs with which **THEN** or **ELSE**. A **CASE** statement is perhaps more appropriate to this function. See [CASE Construct](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/case_construct.htm) for more information.

|  |
| --- |
| IF DEALERSCORE > 21 THEN     PRINT "I WENT OVER. YOU WIN."     YOURWINS = YOURWINS + 1  END ELSE     IF NOT(DEALERSCORE < YOURSCORE) THEN        PRINT "MY SCORE IS " : DEALERSCORE : ".            I WIN."        IF DEALERSCORE = YOURSCORE THEN            PRINT "" ,"HOUSE RULES-DEALER ALWAYS WINS IN A TIE."            MYWINS + = 1        END     END ELSE     IF NOT(HIT = 11) THEN        PRINT "MY SCORE IS " : DEALERSCORE : ".             I HAVE TO HOLD. "        PRINT "YOU WIN."        YOURWINS + = 1        END ELSE PRINT "5 CARDS. I WIN."; MYWINS             + = 1     END  END |

## MATREAD Statement

The **MATREAD** statement reads a file item and assigns each attribute to elements of a dimensioned array.

Format

|  |
| --- |
| MATREAD *array* FROM [*filevar*,]*item-ID* [SETTING *var*]  [THEN  statements  END] [ELSE  statements  END] |

Parameter(s)

|  |  |
| --- | --- |
| ***array*** | Dimensioned array to be assigned. The array must have been dimensioned with a **DIMENSION** or **COMMON** statement before it may be assigned with the **MATREAD** statement. |
| ***filevar*** | File variable to which the file was opened. If ***filevar*** is not specified, the default file variable is used, which is the last file opened without a file variable assigned. |
| ***item-ID*** | An expression evaluating to the item-ID to be read. If the item is not found, the contents of ***array*** remain unchanged. |
| **SETTING *var*** | Assigns to ***var*** the number of attributes in the file item, regardless of whether this number is greater than the dimensions of the ***array***. |
| **THEN *statements*** | Executes ***statements*** if the item-ID is found. |
| **ELSE *statements*** | Executes ***statements*** if the item-ID is not found. |

Description

The **MATREAD** statement assigns the attributes of a file item to consecutive elements of the specified dimensioned array. The first attribute of the item becomes the first element of array, the second attribute of the item becomes the second element of array, and so on. The array must be named and dimensioned in a **DIMENSION** or **COMMON** statement before it is used in this statement.

A **MATREAD** statement does not set an update lock on the specified record. That is, the record remains available for update to other users. To prevent other users from updating the record until it is released, use a **MATREADU** statement. See [MATREADU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matreadu_statement.htm) for more information.

If the number of attributes in the file item is greater than the dimensions of the array, the remainder of the attributes are placed into the last element of the array, separated by attribute marks (**CHAR(254)**). If the number of elements in the array is greater than the number of attributes in the item, the extra elements in the array are assigned a null value.

The **MATREAD** statement performs the same function as using the **READ** statement to read a dynamic array and then using the **MATPARSE** statement to assign a dimensioned array to the same elements.

Example

In this application, the file item containing an employee’s statistics is read into the dimensioned array **EMP**. The **EMP** array is dimensioned to 100 elements, and the **MATREAD** statement reads the elements of the array from the employee’s file.

The **SETTING** clause of the **MATREAD** statement assigns a variable to the number of attributes in the file item, which is checked to see if it exceeds the dimensions of the array.

|  |
| --- |
| DIMENSION EMP(100)  ID  =  "PHONE"  MATREAD EMP FROM EMPLOYEES, NAME SETTING NUM  ELSE     PRINT "CANNOT LOCATE EMPLOYEE"     STOP  END  IF NUM > 100 THEN     .     .     . |

## OPEN Statement

The **OPEN** statement is necessary to access a file in the current program.

Format

|  |
| --- |
| OPEN ['DICT',] *file* [TO *filevar*] THEN/ELSE *statements* |

Parameter(s)

|  |  |
| --- | --- |
| **'DICT'** | Opens the file dictionary. If the dictionary is not specified, the data file is assumed. |
| ***file*** | An expression evaluating to the file name to be opened. If the file is one of several data files associated with a single file dictionary, it may be opened by the syntax, *'****dictname*,*file'***, with ***dictname*** the name of the file dictionary. |
| **TO *filevar*** | Defines ***filevar*** as the file variable name by which the file is accessed. If the **TO** ***filevar*** clause is not specified, the file may only be accessed as the default file variable. |
| **THEN *statements*** | Executes statements if ***filename*** is opened successfully. |
| **ELSE *statements*** | Executes statements if ***filename*** cannot be opened. This clause is generally used to cause print error messages or to stop or abort the program. |

Description

The **OPEN** statement prepares a file for use by the current mvBASIC program. All references to a file within an mvBASIC program must be preceded by a separate **OPEN** statement for that file.

If a file variable is not assigned with the **TO** keyword, the file is assigned to the default file variable. Any subsequent file I/O statements that do not specify a file variable default to this file. Note that default file variables are not local to the program from which they are executed; when a subroutine is called, the current default file variable is shared with the calling program.

There is no limit to the number of files which can be open at a given time. However, if multiple files are opened and accessed concurrently, file variables must be used. The default file variable can represent only one file at a time.

Example

In this application, the **OPEN** statement is used to open a reservation file, and the operator is asked to enter the customer's last name, to be used as an item-ID. If the reservation file is not found, the programs abort. A **READ** statement is then used to find the file item. If the item is found, any current reservations is shown; if not, a new reservation may be entered.

|  |
| --- |
| OPEN "RESERVATIONS" TO RES.FILE ELSE     ABORT 201,"RESERVATIONS"  END     .     .     .  LOOP     PRINT "LAST NAME :  " :     INPUT ITEM.ID     READ RECORD FROM RES.FILE,ITEM.ID THEN        PRINT ITEM.ID : " ON FILE."        GOSUB SHOW.RES     END ELSE        PRINT ITEM.ID : " NOT ON FILE"        GOSUB ENTER.RES     END  UNTIL LAST.NAME = "" DO REPEAT |
|  |

It is also possible to open a file in a very simple way – to the ‘default’.

|  |
| --- |
| OPEN "RESERVATIONS" ELSE STOP  LOOP     PRINT "LAST NAME :  " :     INPUT ITEM.ID     READ RECORD ITEM.ID THEN |

This is seen in a few programs in the SAIG System.  
The code should have been written this way:

|  |
| --- |
| OPEN "RESERVATIONS" TO RES.FILE ELSE STOP 201,"RESERVATIONS"  LOOP     PRINT "LAST NAME :  " :     INPUT ITEM.ID     READ RECORD FROM RES.FILE,ITEM.ID THEN |

## The Print and CRT Statements

## Sending Output to the Screen and Printer

There are two standard output devices available to an mvBASIC program:

* Terminal screen
* Printer

### Output Devices (PRINT, CRT, DISPLAY)

The **CRT** and **DISPLAY** statements send output only to the terminal screen, and the **PRINT** statement sends its output either to the terminal screen or to the printer, depending on which has been selected as the output device. The syntax of all three statements is identical, except that the **PRINT** statement accepts the ON keyword for multiple print units.

In a broader sense, file items and attached tape devices or floppy disk devices can also be considered output devices. See [Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm) and [Reading and Writing Tapes or Floppy Disks](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_writing_tapes_or_floppy_disks.htm) later in this section for information on file I/O, tape I/O, and floppy disk I/O.

### Sending Output to the Printer (PRINTER)

The **PRINT** statement by default sends output to the screen. There are two ways, however, to force the **PRINT** statement to send output to the printer: by the P option to the RUN command, or by the **PRINTER ON** statement. The **PRINTER ON** statement signifies that all subsequent **PRINT** statements will send output to a Spooler print unit. At the end of the program, the print unit will be sent to the Spooler.

The **PRINTER OFF** statement returns to the default condition: all **PRINT** statements after a **PRINTER OFF** statement will send output to the terminal screen again. To print output before the end of the program, the **PRINTER CLOSE** statement is available to send everything in the print unit directly to the Spooler.

### Print Units

When output is being sent to a printer, the **ON** keyword to **PRINT** becomes significant. Generally, all printer output is sent to print unit 0. However, if several reports are being generated simultaneously, the ON keyword can be used to place output in several different print units (ranging from 0 to 599).

For example, suppose a program generates two reports, one displaying the names of all customers who are two months late on their bills, and the other displaying the names of all customers who have birthdays approaching. The program goes through each customer’s record in sequence. If bills have not been paid, the customer’s name and address are printed from print unit 0, and the customer is billed. If the customer has a birthday coming up, the name and address are printed from print unit 1, and the customer is sent a birthday card. At the end of the program, two complete (and hopefully distinctive) lists are printed out.

### Formatting and Positioning Output

Normally output will be printed at the current position, and will force a carriage return and linefeed at the end of output. The print expression, however, may include features to tabulate output, to suppress the carriage return and linefeed, and (in the case of screen output) to place output at any coordinate, clear the screen, clear the line, or access any of several terminal capabilities.

In addition, format can be masked directly in an output expression. Format masking is a mechanism in mvBASIC by which data can be converted into a readable format without changing the data itself.

### Tabulation and Carriage Return Suppression

A comma (**,**) in the print expression will force a tab to be printed at that position. A trailing colon (**:**) specifies that the automatic carriage return and linefeed will be suppressed in output.

### Formatted Screens (@)

The **@** function provides direct control of a terminal screen. When the **@** function is used in a print expression, it generates a command sequence that is sent to the terminal screen, and the screen responds accordingly. In particular, the **@** function can be used to move the cursor to any coordinate position on the screen. It can also be used to clear the screen, to clear to the end of the line, or to place the text in blinking mode. A full list of the features for the **@** function is included in the reference page for **@**.

Using the **@** function, a formatted screen can be generated. Programs can use the **@** function to clear the screen and show a menu by sending menu options to different coordinates on the screen. The programmer might choose to turn the echo feature off to prevent user input from appearing on the screen.

For formatted screens, the **INPUT @** statement can take input from any coordinate on the screen. In addition, the **INPUT @** statement performs format masking directly on the input. See [Terminal Input](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/terminal_input.htm) later in this section for more information on **INPUT @**.

Example

To print a menu on the screen, the source code might read:

|  |
| --- |
| PRINT @(-1) :  PRINT @(8,3) : "CHOOSE ONE: " :  PRINT @(16,6) : @(-13) : "E" : @(-14) : "DIT AN ENTRY" :  PRINT @(16,8) : @(-13) : "N" : @(-14) : "EW ENTRY" :  PRINT @(16,10) : @(-13) : "D" : @(-14) : "ELETE AN ENTRY" :  PRINT @(16,12) : @(-13) : "Q " : @(-14) : "UIT" :  ECHO OFF  INPUT @(1,23) : ANSWER,1 |

The code in the preceding example does the following:

* The first line of code clears the screen.
* The second line prints **CHOOSE ONE** at column **8**, row **3**.
* The third through sixth lines print the menu options at specific coordinates, with the first character in reverse video mode. Thus, the first character stands out on the screen.
* The seventh line turns off the echo.
* The eighth line places the cursor at the bottom of the screen and accepts a single character as a response.

### Masking Data (FMT)

Numbers are stored in internal format. Internal format is a representation of data which makes calculation easier but is more difficult to read. Format masks convert numbers into a format that is easier to read. In addition, the **ICONV** function converts string data into internal format, and the **OCONV** function converts strings back into external format.

For example, if the dollar amount $14,912.15 were stored with the dollar sign and comma, then any calculations on that number would be impossible—dollar signs and commas are not permitted in numeric values. Also, if interest is being calculated on this dollar amount, it would be much more accurate if more than two decimal places were being kept.

Suppose that the given dollar amount represents the balance of a bank account. The bank keeps this figure to 5 digits of precision, to ensure that any calculations are accurate—suppose the actual figure stored is 1491214987. When this figure needs to be printed in a monthly statement, the data needs to be converted into a readable form. The program which generates the monthly statements will therefore print the data with a format mask, which will descale the number, round it to 2 decimal places, enter a comma where necessary, and precede it with a dollar sign. If the variable **BALANCE** contains 1491214987, and the program contains the lines:

|  |
| --- |
| PRECISION 4  .  .  .  PRINT BALANCE "29,$"  the output will be:  $14,912.15 |

A data mask can be implemented in two ways: either by using the **FMT** keyword, or by simply following data with the mask expression (as shown in the example). In the source code, the 2 signifies that the output should be rounded to two decimal places. The 9 is a descaling code, which determines where to place the decimal point—in this case, with a precision of 4 and a descaling code of 9, the decimal point is placed (9-4)=5 digits from the right. The, signifies to enter a comma every thousands place, and the **$** says to precede the expression with a dollar sign. There are many other codes available for masking data. For a full list and explanation of these codes, see the reference page for **FMT** in *Statement and Function Reference*.

### Headings and Footings

The **HEADING** statement can be used to specify a heading to be printed at the top of each page. It also has the facility to set up page parameters for use by **FOOTING** and **PAGE**.

If the output is being sent to the screen, then output will stop after each page of text once a **HEADING** statement is used. If a **FOOTING** statement is specified, a footing will be supplied at the end of the page, and the program will wait for a carriage return before continuing with output.

The **PAGE** statement can be used to force a new page at any point in the program, as long as a **HEADING** has been specified.

Note that **HEADING**, **FOOTING**, and **PAGE** will only affect the same output device that **PRINT** does. If multiple print units are being printed together, **HEADING**, **FOOTING**, and **PAGE** will affect print unit 0 (the default).

### The PRINTERR Statement

The **PRINTERR** statement allows mvBASIC programs to produce output messages using the Error Message Processor and the ERRMSG file.

Before the introduction of the **PRINTERR** statement, the Error Message Processor was only available through the use of the **STOP** and **ABORT** statements, both of which terminated the program.

## PRINT Statement

The **PRINT** statement sends data to the display terminal or to another specified print unit.

Format

|  |
| --- |
| PRINT[ON *unit#*] *print-expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***ON unit#*** | Specifies that data should be output to a Spooler print unit ***unit#***. ***unit#*** may be any integer in the range 0 to 254, with 0 as the default. Print unit 0 is interpreted as either the display terminal or the printer, depending on previous use of the **PRINTER** statement. When the program is terminated or when a **PRINTER CLOSE** statement is used, all print units are output to the printer. This option is used when several different reports are generated by the program simultaneously. |
| ***print-expr*** | Print expression optionally combined with commas and colons to designate the format of the output (as described below). If ***print-expr*** is omitted, a blank line is output. |

Description

The **PRINTER** statement determines the output device to which data will be written by the **PRINT** statement. See [PRINTER Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/printer_statement.htm) for more information. There is also a **CRT** (or **DISPLAY**) statement available in mvBASIC, which is identical to the **PRINT** statement except that it always prints its output to the screen, regardless of whether a **PRINTER ON** statement had been issued. In addition, there is also a **SEND** statement which may be used to send data to the screen of an attached line.

Formatted Output

The **FMT** function may be used to provide complex formatting specifications for output. See [FMT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/fmt_function.htm) for more information. Within the **PRINT** statement, however, commas and colons may be used to specify tab stops and suppress linefeeds.

* Expressions separated by commas (**,**) are printed at preset tab positions. Multiple commas may be used together to cause multiple tabulations between expressions. However, tab positions cannot be specified without being surrounded by expressions.
* Colons (**:**) encountered between expressions are interpreted normally as the string concatenation operator. If the last character of the **PRINT** statement is a colon, however, the linefeed and carriage return which usually follow the print statement are suppressed. This is especially useful when an **INPUT** statement is to follow, or in formatted screen programs.
* The **@** function may be used with the **PRINT** statement to send the cursor to a specified location on the screen. Also, as shown in the second example, the **@** function may be used to alter the attributes applied to text. For additional information regarding the **@** function, refer to [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm).

Example

In this application, the full contents of a dimensional array are printed via a **FOR...NEXT** loop. Only one element is printed at a time, however; hence, to imitate the actual structure of the array, tab stops are generated with commas and new lines are suppressed with colons. Before a new row is begun, the carriage return is generated by a null **PRINT** statement.

|  |
| --- |
| FOR I=1 TO 4     FOR J=1 TO 3        PRINT CUSTOMER(I,J), " " :     NEXT J     PRINT  NEXT I |

If the array **CUSTOMER** contains the name, telephone number, and marital status of each customer, the output might be:

|  |  |  |
| --- | --- | --- |
| FRED HENKEL | 555-1234 | SINGLE |
| ARCHIE ANDREWS | 555-4321 | MARRIED |
| MARGARET WOOD | 867-5309 | SEPARATED |
| LUCY RICARDO | 338-6887 | MARRIED |

In the next example, the **@** function is used with the **PRINT** statement to print an error in blinking text onto the bottom of the screen. It also clears the rest of the line, in case there had already been text on that line. Note that the cursor remains at the bottom of the screen after the error message is printed.

|  |
| --- |
| PRINT "ENTER YOUR SOCIAL SECURITY NUMBER : "  PROMPT ""  INPUT ANSWER,11,"\*#"\_  IF ANSWER MATCHES "3N-2N-4N" THEN     GOSUB SOCSEC  END ELSE     PRINT @(0,23) : @(-5) : ANSWER : " :  DOES NOT MATCH SS #" :     PRINT @(-6) : @(-4) :  END     .     . |

Aligned with this command is the PRINTER ON Statement.

When the PRINTER ON Statement is used, then until the PRINTER CLOSE statement is issued, all output will go to the previously assigned Printer Queue

PRINTER ON

PRINT HEADING “XYZZY LTD ‘T’ ‘D’ Page ‘P’”

PRINT “DETAILS”

PRINTER OFF

PRINT “Report has been produced”

END

   .

## CRT (or DISPLAY – not used) Command

## CRT Statement

The **CRT** statement sends data to the terminal display screen. It is identical to the **PRINT** statement except that it writes only to the terminal. The **DISPLAY** statement is identical to the **CRT** statement.

Format

|  |
| --- |
| CRT *print-expr*  DISPLAY *print-expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***print-expr*** | A print expression, optionally combined with commas and colons to designate the format of the output. If ***print-expr*** is omitted, a blank line is output. See [DISPLAY Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/display_statement.htm) for information on the format for a print expression. |

Description

The **CRT** and **DISPLAY** statements cause data to be output to the terminal screen, regardless of whether a **PRINTER ON** statement has been executed. See [DISPLAY Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/display_statement.htm) for more information on **CRT** and **DISPLAY**.

Example

To print the string **"HELLO…"** to the screen, the code might read:

|  |
| --- |
| CRT "HELLO…" |

or

|  |
| --- |
| DISPLAY "HELLO…" |

There are a myriad of CRT commands including functions such as:

CRT CHAR(27):”E” which is the equivalent of displaying ESC(ape) E to the screen

CRT @(-1) which is the same as Clear the Screen

CRT CHAR(7) which will provide a bell/beep sound from the user’s laptop speaker

These are detailed elsewhere, but are all used by the CRT command.

Please do not use the CRT ON command at SAIG. The files are not set-up/created for it’s use. The TANDOM command is of greater strength, and function.

Using the mvBASIC Debugger

This document summarizes Debugger commands, and describes how to fix a program error (bug), along with performing other Debugger tasks. This section contains the following topics:

|  |  |
| --- | --- |
| Debugger Commands: Quick Reference | Provides a summary listing of Debugger commands. |
| [Fixing a Bug](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/fixing_a_bug.htm) | Describes the process of fixing mvBASIC program errors. |
| [A Sample Program](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/a_sample_program2.htm) | Provides a sample mvBASIC program in order to demonstrate several programming tasks. |
| [Entering the Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/entering_the_debugger.htm) | Describes methods to activate the Debugger, and basic Debugger startup functions. |
| [Exiting the Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/exiting_the_debugger.htm) | Describes methods for shutting down the Debugger, and Debugger shutdown functions. |
| [Displaying and Changing a Variable](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/displaying_and_changing_a_variable.htm) | Describes how to use the Debugger to determine which variables are being assigned incorrectly. |
| [Accessing Source Code](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/accessing_source_code.htm) | Describes methods to identify and display a program’s source code, even if that code is contained in the data section of a file. |
| [Using Breakpoints and Tracing](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_breakpoints_and_tracing.htm) | Defines and describes how to use breakpoints in mvBASIC programs. |
| [Using Execution Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_execution_control.htm) | Describes how to control the execution of an mvBASIC program while in the Debugger in order to facilitate problem diagnosis. |
| [Printing Output](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/printing_output.htm) | Describes how to use print commands in the Debugger to toggle program output, toggle line printing, and to spool output to the printer. |
| [Using the Return Stack](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_the_return_stack.htm) | Describes the function and use of the Debugger’s return stack. |
| External Commands | A supplement for mvBase |

## Debugger Commands: Quick Reference

|  |  |
| --- | --- |
| **B** | Set breakpoint condition. |
| **D** | Display breakpoint and trace tables. |
| **DE** | Escape to system Debugger. |
| **DEBUG** | Escape to system Debugger. |
| **E** | Set or disable execution step. |
| **END** | End program and return to TCL. |
| **G** | Continue execution until next breakpoint, fatal error or step. |
| **K** | Delete breakpoint condition. |
| **LP** | Toggle output to the printer. |
| **M** | Toggle modal trap. |
| **N** | Ignore specified number of breakpoints. |
| **OFF** | Log off. |
| **P** | Toggle printing of program output. |
| **PC** | Spool print buffer. |
| **R** | Pop return stack. |
| **S** | Display subroutine stack. |
| **T** | Set trace variable or toggle trace table off and on. |
| **U** | Remove trace variable from table. |
| **V** | Verify the object code. |
| **Z** | Identify source code. |
| **$** | Print current line number. |
| **?** | Print current line number. |
| **/** | Print and change the value of a simple or array variable. |
| **/\*** | Print entire symbol table (all variables). |
| **[** | Set or remove a string window. |

## Fixing a Bug

A bug in a program is an error in the program’s logic which either prevents or impedes its performance. If a program doesn’t work perfectly, it is said to have a bug, and the programmer is obliged either to fix it or find a way around it.

Most new programmers debug a program by running it, reading the error message and its associated line number, and then examining the source code at the specified line. In many cases, this is enough: messages like **RETURN EXECUTED WITH NO GOSUB** are fairly simple to fix for a short program.

A programmer can also place trace statements at key points of execution; that is, statements that report what’s happening as the program executes. A **PRINT** statement can be placed within a conditional to determine whether a condition has proven true or not, or to display the value of variables. A **PRINT** statement can also be placed in a program loop, to report how many times the loop is being executed.

These debugging methods are fine if they are effective. However, they tend to be tedious to implement, and they involve recompiling the program with each attempt. Using the mvBASIC interactive Debugger, the debugging process becomes simpler and tidier.

Your first experience with the mvBASIC Debugger will probably be the result of an accident, either through a fatal error or because the BREAK key was pressed by mistake. At first, the only thing you need to know about the Debugger is how to get out of it (press END at the Debugger prompt). However, with a little patience you can learn to make the Debugger successfully fine-tune a program.

## A Sample Program

To demonstrate how the Debugger can be used, take the example of an internal office program called **BIRTHDAY**. The **BIRTHDAY** program asks users for their birthdays and then tells them how many days they have until their next birthday. The program works fine for the programmer:

|  |
| --- |
| >**RUN BP BIRTHDAY** |

The screen is cleared and the following prompt appears (with the underline representing the position of the cursor.) :

|  |
| --- |
| ENTER YOUR BIRTHDAY: **MM/DD/YY** |

 A date is entered:

|  |
| --- |
| ENTER YOUR BIRTHDAY: **6/4/65** |

The program converts the date and then prints out how many days there are until the programmer’s next birthday.

|  |
| --- |
| ENTER YOUR BIRTHDAY: **04 JUN 1965**  ON 06/04/1990, IN 319 DAYS, YOU WILL BE 25 YEARS OLD.    > |

The programmer was satisfied by this performance, since the tricky part of the program was to make sure that it did not report a birthday that had already passed (e.g., **IN -46 DAYS, YOU WILL BE 24**).

Since the test above was performed in July and the date entered was in June, the tricky part seemed to have been solved.

Later the same day, however, it was reported that the program did not work. In particular, when supplied a date in September, the program still jumped ahead a year:

|  |
| --- |
| ENTER YOUR BIRTHDAY: **3 SEP 1961**  ON 09/03/1990, IN 410 DAYS, YOU WILL BE 29 YEARS OLD. |

The programmer quickly examined the source code but could not find an error. However, using the Debugger, perhaps the user can discover something the programmer could not.

### Printing Source Code

First, run the program with the **D** option, which forces the program to enter the Debugger before executing line 1.

|  |
| --- |
| >**RUN BP BIRTHDAY (D)** |

Before the program begins execution, the Debugger is invoked and **E1** is printed, signifying that it stopped before executing line 1. The asterisk (**\***) is the Debugger prompt.

|  |
| --- |
| \*E1  \* |

The first thing is to examine the source code. Before you can access the source code, you need to turn trace on with the **T** command.

|  |
| --- |
| **\*T<ENTER>** |

The source code is now available for listing by the Debugger with the **L** command. Since it is a short program, **L** can be used with the **\*** option, specifying that the entire source code item should be printed.

|  |
| --- |
| \*L\*  001   PROMPT " "  002   DIM BIRTHDAY(3), TODAY(3)  003   EQUATE TRUE TO 1,  004      FALSE TO 0,  005      BIRTH.MONTH TO BIRTHDAY(1),  006      BIRTH.DATE TO BIRTHDAY(2),  007      BIRTH.YEAR TO BIRTHDAY(3),  008      THIS.MONTH TO TODAY(1),  009      THIS.DATE TO TODAY(2),  010      THIS.YEAR TO TODAY(3)  011  012   INCREMENT = FALSE  013   PRINT @(-1) : "ENTER YOUR BIRTHDAY:":  014   INPUT @(20,0) BIRTHDAY.INT "D"  015   TODAY.INT = DATE()  016   TODAY.EXT = OCONV(TODAY.INT,"D/")  017   MATPARSE TODAY FROM TODAY.EXT,"/"  018   BIRTHDAY.EXT = OCONV(BIRTHDAY.INT,"D/")  019   MATPARSE BIRTHDAY FROM BIRTHDAY.EXT,"/"  020   IF THIS.MONTH > BIRTH.MONTH THEN  021      INCREMENT = TRUE  022   END  023   IF (THIS.MONTH = BIRTH.MONTH OR THIS.DATE  BIRTH.DATE) THEN  024      INCREMENT = TRUE  025   END  026   IF INCREMENT THEN  027      THIS.YEAR += 1  028   END  029   AGE = THIS.YEAR - BIRTH.YEAR  030   NEXT.BIRTHDAY = BIRTH.MONTH : "/" : BIRTH.DATE : "/" : THIS.YEAR  031   NEXT.BIRTHDAY.INT = ICONV(NEXT.BIRTH-DAY,"D")  032   DAYS.TO.BIRTHDAY = NEXT.BIRTHDAY.INT - TODAY.INT  033   IF DAYS.TO.BIRTHDAY = 0 THEN  034      PRINT  035      PRINT "HAPPY BIRTHDAY!"  036      PRINT "TODAY, " : TODAY.EXT : ", YOU ARE " : AGE : " YEARS OLD"  037   END ELSE  038      PRINT  039      PRINT "ON ":NEXT.BIRTHDAY:", IN ": DAYS.TO.BIRTHDAY :" DAYS,":  040      PRINT " YOU WILL BE ":AGE:" YEARS OLD."  041   END  042 STOP  043 END    \* |

At first glance, it is obvious that the problem lies in the variable T**HIS.YEAR: THIS.YEAR** is being incremented when it should not be. Before you start editing the program, however, use the Debugger to confirm your suspicions.

### Using Breakpoints and Trace Variables

To examine the value of **THIS.YEAR** at the end of the program, the user needs to set up a breakpoint so that variables can be examined before the program ends. A breakpoint condition is a condition that invokes the Debugger whenever it is true.

The **B** command is used to assign a breakpoint condition. You can choose to break when the **THIS.YEAR** variable is equal to **1990**.

|  |
| --- |
| **\*BTHIS.YEAR=1990<ENTER> +** |

The breakpoint condition says to transfer into the Debugger when the variable **THIS.YEAR** is equal to **1990**. The plus sign (**+**) is printed after pressing the ENTER key to signify that the breakpoint was accepted into the breakpoint table.

Enter **THIS.YEAR** as a trace variable. A trace variable is a variable that is printed whenever a breakpoint is encountered. Enter it into the trace table with the **T** command:

|  |
| --- |
| **\*TTHIS.YEAR<ENTER> +** |

Add to the trace table the variables **THIS.MONTH**, **BIRTH.MONTH**, **THIS.DATE**, and **BIRTH.DATE**. It is suspected that the problem may be that these variables are not being assigned correctly.

|  |
| --- |
| **\*TTHIS.MONTH<ENTER> +**  **\*TBIRTH.MONTH<ENTER> +**  **\*TTHIS.DATE<ENTER> +**  **\*TBIRTH.DATE<ENTER> +** |

To display the breakpoint and trace tables, use the **D** command:

|  |
| --- |
| **\*D**  T1 THIS.YEAR  T2 THIS.MONTH  T3 BIRTH.MONTH  T4 THIS.DATE  T5 BIRTH.DATE  T6  B1 THIS.YEAR=1990  B2  B3  B4    \* |

The **G** command continues execution of the program.

|  |
| --- |
| **\*G<ENTER>** |

The screen clears and the prompt is printed. Type **9/3/61** and press ENTER:

|  |
| --- |
| ENTER YOUR BIRTHDAY: **9/3/61** |

The program halts when the breakpoint is reached.

|  |
| --- |
| ENTER YOUR BIRTHDAY: **3 SEP 1961**  \*B1 28       END  THIS.YEAR 1990  THIS.MONTH 07  BIRTH.MONTH 09  THIS.DATE 20  BIRTH.DATE 03  \* |

As expected, the condition is true after line 27 has been executed. The message **B1 28** means that item 1 on the breakpoint table caused the break, and the line about to be executed is line 28. (The actual text of that line is displayed in half-intensity on the terminal screen.) The current values of the trace variables are printed.

### Displaying and Changing a Variable

The error in the program becomes increasingly obvious as you continue in the Debugger. Since **THIS.MONTH**, **BIRTH.MONTH**, **THIS.DATE**, and **BIRTH.DATE** contain the correct data, the problem is in the way they are being compared. For **THIS.MONTH** to be incremented, the **INCREMENT** variable must be true. Confirm this by using the **/** command to print out the current value of **INCREMENT**:

|  |
| --- |
| **\*/INCREMENT** |

When ENTER is pressed, the value of **INCREMENT** is displayed, and you are given the opportunity to change its value.

|  |
| --- |
| **\*/INCREMENT<ENTER> 1=\_** |

Type **0** as the new value for **INCREMENT** and press ENTER.

|  |
| --- |
| **\*/INCREMENT<ENTER> 1=0<ENTER>**    \* |

Reset the value of **THIS.YEAR** to **1989**.

|  |
| --- |
| **\*/THIS.YEAR<ENTER> 1990=1989<ENTER>**    \* |

### Using Execution Steps

To find out which comparison is failing, step through the crucial lines of the program this time around to see what exactly is happening. To begin the program executing again after line 19 with an execution step of 1 (an execution step is a number of lines that should be executed before returning to the Debugger). The **E** command should be used to specify the execution step:

|  |
| --- |
| **\*E1** |

Add the variable **INCREMENT** to the trace table.

|  |
| --- |
| **\*TINCREMENT<ENTER>+** |

The **G** command continues execution after line 19. One line executes and the program returns to the Debugger. The line about to be executed will be printed on the screen, along with any trace variables.

|  |
| --- |
| **\*G19<ENTER>**  \*E20       IF THIS.MONTH > BIRTH.MONTH THEN  THIS.YEAR 1989  THIS.MONTH 07  BIRTH.MONTH 09  THIS.DATE 20  BIRTH.DATE 03  INCREMENT 0  \* |

Step through 3 more times, until you find that the **INCREMENT** variable has been changed.

|  |
| --- |
| **\*G**  \*E25       END  THIS.YEAR 1989  THIS.MONTH 07  BIRTH.MONTH 09  THIS.DATE 20  BIRTH.DATE 03  INCREMENT 1  \* |

By stepping through the program, you will see that the **INCREMENT** variable is changed immediately before line 25. List lines 23 through 25 with the **L** command:

|  |
| --- |
| **\*L23-25**  023   IF (THIS.MONTH = BIRTH.MONTH OR THIS.DATE > BIRTH.DATE) THEN  024      INCREMENT = TRUE  025   END  \* |

The problem is in the conditional for the **IF** statement. It becomes obvious that, as usual, the bug in the program is a simple logical error: the **OR** in line 23 should be an **AND**. With that simple edit, the program should run correctly.

Exit the Debugger with the **END** command, edit the source item, and recompile. The output now reads:

|  |
| --- |
| ENTER YOUR BIRTHDAY: **3 SEP 1961**  ON  09/03/1989, IN 45 DAYS, YOU WILL BE 28 YEARS OLD.    > |

### Assigning New Values for Testing

It seems as if the bug is gone, for now. However, you are not confident that the program will still behave correctly at the end of the year. Using the Debugger, you can change the value of the variable **TODAY.INT** in the program and see whether the program still works.

Run the program again with the **D** option, use the **T** command to turn trace on, and set a breakpoint to stop executing before line **16**. (The **$** symbol on the breakpoint table represents the current line number.) Then continue execution with the **G** command.

|  |
| --- |
| >**RUN BP BIRTHDAY (D)**    \*E1  \*T<ENTER>  \*B$=16<ENTER> +  \*G<ENTER> |

You will be prompted for a birthday. After entering a date, press ENTER. The program will halt before executing line **16**.

|  |
| --- |
| ENTER YOUR BIRTHDAY  **03 SEP 1961**  \*B1 16 TODAY.EXT = OCONV(TODAY.INT,”D/”)  \* |

Before line **16** is executed, reassign the value of the **TODAY.INT** variable with the **/** command. Then type the **G** command to continue execution.

|  |
| --- |
| **\*/TODAY.INT<ENTER>  7872=8036<ENTER>**    **\*G<ENTER>** |

**8036** is the internal value of December 31, 1989 and the program runs successfully:

|  |
| --- |
| ON 09/03/1989, IN 246 DAYS, YOU WILL BE 29 YEARS OLD.  > |

Now that you have an idea of what the Debugger can do and why you would use it, you can go over the specifics of its operation.

## Entering the Debugger

There are three ways to enter the Debugger:

* Pressing BREAK during program execution invokes the mvBASIC Debugger. (This feature can be turned off within a program with the **BREAK** statement.)
* The **D** option to the **RUN** command will cause the program to enter the Debugger before starting execution.
* If the **DEBUG** statement is encountered in execution, the program will enter the Debugger at that point.

**Fatal errors will also invoke the Debugger, with or without the user’s consent**  (Nonfatal errors will invoke the Debugger only if the **RUN** command is used with the **E** option.) When control passes to the Debugger for any of the above reasons, the current line number (preceded by

**I** for Interrupt,

**E** for Execution step, or

**M** for Modal)

is printed, and the **\*** prompt  displays.

Once in the Debugger, the user can print and change variable values, set breakpoint conditions or execution steps, and continue execution with the **G** command. When a breakpoint condition or execution step is reached, the Debugger will be instantly re-entered.

The Symbol Table

Variables within a program are each assigned a symbol, to be referenced by the interactive Debugger. When a program or subroutine is compiled, a symbol table is generated with the object code. The Debugger accesses the symbol table through a pointer in the file dictionary. If the program calls an external subroutine, a complete symbol table can be accessed by the Debugger for the external subroutine.

The **S** option to **COMPILE** suppresses the symbol table from being generated, but it should be used only when a program is fully operational. Without the symbol table, the Debugger’s function is greatly impeded.

## Exiting the Debugger

Other than returning to program execution with **G**, the Debugger can be exited using the following commands:

|  |  |
| --- | --- |
| **END** | Exits program, Debugger, and calling Proc or program (if any), and returns directly to TCL. |
| **OFF** | Exits both program and Debugger, and logs the user off the system directly. |
| **DE[BUG]** | Transfers execution to the system Debugger, and the **!** prompt will be displayed. The user can return to the mvBASIC Debugger (**\*** prompt) with the **G** command to the system Debugger. |
| **E and G ?? To be validated** | E will debug at 0 lines (None), and G will Go forward and processes unlimited lines (perform the program). ?? To e validated |

## Displaying and Changing a Variable

One of the most valuable things a programmer can learn about a failed program is what happens to the variables at different points in the program. By examining variable values, the programmer can determine which variables are being assigned incorrectly, and thus find out which statements are not being executed properly.

### Displaying All Variables

The **/\*** command displays all variables in the symbol table. All variables are reported, including file variables, select-list variables and dimensioned array variables. For example:

|  |
| --- |
| \*/\*  FILEVAR  PROGRAM=TESTIT  FILE.ARRAY(1,1)=JOEY  FILE.ARRAY(1,2)=FRED FLINTS  .  .  . |

By using the **/\*** command, the user can scan the values for all variables at once. However, the **/\*** command will not give the user the opportunity to change any values, and if there are many variables or some extremely long string variables, the values may scroll past the screen too quickly.

### Displaying and Changing Simple Variables

The values of simple variables can be printed and (optionally) reassigned with the **/** command.

Format

|  |
| --- |
| /*var* |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Variable name. |

Description

A single variable can be displayed with the **/** command and the variable name. The user will be shown the current value, and be prompted with an equals sign to change the value at will. Whatever the user types before pressing ENTER becomes the new value for the variable.

For example, to display the current value of the variable **STRING**, the user might type:

|  |
| --- |
| **\*/STRING** |

The Debugger will respond with the current value of **STRING** and an equals (**=**) sign. If **STRING** contains the word **HELLO**, the user sees the following with the underscore representing the user’s cursor position:

|  |
| --- |
| **\*/STRING<ENTER> HELLO=\_** |

The user can then enter a value for **STRING** and press ENTER. To leave the value unchanged, the user should press ENTER without reassigning the variable.

### Displaying and Changing Dimensioned Array Elements

The values of a single element or of all elements of a dimensioned array can also be printed and (optionally) reassigned with the **/** command.

Format

|  |
| --- |
| /*array* [(*n* [,*m*])] |

Parameter(s)

|  |  |
| --- | --- |
| ***array*** | Name of the dimensioned array. |
| ***n*** | Row number of the array element. If omitted, all elements of the array will be printed. |
| ***m*** | Column number of the array element. If array is two-dimensional, m must be supplied if ***n*** is supplied. |

Description

The individual elements of a dimensioned array can be treated like simple variables by referencing them with parentheses.

Example

To display the current value of element **2,3** of array **NAME.ARRAY**, the user might enter:

|  |
| --- |
| **\*/NAME.ARRAY(2,3)** |

The Debugger will respond with the current value of **NAME.ARRAY(2,3)** and an equals sign. If element **2,3** contains **HERB**, the user would see:

|  |
| --- |
| **\*/NAME.ARRAY(2,3)<ENTER> HERB=\_** |

(with the underscore representing the user’s cursor position).

The user then has the option to fill in a value for **NAME.ARRAY(2,3)**, or to leave it unchanged by pressing ENTER.

Alternatively, all elements of a dimensioned array can be displayed and changed by omitting the element reference. For example, to display the current values of all elements of array **NAME.ARRAY**, the user can type:

|  |
| --- |
| **\*/NAME.ARRAY** |

and will be prompted with the value of each element of the array as if each were specified individually. For example:

|  |
| --- |
| \*/NAME.ARRAY<ENTER> NAME.ARRAY(1,1)="JOEY"=  NAME.ARRAY(1,2)="FRED FLINTS"=\_   .   .   . |

### String Windows ([)

The value of some string variables might be too long to be printed on a single screen. For these strings, the **[** command should be used to specify a subset of each string to be printed.

Format

|  |
| --- |
| [*n*,*m*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Starting column of the substring. |
| ***m*** | Length of the substring. If 0, turn off string windowing. |

Description

If ***n*** and ***m*** are omitted, string windowing is turned off.

Example

Suppose the string **RECORD** contains over a hundred addresses separated by attribute marks, totaling over 2000 characters. When the user tries to print the output of **RECORD** in the Debugger, the entire screen is filled and the beginning of the string is scrolled off the screen. The programmer is only interested in characters towards the middle of the string; therefore, the number of characters printed out should be limited with:

|  |
| --- |
| **\*[800,400]**  **\*/RECORD** |

By limiting the output to 400 characters, the relevant portion of **RECORD** can be made accessible to the programmer.

## Accessing Source Code

The **L** command has been enhanced to read source code from the data section of the file containing the object code, unless the **Z** command has been used to identify the source code.

### Identifying Source Code (Z)

After entering the **Z** command, enter the file name followed by the program name, separated by a space. If file name and program are not specified, it is assumed that the source code is in the data section of the file containing the object code, and that the source code has the same name as the object code. For example, if the source code is in item-ID **TESTIT** in file **BP**, type:

|  |
| --- |
| **\*Z BP TESTIT** |

If the Debugger prompt (**\***) returns, the command was accepted. If the command is illegal for any reason (misspelling, extraneous spaces, etc.), the message **NO SOURCE** will be printed.

The **Z** command not only enables source code listing but also permits the current source line to be printed at half-intensity each time the Debugger is re-entered, or when the **?** or **$** commands are used.

If the program calls an external subroutine, the **Z** command can be used again to point to the source code of that subroutine. However, it does not have to be reinvoked each time execution transfers between the program and the subroutine.

|  |  |
| --- | --- |
| **NOTE** | It is not necessary to use the **Z** command if the source code exists in the same file as the object code. |

### Displaying Source Code (L, $, ?)

The L command may be used to read the source code from the data section of the file containing the object code (source code), and then display the source code.

Format

|  |
| --- |
| L [*n* [-*m*] | \*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Shows line ***n*** in the source code. |
| ***n*-*m*** | Shows the specified range of lines in the source code. |
| **\*** | Shows all lines of the source code. |

For example, to print out lines 59 through 61:

|  |
| --- |
| **\*L59-61**  059      INPUT NAME:  060      IF NAME = "" THEN  061         GOTO EXIT    \* |

In addition to **L**, the **$** and **?** commands print out the current line number and the corresponding source line.

The **$** and **?** commands are identical in function. For example the current line can be shown with:

|  |
| --- |
| **\*$<ENTER>** CUST.ENTRY L 59      INPUT NAME: OBJECT VERIFIES |

In the example, **CUST.ENTRY** is the name of the program, **L 59** refers to line number 59, and **INPUT NAME:** is the statement on that line of code.

The source line (**INPUT NAME:**) will be printed at half-intensity. If the **Z** command is not used before the **$** command, the source code will be omitted.

## Using Breakpoints and Tracing

A breakpoint is a condition that invokes the Debugger whenever it is true. A trace variable is a variable that is defined to be printed automatically when a breakpoint is encountered. The mvBASIC Debugger can support up to 4 breakpoints and up to 6 trace variables at a time. Each external subroutine to the program will have its own breakpoint and trace table, independent of the one created for the program.

### Establishing a Breakpoint (B)

The B command can be used to define a breakpoint in a program.

Format

|  |
| --- |
| B*var* *op* *value* [&*var* *op* *value*] |

Parameter(s)

|  |  |  |
| --- | --- | --- |
| ***var*** | Variable name to be tested. Alternatively, ***var*** can be the symbol $, specifying that the line number should be tested. | |
| ***op*** | Operator. One of the following: | |
| **=** | equals |
| **#** | not equals |
| **>** | greater than |
| **<** | less than |
| ***value*** | Value to test the variable by. Can be a numeric value, a string, or another variable in the program. If the value is a string, it must be enclosed in single or double quotes. A backslash is not accepted as a delimiter. | |
| **&** | Logical connector for two conditions. | |

Description

Although spaces have been supplied above for clarity, spaces are not accepted in the syntax for the **B** command. If the command is accepted, a plus sign (**+**) is printed. If the breakpoint table is already full with its maximum of 4 breakpoints, the message **TBL FULL** will be printed.

To picture a breakpoint condition in mvBASIC language, think of it as being equivalent to entering the following throughout the program:

|  |
| --- |
| **IF** var op expr  [**AND** var op expr] **THEN**  **DEBUG**  END |

For example, to enter the Debugger whenever the variable **COUNT** is greater than **10** and the variable **FOUND** has a logical value of false (0), type:

|  |
| --- |
| **\*BCOUNT>10&FOUND=0** |

Line numbers can be tested as well as variables. To specify that a line number is being tested, use a dollar sign (**$**) in place of a variable name. For example,

|  |
| --- |
| **\*B$>75&$<95** |

causes the program to re-enter the Debugger whenever the program is executing a line between **75** and **95** (exclusively). Conditions comparing line numbers can be combined with conditions comparing variables. For example,

|  |
| --- |
| **\*B$>75&FOUND=0** |

After a breakpoint condition is established with **B**, the program can continue execution with the **G** command or with a linefeed (CTRL+J). When a breakpoint condition is encountered, the Debugger is re-entered. The letter **B** with the breakpoint number and the line number will be printed, along with any trace variables. Trace variables are discussed in a later section.

### Deleting a Breakpoint (K)

A breakpoint can be deleted from the breakpoint table with the K command.

Format

|  |
| --- |
| K [*n*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Delete breakpoint ***n***. If ***n*** is omitted, delete all breakpoint conditions. ***v*** is determined by its position on the breakpoint table. |

Description

If the command is accepted, a minus sign (**–**) is printed.

### Defining Trace Variables (T)

The **T** command defines a trace variable. Alternatively, it turns the trace table on and off.

Format

|  |
| --- |
| T [*var*] |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Trace the variable ***var***. If ***var*** is omitted, toggle the trace table on or off. |

Description

If the trace variable is accepted, a plus sign (**+**) appears. If the trace table is already full with its maximum of 6 variables, the message **TBL FULL** prints.

If the **T** command is used without any arguments, it toggles the trace table on and off. If it is turned on, the word **ON** prints; if it is turned off, the word **OFF** prints. When the trace table is turned off, trace variables are not printed when a breakpoint is reached.

For example, to print the value of the variable **COUNT** every time a breakpoint is reached, type:

|  |
| --- |
| **\*TCOUNT** |

### Deleting a Trace Variable (U)

A variable can be deleted from the trace table with the **U** command.

Format

|  |
| --- |
| U [*var*] |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Delete variable ***var*** from the trace table. If ***var*** is omitted, all variables are deleted. |

Description

If the command is accepted, a minus sign (**–**) will be printed.

### Displaying Breakpoints and Trace Variables (D)

The breakpoint and trace tables can be displayed with the **D** command. For example:

|  |
| --- |
| **\*D**  T1 COUNT  T2 CUST.ARRAY(5)  T3  T4  T5  T6  B1 COUNT>10&FOUND=0  B2 $>75&FOUND=0  B3  B4 |

In the example, 2 trace variables and 2 breakpoints have been established.

## Using Execution Control

Having entered the Debugger, the programmer often needs to execute the program again in order to see what actually happens.

The program can be controlled by establishing either breakpoints or execution steps before continuing execution, to specify that the Debugger should be reinvoked when a condition becomes true or when a number of statements have been executed.

### Continue Execution (G)

The **G** command continues program execution.

Format

|  |
| --- |
| G [*n*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | The line number to continue execution at. If omitted, the current line number will be assumed. |

Description

Once the **G** command is used, the program will continue execution until the next breakpoint or, if an execution step has been specified, until the specified number of lines have been executed. See the next section for more information on execution steps.

A linefeed (CTRL+J) is a synonym for the **G** command with no arguments—that is, it will continue execution at the current line number. The linefeed has the advantage that it does not need to be followed by ENTER in order to be interpreted.

|  |  |
| --- | --- |
| ***CAUTION*** | The **G** command should not be used if the operator has entered the Debugger because of a fatal run-time error. Continuing execution after a fatal error may result in corrupted data structures. In such an instance, the operator should exit the Debugger immediately with the **END** or **OFF** command. |

### Setting an Execution Step (E)

The **E** command establishes or removes an execution step. An execution step is a number of lines to be executed before automatically re-invoking the Debugger:

Format

|  |
| --- |
| E [*n*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Returns control to the programmer every ***n*** lines. If ***n*** is omitted or 0, turn off the previous E command. |

Description

After enabling **E**, you can return to the program with the **G** command. With each ***n*** lines executed, the program will return to the Debugger with the current line number preceded by **E**. By using an execution step of 1, the program can be stepped through: the programmer can examine every source line before it is executed, thus tracing the action as it occurs.

While an execution step is in effect with **E**, breakpoints are disabled. The program will not stop at breakpoints until **E** has been disabled again.

Execution steps are global; that is, if the program enters a subroutine, the step remains unchanged.

### Ignoring Breakpoints (N)

The **N** command specifies that the next ***n*** breakpoints should be ignored.

Format

|  |
| --- |
| N [*n*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Bypass ***n*** breakpoints before returning control to the programmer. If ***n*** is omitted, bypass 1 breakpoint. |

Description

Although breakpoints are ignored by using the **N** command, the trace table is still printed at each bypassed breakpoint if it is enabled. By using the **N** command, the user can monitor the sequence of execution and the values of trace variables at each breakpoint, without re-entering the Debugger. The trace table can be disabled with the **T** command.

The **N** command is global;  that is, if the program enters a subroutine, the **N** command will still be in effect.

## Printing Output

The Debugger uses print commands to toggle program output, toggle line printing, and to spool output to the printer.

### Toggling Program Output (P)

The **P** command toggles printing of output from the program. If **P** is toggled **OFF**, only output from the Debugger will be printed; the output from the program execution will not be shown. When output is disabled with the **P** command, the word **OFF** is printed; when it is re-enabled, the word **ON** is printed. The default setting is **ON**.

### Toggling Line Printing (LP)

The **LP** command toggles printing of Debugger output on the line printer. If **LP** is toggled **ON**, output from the Debugger will be directed to the printer. When line printing is enabled with the **LP** command, the word **ON** is printed; when it is disabled again, the word **OFF** is printed. The default setting is **OFF**.

The LP command is an equivalent to the **PRINTER ON** and **PRINTER OFF** statements in mvBASIC.

### Close the Printer (PC)

The **PC** command spools the Debugger printer output to the printer. The **PC** command in the Debugger is an equivalent of the **PRINTER CLOSE** statement in mvBASIC. All Debugger output held for the printer will be sent to print immediately, rather than after the program is completed.

## Using the Return Stack

When the program transfers to an internal subroutine, the return address is pushed onto the return stack. The return address is the line the program transfers to at the end of the subroutine—that is, the line containing the statement immediately following the **GOSUB** that called the subroutine. If the subroutine calls another subroutine, the second address is pushed on top of the first address on the return stack, and so on for each embedded subroutine. As an **ENTER** statement is encountered, the top value on the stack is popped and taken as the return address for that subroutine.

### Displaying the Return Stack (S)

The **S** command generates a list of line numbers on the return stack. Possible responses are:

|  |  |
| --- | --- |
| ***=line#* SOURCE LINE** | Current subroutine will exit to line line#. When an **ENTER** statement is encountered, execution will transfer to line line#. (The corresponding source line is printed in half-intensity.) |
| ***=line1#* SOURCE LINE [*=LINE2#* SOURCE LINE...]** | Multiple subroutines are in effect. The current (top-level) subroutine will exit to line line1#. The next level subroutine will exit to line line2#, and so on. (Corresponding source lines are printed in half-intensity. |
| **STK EMP** | Nothing in stack. |
| **ILSTK** | Illegal stack. This usually signifies that an external subroutine is being executed. (External subroutines are not monitored by the return stack.) |

### Popping the Return Stack (R)

The **R** command pops the return stack. The program returns from the current subroutine as if an **ENTER** statement had been encountered at that point.

### Modal Traps (M)

A modal trap occurs whenever an external subroutine is called, returned from, or when a mainline program enters another mainline program. In response to the modal trap, type **M** to toggle the modal trap ON or OFF depending on the trap’s previous state.

When a modal trap occurs it will be indicated by entering the BASIC debugger (**\***), and one of these prompts appears:

|  |  |
| --- | --- |
| **\*MC*xxx*** | Indicates the modal trap was caused by a **CALL** statement. ***xxx*** indicates the source code line number where the **CALL** statement occurred. |
| **\*MR*xxx*** | Indicates the modal trap was caused by an **ENTER** statement. ***xxx*** indicates the source code line number where the **ENTER** statement occurred. |
| **\*ME*xxx*** | Indicates the modal trap was caused by an **ENTER** statement. ***xxx*** indicates the source code line number where the **ENTER** statement occurred. |

In response to the above modal traps, type **M** at the **\*** prompt to toggle the modal trap ON or OFF.

## Additional mvBase specific commands

When in the debugger, and wanting to know specific information outside the program – e.g. the Time, or Which port, and so on there is a command that will allow temporary TCL access in the middle of the process:

The >> Command

Typing in >> will temporarily drop the developer to TCL

Typing in << returns control back to the program processor.

For example

**\* G279  
PRINT “Here be monsters”  
\*>>  
>WHO  
298 MERRY  
>Time  
19:10:19  
<<  
\*  
PRINT “Here be monsters”**

This command is particularly useful for checking the contents of a data record during program execution.

Format, Data and Expressions

This section provides an overview of the essential components of the mvBASIC language. It describes program format, types of data, and how to store and access data within an mvBASIC program.

The following topics are presented in this section:

* [**Program Format**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/program_format.htm)
* [**Constants, Variables, and Data Types**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/constants,_variables,_and_data_types.htm)
* [**Building Expressions**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/building_expressions.htm)
* [**Advanced Data Types**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/advanced_data_types.htm)

# Program Format

An mvBASIC program is a sequence of statements directing the computer to perform a series of tasks in a specified order. A statement consists of keywords, constants, and variables.

|  |  |
| --- | --- |
| Keywords | Special words recognized by the mvBASIC compiler. |
| Constants | Values that do not change during the execution of a program. |
| Variables | Values that can change. |

An mvBASIC source line corresponds to a single attribute in the source code item. The line can begin with a statement label, but statement labels are not mandatory for all lines in mvBASIC. More than one mvBASIC statement can be placed on the same line, as long as they are separated by semicolons (**;**).

The syntax for an mvBASIC source line follows:

Format

|  |
| --- |
| [*label*] *statement* [; *statement* [; *statement* ...]] |

Example

|  |
| --- |
| 100   PRINT "HELLO, WORLD"  ;  PRINT "GOODBYE, WORLD" |

In the example, the statement label is **100**. It is followed by two **PRINT** statements, separated by a semicolon (**;**). In each statement, the word **PRINT** is a keyword and is followed by a constant string value (**"HELLO WORLD"** or **"GOODBYE WORLD"**), delimited by quotation marks.

A statement cannot be broken onto more than one line, unless it contains a comma in its syntax (for example, in the **EQUATE**, **COMMON**, or **DIMENSION** statements), in which case it can be broken into multiple lines after each comma.

The following topics are presented in this section:

* [**Types of Statements**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/types_of_statements.htm)
* [**Statement Labels**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_labels.htm)
* [**Writing Readable Code**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writing_readable_code.htm)

## Types of Statements

mvBASIC statements can be broken into the following general categories:

|  |  |
| --- | --- |
| Input/Output Control | Input statements control where the computer can expect data, and output statements control where the data is displayed or stored. The input or output device can be a terminal, a printer, a tape, a floppy disk, a file item or backup media device. Input statements include **INPUT**, **READ**, **READT**, and **GET**; output statements include **PRINT**, **WRITE**, **WRITET**, and **SEND**. |
| Program Control (Sequence of Execution) | In general, mvBASIC statements are executed in the order in which they appear, and program control statements are used to alter that sequence. Program control statements include **IF**, **CASE**, **LOOP**, **FOR**, **GOTO**, **GOSUB**, **CALL**, and **EXECUTE**. |
| Assignment of Variables and Constants | Assignment statements assign values to variables and names to constant values. Variables can be directly assigned with the **ASSIGNMENT** statement (**=**), and constants can be assigned with the **EQUATE** statement. |

## Statement Labels

A statement label is a unique identifier that identifies a particular program line. It consists of a string of numeric or alphanumeric characters at the beginning of a source line. Source lines do not require statement labels. If the program is directed to a statement label with the **GOTO**, **GOSUB** or **INPUTTRAP** statements, however, the label must exist somewhere in the program.

A numeric statement label can be any constant number (decimals allowed). Numeric statement labels can end with a colon (**:**), but a colon is not mandatory. An alphanumeric statement label must begin with a letter and be followed by any combination of letters, numbers, periods, or dollar signs. An alphanumeric statement label must be followed by a colon, or it is not recognized as a statement label.

## Writing Readable Code

An mvBASIC program should be made relatively easy to read, both for the programmer and for those who must maintain the program. The readability of a program can be greatly enhanced by:

* Blank spaces to indent sections of code.
* Blank lines to group sections of code together.
* Recognizable variable and subroutine names.
* Comments or remarks to document a program.

When blank spaces or lines that are not part of a data item appear in a program line, they are ignored. Therefore, blanks can be used freely in order to improve the appearance and readability of a program. The programmer can use blanks to indent sections of code and make the program structure more apparent. Blank lines can also be used to set apart a subroutine or any other significant part of the program.

The programmer should make a habit of assigning recognizable names to variables and constants. It becomes much easier to keep track of what the variable signifies if variable names are kept coherent—for example, an array containing customer names and addresses would be easier to identify if it were called CUSTOMERS rather than **X**.

### Using Remarks

Program documentation includes comments in the mvBASIC program that explain or identify various parts of the program. Comments are part of the source code only (the original program), and as such they are not executable. They do not substantially affect the size of the object code.

Comments must begin with one of these symbols:

|  |
| --- |
| REM  \*  ! |

To place a comment on the same physical line as another statement, the first statement must first be ended with a semicolon (**;**), as in this example:

|  |
| --- |
| IF 2INT\_SUM < 0 THEN  LOSS =2INT\_SUM ;  \*CORRECTLY FORMATTED COMMENT  END ELSE PROFIT = 2INT\_SUM |

Comments cannot be placed between multiple statements on one physical line. For example, in the second line of the following example all the text following the \* symbol is treated as part of the comment and is not executed:

|  |
| --- |
| IF 2INT\_SUM < 0 THEN  LOSS= 2INT\_SUM ;  \*THE REST OF THIS LINE IS IGNORED ;  END ELSE |

Comments can, however, be placed in the middle of a statement that occupies more than one physical line, as in this example:

|  |
| --- |
| IF 2INT\_SUM < 0 THEN  LOSS = 2INT\_SUM  \* THIS COMMENT IS ON A LINE OF ITS OWN  END ELSE PROFIT = 2INT\_SUM |

Remarks in the Object Code

A special form of comment can be used to embed a comment directly into the object code. A statement beginning with **$\*** places the following text into the object code created when the program is compiled. For example:

|  |
| --- |
| $\* "OPERATING SYSTEM VERSION 2.5" |

Comments in the object code are particularly useful for including the version number of the program, or for entering copyright information.

# Constants, Variables, and Data Types

Constants are values that remain unchanged throughout program execution.

## Assigning and Using Constants

Constants can be used directly, or they can be assigned a name with the assignment (**=**) or **EQUATE** statement.

For assigning a constant, the **EQUATE** statement is preferable since there is nothing to stop a constant assigned with **=** (equal sign) from being changed later in the program’s execution. A constant assigned with **EQUATE**, on the other hand, can never be changed: if **EQUATE** is used, the programmer is ensured that a constant will remain a constant.

The **EQUATE** statement is also more efficient, since a constant assigned with **=** needs to be reassigned each time the program is executed.

Example

|  |
| --- |
| PRINT "HELLO, WORLD" |

The string **"HELLO, WORLD"** is a constant string value used directly in the **PRINT** statement. Alternatively, the program might have read:

|  |
| --- |
| EQUATE GREETING TO "HELLO, WORLD"  PRINT GREETING |

By assigning the name **GREETING** to the constant string **HELLO WORLD**, it can be accessed by that name any time later in the program.

## Assigning and Using Variables

Variables are symbolic names that represent stored data values and can change in value during program execution. The value can be explicitly assigned by the programmer, can be read as input, or can be the result of operations performed by the program during execution.

At the start of program execution, all variables are set to an unassigned state. Any attempt to use a variable in the unassigned state produces an error message, and a value of 0 is assumed.

Names for both variables and constants must begin with an initial alphabetic character. They can also include one or more digits, letters, periods, or dollar signs. (Note that hyphens and underscores are not valid in a variable name.) Uppercase and lowercase are interpreted differently. A variable name can be any length, but it cannot be the same as any reserved word.

## Data Typing in mvBASIC

In many other programming languages, such as Pascal and PL/I, a distinction is made among types of data. In these languages, all constants, variables, and their data types (integer, real, string, character, etc.) have to be declared at the beginning of the program so that the compiler will know how to store the data. Furthermore, the size of the variable often has to be declared so that the compiler will know how much space to set aside.

In mvBASIC, on the other hand, no data typing is made by the compiler: all data typing is made at run time, by context. A variable can therefore alternate between numeric and string values within the program. The only thing to be careful of is that when string values are assigned in the program text, they must be delimited by single quotes (**'**), double quotes (**"**), or backslashes (**\**). Otherwise, they are assumed to be variable names.

There is, of course, a difference between the way a numeric value and a string value can be treated: it is unreasonable to expect a program to take the square root of the string **CARL**. In such a situation, however, a fatal error will not occur—when a string value is applied to a numeric function, a value of 0 is assumed, a warning message is printed, and the program continues from there. String operations, on the other hand, can be executed on numeric values without conflict.

The advantage to no data typing is obvious; less work for the programmer and more flexibility for the program. The disadvantage is that errors which one might expect the compiler to detect are not caught. For example, if a variable name is misspelled, the compiler will simply assume that it is a new variable, and the program will successfully compile without an error or warning. Similarly, if a string variable containing **CARL** were accidentally used in the **SQRT** function, the programmer would not find out until the program was executed.

# Building Expressions

The assignment statement is used in mvBASIC to assign values to variables.

The following topics are presented in this section:

* [**Simple Assignment**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/simple_assignment.htm)
* [**Using Operators and Functions**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_operators_and_functions.htm)
* [**Numeric Expressions**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/numeric_expressions.htm)
* [**String Expressions**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/string_expressions.htm)
* [**Logical Data (Booleans)**](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/logical_data_(booleans).htm)

## Simple Assignment

There are several forms of the assignment statement, but its most common use is of the form:

Format

|  |
| --- |
| *variable* = *expr* |

|  |  |
| --- | --- |
| ***variable*** | Variable name. |
| ***expr*** | Valid expression. An expression is a value that is evaluated at the time of execution and can be anything from a simple constant to a complex sequence of variables, operators, and functions. |

For a simple example, to assign the variable **NUMBER** to the constant number 4:

|  |
| --- |
| NUMBER = 4 |

Similarly, to assign the variable **NAME** to the constant string **FRED**:

|  |
| --- |
| NAME = "FRED" |

## Using Operators and Functions

In the preceding examples, the assignment is made to a simple constant. However, any valid expression can be used instead. A simple expression might be a variable name combined with an operator and a constant.

|  |  |
| --- | --- |
| **Operators** | Perform mathematical, string, and logical operations on two surrounding values. |
| **Operands** | Surrounding values on which specified operations are performed. |

Example

To assign the variable **NUMBER2** to be **NUMBER** plus 1:

|  |
| --- |
| NUMBER2 = NUMBER + 1 |

In this example, **+** is the operator, and **NUMBER** and **1** are the operands. **NUMBER** is interpreted as a variable name, and **1** is interpreted as a numeric constant. If **NUMBER** contains 4, then after the above statement, **NUMBER2** will contain 5.

Another simple expression might involve an intrinsic function.

|  |  |
| --- | --- |
| **Functions** | Perform mathematical, string, and logical operations on a value passed within parentheses. |

For example, to assign the variable **ROOT** to the square root of **NUMBER**:

|  |
| --- |
| ROOT = SQRT(NUMBER) |

In this example, **SQRT** is the function and **NUMBER** is the value passed to it. If **NUMBER** contains 4, then after the above statement, **ROOT** will contain 2.

Multiple operators and functions can be combined in an expression to evaluate to a single value. For example, to assign **NUMBER3** to be 1 plus the square root of **NUMBER**:

|  |
| --- |
| NUMBER3 = SQRT(NUMBER) + 1 |

After the above statement, **NUMBER3** will contain 3. Note that this is different from:

|  |
| --- |
| NUMBER3 = SQRT(NUMBER + 1) |

which will return into **NUMBER3** the square root of 5, or 2.236.

Valid expressions can therefore be as simple as a constant or a single variable name, or they can consist of multiple operations to be evaluated at run time.

## Numeric Expressions

Numeric data is represented as a sequence of digits (0 – 9) with an optional decimal point. A leading plus (**+**) or minus (**-**) sign might be used, but commas are not allowed. Any data containing any characters other than numbers and a single decimal point will be interpreted as a string. Some examples of numeric values are:

-34

42368.99

+3.1416

Numeric data can contain up to 19 digits, including a maximum of 9 decimal positions. See the PRECISION statement in [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) for information on how to set the maximum number of fractional digits.

### Arithmetic Operators

Arithmetic operations range from the simplest calculations (such as **COST = COST + 5**) to complex expressions combining trigonometric and logarithmic functions. In general, when several arithmetic operations are used in one expression, they follow accepted mathematical guidelines to precedence. The arithmetic operators available to mvBASIC, in order of precedence, are listed as follows:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Sample Expression** |
| **+** | Unary plus | **+COST** |
| **-** | Unary minus | **–COST** |
| **\*** | Multiplication | **COST \* EXPENSES** |
| **/** | Division | **COST / EXPENSES** |
| **+** | Addition | **COST + EXPENSES** |
| **-** | Subtraction | **COST – EXPENSES** |

In cases where operators equivalent in precedence (such as **\*** and **/**) are used, the order of evaluation follows left to right.

### Parentheses in Expressions

The order of evaluation can be changed by using parentheses. Operations on expressions enclosed in parentheses are performed before the others.

Example

|  |
| --- |
| (14 \* 8) + 12 / 2 + 2 |

In the example above, expression is evaluated as 112+6+2 or 120. On the other hand, the following arithmetic expression is evaluated as 14\*20/4 or 70:

|  |
| --- |
| 14 \* (8 + 12) / (2 + 2) |

In arithmetic expressions parentheses must be placed correctly in order to obtain the desired result.

### Character Strings in Arithmetic Expressions

If a character string variable that evaluates to a number is used within an arithmetic expression, the character string is treated as a numeric variable. That is, the numeric character string is converted to its equivalent internal number and then evaluated numerically within the arithmetic expression.

|  |
| --- |
| 55 + "22" |

The preceding example is evaluated as 77. If a character string variable that does not evaluate to a number is used within an arithmetic expression, a warning message is displayed and the string is treated as zero.

The following example expression is evaluated as 55:

|  |
| --- |
| 55 + "TWENTY TWO" |

A message displays, warning that the data is nonnumeric, resembling the following:

|  |
| --- |
| [B16] LINE 16 NON-NUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED! |

### Intrinsic Mathematical Functions

An intrinsic function is a built-in mvBASIC function to be used on numeric operands. The following is a list of the mathematical functions available in mvBASIC:

|  |  |
| --- | --- |
| **Function** | **Description** |
| **ABS** | Computes the absolute value of a given arithmetic expression. |
| **COS** | Returns the cosine value of the angle given in the expression. |
| **EXP** | Returns an exponential value that will raise the base number e (2.7183) to the value of the expression. |
| **INT** | Truncates the decimal portion of a given arithmetic expression and returns the integer value. |
| **LN** | Generates the natural logarithm (log base e) of the given expression. |
| **PWR** | Raises the value of an expression to the power denoted by a second expression. |
| **REM** | Divides an expression by another, and returns the remainder value only. |
| **RND** | Generates a random number within the range of 0 and the value of the expression minus 1. |
| **SIN** | Returns the sine value of the angle given in the expression. |
| **SQRT** | Computes the square root of any positive numeric expression. |
| **TAN** | Returns the tangent value of the angle given in the expression. |

See [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) for full information on the syntax and behaviour of these functions.

# String Expressions

String data consists of a sequence of ASCII characters. They can represent either numeric or nonnumeric information, and are limited in length to 248K.

Character string constants consist of a sequence of ASCII characters enclosed in apostrophes (**'**), double quotation marks (**"**), or backslashes (**\**). Some examples of character string constants are:

"EMILY DANIELS"

'$42,368.99'

'NUMBER OF EMPLOYEES'

"34 CAIRO LANE"

\"FRED'S PLACE" ISN'T OPEN\

The beginning and terminating delimiters must match. In other words, if you begin a string with a single quotation mark, you must use a single quotation mark to terminate the string. If one of the delimiters is used within the character string, a different delimiter must be used to begin and terminate the string. For example, using apostrophes to enclose the following string is incorrect:

|  |
| --- |
| 'IT'S A LOVELY DAY.' |

Instead, the string should be delimited with double quotes (or backslashes), as follows:

|  |
| --- |
| "IT'S A LOVELY DAY." |

Two adjacent identical delimiters specify a null, or empty, string. Any ASCII character can be used in character string data except the ASCII character 10 (carriage return), which is used to separate the logical lines of a program.

|  |
| --- |
| The CAT String Operator |

String expressions can be concatenated, or linked, by using the concatenation operator (**:** or **CAT**) as follows:

|  |
| --- |
| NAME = FIRST : LAST |

or

|  |
| --- |
| NAME = LAST CAT ", " CAT FIRST |

If, for instance, the current value of **FIRST** is **JANE** and the current value of **LAST** is **GREY**, the preceding string expressions have the values:

|  |
| --- |
| " JANE GREY "  " GREY, JANE " |

Multiple concatenation operations are performed from left to right. Expressions in parentheses are evaluated before other operations are performed.

All operands in concatenated expressions are considered to be string values regardless of whether they are string or numeric expressions. However, the priority of arithmetic operators is higher than the concatenation operator. If both types of operator appear in the same expression, arithmetic operations are performed first.

For example:

|  |
| --- |
| "JANE IS" : "2" + "2" : "3" : "YEARS OLD." |

has the value:

|  |
| --- |
| "JANE IS 43 YEARS OLD." |

## Logical Data (Booleans)

All data has a logical (or Boolean) value, which is to say that it can be computed as true or false. If the data contains only numeric values and the numeric value is zero (0), it is false; any other numeric value is true. If the data contains character string values other than the null string (**""**), it is true; the null string is false. Logical values are used for testing conditionals. If a statement reads:

|  |
| --- |
| IF FOUND THEN... |

the variable **FOUND** is tested to see if it is not null or zero. If it is not, then the condition is determined to be true, and the statements following the **THEN** clause are executed.

### Relational Operators

Relational operators are used to compare both numeric and character string data. The result of the comparison, either true (1) or false (0), can be used for conditional statements. The relational operators are:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Relation** | **Example** |
| **EQ** or **=** | Equality | **X = Y** |
| **NE** or **#** | Inequality | **X # Y** |
| **><** or **<>** | Inequality | **X <> Y** |
| **LT** or **<** | Less than | **X < Y** |
| **GT** or **>** | Greater than | **X > Y** |
| **LE** or **<=** or **=<** | Less than or equal to | **X <= Y** |
| **GE** or **>=** or **=>** or **#<** | Greater than or equal to | **X >= Y** |

When arithmetic and relational operators are both used in a single expression, the arithmetic operation is always performed first.

The same relational operators can be applied to both numeric and string data, but the operations will be calculated differently according to the type of data in the operands. Relational numeric comparisons are calculated as expected, by comparing the literal value of the operands. String comparisons, however, are made by comparing the ASCII values of single characters from each string.

In string comparisons, characters are compared from left to right, and the first string to yield a higher numeric ASCII code equivalent is considered to be greater. If all of the ASCII codes are the same, the strings are considered equal. If the two strings have different lengths but the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered greater than the shorter string. Note that leading and trailing blank spaces are significant, since the space character has an ASCII value of 032.

If both string values can be converted to numeric, then the comparison is always made numerically. If only one operand is numeric, the comparison will be made as if both were string values.

The following string comparisons are true and return a value of 1:

|  |
| --- |
| "AA" < "AB"  "FILENAME" = "FILENAME"  "FILENAME" < "NAMEFILE"  "CL  " > "CL"  "KG" > " KG"  "8/12/78"< "9/12/78" |

### Logical Operators

Logical operators perform Boolean operation tests on logical expressions. They have the lowest precedence among all operators: they are evaluated after all other operators have been evaluated.

The two forms of logical operation in mvBASIC are:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Syntax** | **Definition** |
| **AND** | **x AND y** | True (evaluates to 1) if both **x** and **y** are true. |
| **&** | **x & y** | True (evaluates to 1) if both **x** and **y** are true. |
| **OR** | **x OR y** | True (evaluates to 1) if either **x** or **y** is true. |
| **!** | **x ! y** | True (evaluates to 1) if either **x** or **y** is true. |

The **NOT** function can be used to invert a logical value.

For example:

|  |
| --- |
| IF FOUND AND QUIT = "Y" THEN... |

The variable **FOUND** is tested to see if it is not null or zero, and is evaluated as true if it is not. Then the relational expression **QUIT = "Y"** is evaluated. If both **FOUND** and **QUIT = "Y"** are evaluated as true, then the condition as a whole is evaluated as true and the statements following the **THEN** clause are executed.

### The MATCH Operator

The pattern matching operator, **MATCH** or **MATCHES**, can be used to compare a string expression to a pattern specification and return a value of 1 if they match. The syntax for a **MATCH** operation is:

|  |
| --- |
| *expr* MATCH *pattern-expr* |

The pattern is a general description of the format of the string and can be specified as a constant or as an expression. The pattern specification codes and their definitions are as follows:

|  |  |
| --- | --- |
| **Pattern** | **Definition** |
| ***n*N** | ***n*** numeric characters. |
| ***n*A** | ***n*** alphabetic characters. |
| ***n*X** | ***n*** wildcards (any character). |
| ***string*** | Any literal string. |

*n* must be a whole number. If ***n*** is 0, the relation is true only if all the characters match the specified type. (Note that the null string (**""**) matches **0N**, **0A**, and **0X**.)

For example,

|  |
| --- |
| ZIP.CODE MATCH "0N" |

will be true only if all the characters in the string **ZIP.CODE** are digits. Patterns can be combined in any sequence.

For example,

|  |
| --- |
| IF LICENSE MATCHES "3N'-'3A" THEN... |

confirms that a license number entered consists of 3 digits, a dash, and 3 alphabetic letters.

### Logical Functions

Logical functions are functions that return a value of 0 or 1. The following are the logical functions available in mvBASIC:

|  |  |
| --- | --- |
| **Function** | **Description** |
| **ALPHA** | Tests the given expression for an alphabetical value. |
| **NOT** | Returns the logical inverse of a given expression. |
| **NUM** | Tests the given expression for a numeric value. |

See [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) for a full description of the syntax and behaviour of these functions.

# Overview of mvBASIC Statements and Functions

This section is designed to give a brief topical overview of the statements and functions in mvBASIC. (For full descriptions of the statements and functions, see [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm).) The topics of this section are covered in the following order:

* [Assignment Statements](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/assignment_statements.htm)
* [Intrinsic Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/intrinsic_functions.htm)
* [Internal Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/internal_program_control.htm)
* [External Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/external_program_control.htm)
* [Sending Output to the Screen and Printer](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sending_output_to_the_screen_and_printer.htm)
* [Terminal Input](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/terminal_input.htm)
* [Dynamic Array Processing](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dynamic_array_processing.htm)
* [Generalized String Processing](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/generalized_string_processing.htm)

[Dimensioned Arrays](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dimensioned_arrays2.htm)

[Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm)

[Reading and Writing Tapes or Floppy Disks](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_writing_tapes_or_floppy_disks.htm)

[Communications](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/communications.htm)

[Execution Locks](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/execution_locks.htm)

[Compiler Directives](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/compiler_directives.htm)

[Miscellaneous Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/miscellaneous_statements_and_functions.htm)

[The Error Message Processor](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/the_error_message_processor.htm)

# Assignment Statements

The simple assignment statement in mvBASIC is of the form ***var*** **=** ***expr***. A full list of operators is given in [Format, Data, and Expressions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/format,_data_and_expressions.htm). For example, the variable NUMBER can be set to the value 7 with:

|  |
| --- |
| NUMBER = 7 |

Any valid expression can be used in an assignment statement. For example, **NUMBER2** can be assigned the value of **NUMBER** plus **2** with:

|  |
| --- |
| NUMBER2 = NUMBER + 2 |

mvBASIC also supports operator assignment. For example, **2** can be added to the value of **NUMBER** with:

|  |
| --- |
| NUMBER += 2 |

as a shorthand for:

|  |
| --- |
| NUMBER = NUMBER + 2 |

Accepted operators for operator assignment are:

|  |  |
| --- | --- |
| **+** | Addition |
| **-** | Subtraction |
| **\*** | Multiplication |
| **/** | Division |
| **:** | Concatenation |

Initializing Variables (CLEAR)

The **CLEAR** statement acts to initialize all variables to the value 0. It cannot be used to initialize a single variable, however, and initializing all variables to 0 may result in errors due to unassigned variables remaining undiscovered.

Assigning Constants (EQUATE)

The **EQUATE** statement is used to make a variable functionally equivalent to another or to assign a constant. It cannot be used to assign a variable, since values assigned with an **EQUATE** statement cannot be reassigned during the program.

The **EQUATE** statement assigns values at compile time. No operators or functions can be incorporated into an **EQUATE** statement, with the exception of the **CHAR** function. Thus, the **EQUATE** statement can be used to supply a meaningful name for a special character in a program or for an element of a dimensioned array, for example, to equate **AM** to an attribute mark:

|  |
| --- |
| EQUATE AM TO CHAR(254) |

or to equate **QTY** to element **4** of the dimensioned array **INVENTORY**:

|  |
| --- |
| EQUATE QTY TO INVENTORY(4) |

# Intrinsic Functions

An intrinsic function is a built-in mvBASIC function to be used on numeric operands.

Numeric Functions

In addition to the standard numeric operators (**+**, **-**, **\***, **/**), mvBASIC provides several functions for evaluating numeric calculations.

|  |  |
| --- | --- |
| **ABS** | Returns the absolute value of a given expression. The absolute value of a number is its positive value, or the difference between itself and zero. |
| **INT** | Gives the integer value of an expression. It truncates the decimal portion of a number and returns the result. |
| **REM** | Takes two arguments and returns the remainder value when the first expression is divided by the second. |
| **SQRT** | Returns the square root of a positive expression. |
| **RND** | Returns a random number between 0 and the given expression minus 1. |
| **PWR** | Takes two arguments and returns the first value to the power of the second. |

In addition, the following trigonometric functions are available in mvBASIC:

|  |  |
| --- | --- |
| **SIN** | Returns the sine of the angle. |
| **COS** | Returns the cosine of the angle. |
| **TAN** | Returns the tangent of the angle (**SIN**/**COS**). |
| **LN** | Returns the natural logarithm (log base e) of the expression. |
| **EXP** | Returns e to the power of the expression (the inverse of **LN**). |

The accuracy of each numeric function is dependent on the decimal precision used by the program; i.e., the number of decimal places to which numeric values are calculated. By default, all numeric values are calculated to four decimal places. To reassign this value, use the **PRECISION** statement. The maximum precision supported by mvBASIC is 9.

Logical Functions (NOT, NUM, ALPHA)

A logical function, or Boolean function, is one which returns either 0 or 1. A return value of 0 is taken to mean false, and a return value of 1 is taken to mean true. Logical functions are most useful in conditional statements (**IF**, **CASE**), or in the exit for loops.

In addition to the logical operators (**=**, **<>**, **>**, **>=**, **<**, **<=**, **MATCH**), the following intrinsic logical functions are supported in mvBASIC:

|  |  |
| --- | --- |
| **NOT** | Returns the logical inverse of a given expression. That is, if the expression evaluates to 0 or the null string (**" "**), the **NOT** function returns 1; if the expression evaluates to anything other than 0 or the null string, the **NOT** function returns 0. |
| **NUM** | Returns 1 if the given expression is numeric, or 0 if it is nonnumeric. (Note that the NUM function might return 0 for a clearly numeric value if it contains more decimal places than the current precision.) |
| **ALPHA** | Returns 1 if the given expression is alphabetic, or 0 if it is non alphabetic. |

Example

Suppose a program expects a positive number in Attribute **3** of a file item. The source code might read:

|  |
| --- |
| PRICE = RECORD<3>  IF NOT(NUM(PRICE)) THEN  PRINT "ERROR — NON-NUMERIC DATA IN ATTRIBUTE 3."  STOP  END ELSE  IF PRICE < 0 THEN  PRINT "ERROR — NEGATIVE DATA IN ATTRIBUTE 3."  STOP  END  END |

Using the Boolean operators (**AND**, **OR**), the same code might read:

|  |
| --- |
| PRICE = RECORD<3>  IF NOT(NUM(PRICE)) OR PRICE < 0 THEN  PRINT "ERROR IN ATTRIBUTE 3 — ":  PRINT "POSITIVE NUMBER EXPECTED."  STOP  END |

# Internal Program Control

Statements in an mvBASIC program are executed in the order in which they appear in the source code. Program control statements are those which can be used to alter that sequence. This section discusses the internal program control constructs that do not involve other programs, external subroutines, or TCL commands. More advanced program control statements are discussed later in this section.

## The IF Conditional

The IF construct is used to execute a statement (or series of statements) if a condition has a logical value of true, and (optionally) a different set of statements if the condition has a logical value of false.

## The THEN and ELSE Clauses

Either a **THEN** or **ELSE** clause (or both) must be supplied with **IF**. The syntax of the **THEN** and **ELSE** clauses is important to understand, because they are used not only in **IF** statements, but also in numerous file I/O, tape I/O, floppy disk I/O, and communication statements.

**THEN** and **ELSE** clauses can be written on the same statement line in the following manner:

|  |
| --- |
| IF NET >= 0 THEN PRINT "PROFIT IS ": ELSE PRINT "LOSS IS ": PRINT ABS(NET) |

If there are multiple conditional statements for either the **THEN** or **ELSE** clause, they can be separated by semicolons (**;**). For example:

|  |
| --- |
| IF NET >= 0 THEN FLAG = 1; PRINT "PROFIT IS ": ELSE FLAG = 0; PRINT "LOSS  IS ": PRINT ABS(NET) |

However, the **IF** statement becomes difficult to read if the **THEN** and **ELSE** clauses are written on a single line. It is preferable to write it on several lines, even if the conditional statement is very short.

When splitting an **IF** statement onto several lines, the **THEN** or **ELSE** keyword must end a program line, with the conditional statements beginning on the next. At the end of the conditional statements, an **END** statement must be used to group them together.

Thus, if the condition tested by **IF** is true, all statements between the **THEN** clause and the corresponding **END** statement are executed; otherwise, all statements between the **ELSE** clause and the corresponding **END** statement are executed.

So the above lines of a program printing out profit or loss on a transaction might read:

|  |
| --- |
| IF NET >= 0 THEN  FLAG = 1  PRINT "PROFIT IS " :  END ELSE  FLAG = 0  PRINT "LOSS IS " :  END  PRINT ABS(NET)  NULL Statements |

**NULL** statements are often included in **THEN** or **ELSE** clauses as placeholders. A **NULL** statement performs no action; however, a programmer can use **NULL** statements to make the logic of a conditional somewhat clearer: for example, if a programmer wanted to test if a value was numeric, he might write:

|  |
| --- |
| IF NUM(PRICE) THEN  NULL  END ELSE  PRINT "ERROR: NON-NUMERIC PRICE. STOP"  STOP  END |

There are ways of doing this without using a **NULL** statement (by using the **NOT** function in the condition, or by omitting the **THEN** clause entirely). However, a programmer might prefer to use **NULL** statements to make conditionals easier to read.

## CASE Constructs

The **CASE** construct acts to perform multiple **IF** conditionals. It tests several conditions until one returns a value of true. It then executes the associated set of statements.

A **CASE** construct must begin with a **BEGIN CASE** statement and end with an **END CASE** statement. In between, each **CASE** statement tests a single condition, and, if true, the statements between the current **CASE** and the next **CASE** are executed. The program then jumps to the position after the **END CASE** statement, ignoring all remaining **CASE** statements in that group.

For example, the above profit-or-loss example might read:

|  |
| --- |
| BEGIN CASE  CASE NET > 0  PRINT "PROFIT IS " : NET  CASE NET < 0  PRINT "LOSS IS " : ABS(NET)  CASE NET = 0  PRINT "NO PROFIT OR LOSS ON THIS TRANSACTION."  END CASE  Loops (LOOP, FOR) |

Program loops are constructs that repeat the same sequence of statements while a condition holds true or until a condition is met.

## The LOOP Construct

The **LOOP** statement is the general-purpose looping construct in mvBASIC. It has (optionally) two sets of statements, the first of which is executed before testing the condition clause, and the second of which is executed only if the condition is verified. The condition is written as either a **WHILE** or an **UNTIL** clause. If the **WHILE** clause is used, the loop will continue if the condition is still true; if the **UNTIL** clause is used, the loop will continue if the condition is still false.

The following example will continue to prompt for a number until a numeric value is entered:

|  |
| --- |
| LOOP  PRINT "ENTER A NUMBER":  INPUT NUMBER  UNTIL NUM(NUMBER) DO  PRINT "NUMERIC INPUT EXPECTED!"  REPEAT |

The **UNTIL** clause can be replaced with a **WHILE** by negating the condition:

|  |
| --- |
| WHILE NOT(NUM(NUMBER)) DO |

### FOR Loops

In its simplest form a **FOR** loop performs a set of statements while incrementing a number by 1. When the number reaches or surpasses a specified maximum, the **FOR** loop exits. For example, a program printing out the first ten perfect squares might read:

|  |
| --- |
| FOR I = 1 TO 10     PRINT I \* I  NEXT I |

In mvBASIC, the **FOR** loop has been enhanced in two ways: an increment other than 1 can be specified with the **STEP** clause, and the **WHILE** and **UNTIL** clauses of the **LOOP** statement have been incorporated into it.

## Stopping a Program (STOP, ABORT, END)

Two statements will cause an immediate stop to program execution: the **STOP** statement and the **ABORT** statement. The difference between them is that if the current program is called by a Proc or another program, the **STOP** statement will return to the calling Proc or program, but an **ABORT** statement will return directly to TCL.

In general, **STOP** statements are used for a normal or nonfatal termination of a program, and an **ABORT** statement is used for abnormal termination.

**STOP** and **ABORT** are often used in **ELSE** clauses to file I/O, tape I/O, floppy disk I/O, or communication statements, when the program becomes pointless if the statement fails. **STOP** statements are also used between the main part of a program and its internal subroutines. When a program is written with internal subroutines at the end, a **STOP** statement is necessary to ensure that the subroutines are not directly executed at the end of the program.

## The END Statement

Beyond its function for delimiting **THEN** or **ELSE** statements, the **END** statement is also used to designate the end of compilation. When the compiler reaches an **END** statement that does not correspond to a **THEN**, **ELSE**, or **LOCKED** clause, all compilation stops. Any statements or subroutines that come after the **END** statement in the source code will be ignored.

## Internal Subroutines (GOSUB, RETURN)

An internal subroutine is a discrete sequence of statements starting with a statement label and ending with a **RETURN** statement. In the source code, subroutines are placed after the main part of the program, and precautions are generally taken to ensure that they are never executed directly. Internal subroutines are executed by **GOSUB** statements, which point to the statement label.

Example

A subroutine labelled **REPORT** prints a report of the session’s transactions. The **REPORT** subroutine might call the following code:

|  |
| --- |
| PRINT "PRINTING A REPORT ..."  GOSUB REPORT  DISPLAY "REPORT PRINTED."  The REPORT subroutine might read:  REPORT:  PRINTER ON  PRINT "NUMBER OF TRANSACTIONS" ,  NO.OF.TRANS  .  .  .  PRINTER CLOSE  PRINTER OFF  RETURN |

When the **GOSUB** is executed, program control will transfer to the statement label **REPORT** and will continue until the **RETURN** statement is encountered. Program control will then continue with the statement following the **GOSUB**, and the message **REPORT PRINTED** will be displayed to the screen.

The GOTO Statement

The **GOTO** statement is often grouped together with **GOSUB**, because they share the same syntax and perform similar functions. However, the **GOTO** statement serves only to transfer program execution to the statement label and will never return unless another **GOTO** is used.

# External Program Control

The **CALL** statement transfers execution to an external subroutine.

## External Subroutines (CALL, SUBROUTINE, RETURN)

An external subroutine is a sequence of statements that performs a discrete function, compiled separately from the calling program. Unless the subroutine is in the same file as the calling program, it must be cataloged in the account before being called.

The first statement of the subroutine must be the **SUBROUTINE** statement, and the last statement executed must be the **RETURN** statement. The **SUBROUTINE** statement can take several parameters that correspond to the parameters on the **CALL** statement that calls the subroutine. The ***n***th parameter on the **SUBROUTINE** statement and the ***n***th parameter on the **CALL** statement become equivalent.

Example

Suppose a simple subroutine named **ADDEMUP** is called with the following source lines:

|  |
| --- |
| PRINT "ENTER A NUMBER":  INPUT NUMBER1  PRINT "ENTER ANOTHER NUMBER":  INPUT NUMBER2  CALL ADDEMUP(NUMBER1, NUMBER2, NUMBER3)  PRINT NUMBER1 : " PLUS " : NUMBER2 : " IS " : NUMBER3 |

and the subroutine **ADDEMUP** reads:

|  |
| --- |
| SUBROUTINE ADDEMUP(A,B,C)  C = A + B  RETURN |

The value of **NUMBER1** is passed to the variable **A** in the subroutine, the value of **NUMBER2** is passed to **B**, and the value of **NUMBER3** is passed to **C**. At the conclusion of the subroutine, the parameters are returned with their new values (if any). Thus, **ADDEMUP** serves to place the sum of the first two numbers in the variable **NUMBER3**.

## Passing Parameters (COMMON)

The alternative to passing parameters with the **CALL** and **SUBROUTINE** lines is the **COMMON** area, by which several programs can share the same variables.

The **COMMON** statement permits multiple programs and subroutines to use the same variables by accessing them according to the sequence in which they are stored. Each program using the **COMMON** area must include a **COMMON** statement, and the variables will be considered equivalent according to their positions. For example, in the simple example of a subroutine shown earlier in this section, the main program might have read:

|  |
| --- |
| COMMON NUMBER1, NUMBER2, NUMBER3  PRINT "ENTER A NUMBER":  INPUT NUMBER1  PRINT "ENTER ANOTHER NUMBER":  INPUT NUMBER2  CALL ADDEMUP  PRINT NUMBER1 : " PLUS " : NUMBER2 : " IS " : NUMBER3 |

and the subroutine **ADDEMUP**:

|  |
| --- |
| SUBROUTINE ADDEMUP  COMMON A, B, C  C = A + B  RETURN |

The variable **NUMBER1** in the main program and the variable **A** in the subroutine are considered equivalent because of their positions in the **COMMON** statement. The same is true of **NUMBER2** and **B**, and of **NUMBER3** and **C**.

## Executing a TCL Command (EXECUTE, CHAIN, DATA)

The **CHAIN** and **EXECUTE** statements can both be used for executing a TCL command. **EXECUTE** is by far the more powerful of the two statements, and is preferred to **CHAIN**.

The **EXECUTE** statement executes any TCL command and returns to the current program. In addition, the **RETURNING** clause can be used to determine error messages which may have resulted, and the **CAPTURING** clause can be used to capture the terminal output generated by the command.

The **CHAIN** statement will execute the command but will not return to the calling program.

## The DATA Statement

The **DATA** statement places data in the secondary output buffer, or data stack. If the data stack is not empty, any subsequent requests for input will accept the response directly from the data stack, and the user will not be given the opportunity to respond.

The data stack is helpful for executing TCL commands that request information that the program can supply. For example, if a programmer wishes to copy a file item before altering it in a program, the **COPY** command might be used (rather than writing a new item with the **WRITE** statement). The **COPY** command requests the operator to supply the new item-ID, so the **DATA** statement could be used to store the new item-ID on the data stack before using **EXECUTE** to run the **COPY** command.

Before **EXECUTE** returns to the calling program, it checks the data stack for input. If the data stack is not empty, its contents will be sent to TCL. Any data left in the data stack will be cleared upon exit from **EXECUTE**.

## Using EXECUTE with Select-lists

Although the mvBASIC language has a **SELECT** statement for creating a select-list, the **EXECUTE** statement can be used to run one of the INFO/ACCESS select-list generators (e.g., **SELECT**, **SSELECT**, **QSELECT**). The INFO/ACCESS select-list generators are often preferable to the **SELECT** statement because they can include selection expressions. Selection expressions cannot be supplied with the mvBASIC **SELECT** statement.

A select-list generated by **EXECUTE** will be placed in the external select-list variable. The select-list will be unavailable to subsequent **EXECUTE** statements; however, the **DATA** statement can be used to stack a **SAVE-LIST** command to be executed before returning to the program, and a future **EXECUTE** statement can be used to run a **GET-LIST** to retrieve it. See [Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm) later in this section for more information on select-lists.

## Executing Another mvBASIC Program (ENTER)

To execute another mvBASIC program, the **EXECUTE** statement is recommended. However, there is an **ENTER** statement which acts only to execute another mvBASIC program and then exit without returning to the calling program.

## CAP-HUSH-ON and CAP-HUSH-OFF Commands

The **CAP-HUSH-ON** command turns off the display of captured output for the issuing process. The **CAP-HUSH-OFF** command turns on the display of captured output for the issuing process. Both commands are most effectively used within an **EXECUTE** statement.

## Executing a Windows Command Line Command

Windows command line commands can be executed by prefixing them with the bang (**!**) sign in an **EXECUTE** statement. The output from the executed command can be stored in a variable by use of the **CAPTURING** clause. For example:

|  |
| --- |
| EXECUTE "! dir /w " CAPTURING DirectoryList |

This example does a wide directory listing of the current directory using the Windows **DIR** command and puts the results into the mvBASIC variable DirectoryList.

# Sending Output to the Screen and Printer

There are two standard output devices available to an mvBASIC program:

* Terminal screen
* Printer

## Output Devices (PRINT, CRT, DISPLAY)

The **CRT** and **DISPLAY** statements send output only to the terminal screen, and the **PRINT** statement sends its output either to the terminal screen or to the printer, depending on which has been selected as the output device. The syntax of all three statements is identical, except that the **PRINT** statement accepts the ON keyword for multiple print units.

In a broader sense, file items and attached tape devices or floppy disk devices can also be considered output devices. See [Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm) and [Reading and Writing Tapes or Floppy Disks](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_writing_tapes_or_floppy_disks.htm) later in this section for information on file I/O, tape I/O, and floppy disk I/O.

## Sending Output to the Printer (PRINTER)

The **PRINT** statement by default sends output to the screen. There are two ways, however, to force the **PRINT** statement to send output to the printer: by the P option to the RUN command, or by the **PRINTER ON** statement. The **PRINTER ON** statement signifies that all subsequent **PRINT** statements will send output to a Spooler print unit. At the end of the program, the print unit will be sent to the Spooler.

The **PRINTER OFF** statement returns to the default condition: all **PRINT** statements after a **PRINTER OFF** statement will send output to the terminal screen again. To print output before the end of the program, the **PRINTER CLOSE** statement is available to send everything in the print unit directly to the Spooler.

## Print Units

When output is being sent to a printer, the **ON** keyword to **PRINT** becomes significant. Generally, all printer output is sent to print unit 0. However, if several reports are being generated simultaneously, the ON keyword can be used to place output in several different print units (ranging from 0 to 599).

For example, suppose a program generates two reports, one displaying the names of all customers who are two months late on their bills, and the other displaying the names of all customers who have birthdays approaching. The program goes through each customer’s record in sequence. If bills have not been paid, the customer’s name and address are printed from print unit 0, and the customer is billed. If the customer has a birthday coming up, the name and address are printed from print unit 1, and the customer is sent a birthday card. At the end of the program, two complete (and hopefully distinctive) lists are printed out.

## Formatting and Positioning Output

Normally output will be printed at the current position, and will force a carriage return and linefeed at the end of output. The print expression, however, may include features to tabulate output, to suppress the carriage return and linefeed, and (in the case of screen output) to place output at any coordinate, clear the screen, clear the line, or access any of several terminal capabilities.

In addition, format can be masked directly in an output expression. Format masking is a mechanism in mvBASIC by which data can be converted into a readable format without changing the data itself.

## Tabulation and Carriage Return Suppression

A comma (**,**) in the print expression will force a tab to be printed at that position. A trailing colon (**:**) specifies that the automatic carriage return and linefeed will be suppressed in output.

## Formatted Screens (@)

The **@** function provides direct control of a terminal screen. When the **@** function is used in a print expression, it generates a command sequence that is sent to the terminal screen, and the screen responds accordingly. In particular, the **@** function can be used to move the cursor to any coordinate position on the screen. It can also be used to clear the screen, to clear to the end of the line, or to place the text in blinking mode. A full list of the features for the **@** function is included in the reference page for **@**.

Using the **@** function, a formatted screen can be generated. Programs can use the **@** function to clear the screen and show a menu by sending menu options to different coordinates on the screen. The programmer might choose to turn the echo feature off to prevent user input from appearing on the screen.

For formatted screens, the **INPUT @** statement can take input from any coordinate on the screen. In addition, the **INPUT @** statement performs format masking directly on the input. See [Terminal Input](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/terminal_input.htm) later in this section for more information on **INPUT @**.

Example

To print a menu on the screen, the source code might read:

|  |
| --- |
| PRINT @(-1) :  PRINT @(8,3) : "CHOOSE ONE: " :  PRINT @(16,6) : @(-13) : "E" : @(-14) : "DIT AN ENTRY" :  PRINT @(16,8) : @(-13) : "N" : @(-14) : "EW ENTRY" :  PRINT @(16,10) : @(-13) : "D" : @(-14) : "ELETE AN ENTRY" :  PRINT @(16,12) : @(-13) : "Q " : @(-14) : "UIT" :  ECHO OFF  INPUT @(1,23) : ANSWER,1 |

The code in the preceding example does the following:

* The first line of code clears the screen.
* The second line prints **CHOOSE ONE** at column **8**, row **3**.
* The third through sixth lines print the menu options at specific coordinates, with the first character in reverse video mode. Thus, the first character stands out on the screen.
* The seventh line turns off the echo.
* The eighth line places the cursor at the bottom of the screen and accepts a single character as a response.

## Masking Data (FMT)

Numbers are stored in internal format. Internal format is a representation of data which makes calculation easier but is more difficult to read. Format masks convert numbers into a format that is easier to read. In addition, the **ICONV** function converts string data into internal format, and the **OCONV** function converts strings back into external format.

For example, if the dollar amount $14,912.15 were stored with the dollar sign and comma, then any calculations on that number would be impossible—dollar signs and commas are not permitted in numeric values. Also, if interest is being calculated on this dollar amount, it would be much more accurate if more than two decimal places were being kept.

Suppose that the given dollar amount represents the balance of a bank account. The bank keeps this figure to 5 digits of precision, to ensure that any calculations are accurate—suppose the actual figure stored is 1491214987. When this figure needs to be printed in a monthly statement, the data needs to be converted into a readable form. The program which generates the monthly statements will therefore print the data with a format mask, which will descale the number, round it to 2 decimal places, enter a comma where necessary, and precede it with a dollar sign. If the variable **BALANCE** contains 1491214987, and the program contains the lines:

|  |
| --- |
| PRECISION 4  .  .  .  PRINT BALANCE "29,$"  the output will be:  $14,912.15 |

A data mask can be implemented in two ways: either by using the **FMT** keyword, or by simply following data with the mask expression (as shown in the example). In the source code, the 2 signifies that the output should be rounded to two decimal places. The 9 is a descaling code, which determines where to place the decimal point—in this case, with a precision of 4 and a descaling code of 9, the decimal point is placed (9-4)=5 digits from the right. The, signifies to enter a comma every thousands place, and the **$** says to precede the expression with a dollar sign. There are many other codes available for masking data. For a full list and explanation of these codes, see the reference page for **FMT** in *Statement and Function Reference*.

## Headings and Footings

The **HEADING** statement can be used to specify a heading to be printed at the top of each page. It also has the facility to set up page parameters for use by **FOOTING** and **PAGE**.

If the output is being sent to the screen, then output will stop after each page of text once a **HEADING** statement is used. If a **FOOTING** statement is specified, a footing will be supplied at the end of the page, and the program will wait for a carriage return before continuing with output.

The **PAGE** statement can be used to force a new page at any point in the program, as long as a **HEADING** has been specified.

Note that **HEADING**, **FOOTING**, and **PAGE** will only affect the same output device that **PRINT** does. If multiple print units are being printed together, **HEADING**, **FOOTING**, and **PAGE** will affect print unit 0 (the default).

## The PRINTERR Statement

The **PRINTERR** statement allows mvBASIC programs to produce output messages using the Error Message Processor and the ERRMSG file.

Before the introduction of the **PRINTERR** statement, the Error Message Processor was only available through the use of the **STOP** and **ABORT** statements, both of which terminated the program.

# Terminal Input

Use the **INPUT** statement to request terminal input.

## The INPUT Statement

In the simplest form of terminal input, the user can be prompted for a value for the variable **ANSWER** with the statement:

|  |
| --- |
| INPUT ANSWER |

This statement will print the prompt character on the screen and wait for terminal input at that position. The user can type a response and press ENTER for the response to be accepted. (The prompt character can be reassigned with the **PROMPT** statement.)

## Variations on INPUT

There are several variations to the **INPUT** statement. For example, suppose that the answer the user is prompted for can be only **Y** for yes or **N** for no. The **INPUT** statement can specify that only one character is expected, with the statement:

|  |
| --- |
| INPUT ANSWER,1 |

The maximum number of characters that will be accepted as input is 1. When 1 character is typed by the user, the program will assume that the input is complete and continue execution immediately, without waiting for a carriage return. Any positive integer can be used in an **INPUT** statement as the maximum length of input, up to the size of the input buffer (140).

To force the program to wait for a carriage return before accepting the response, an underscore (**\_**) can be placed at the end of the **INPUT** statement. By using the underscore, the program will send a beep to the terminal if the user tries to type in more than the maximum number of characters, but it will wait for a carriage return before accepting the input. Thus the user is given a chance to verify the response before continuing with the program.

In addition, the **INPUT** statement can be used to print data in the field to be written. The field might contain a default answer, to be accepted (by pressing ENTER) or to be reassigned (by backspacing, typing the new answer, and pressing ENTER). The field might also be filled by a fill character, showing (for example) 5 asterisks when a 5-digit zip code is requested. The only valid fill characters are zeros and asterisks.

A **FROM** clause is supplied with the **INPUT** statement, for accepting input from a remote line. See [Communications](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/communications.htm) later in this section for more information on receiving input from a remote line.

## Input from the Type-ahead Buffer (INPUTIF)

The **INPUTIF** statement accepts input only from the type-ahead buffer. The type-ahead buffer contains characters which have been typed between input requests, to be supplied as a response to the next input request.

The **INPUTIF** statement follows the same syntax as the **INPUT** statement, except that it requires either a **THEN** or **ELSE** clause (or both). If there is at least one character in the type-ahead buffer, the program will wait for a carriage return if one was not already in the type-ahead, and the **THEN** statements will be executed. If the type-ahead is empty, the **ELSE** clause will be instantly executed: the **INPUTIF** statement will not wait for text if the type-ahead is empty.

Example

If a program expects a response within thirty seconds, the source code might read:

|  |
| --- |
| SLEEP 30  INPUTIF RESPONSE ELSE  STOP  END |

The **TA** function returns the number of characters in the type-ahead buffer for the specified line. In addition, the **TA** statement can toggle the type-ahead feature on and off, or clear the type-ahead buffer of the current process. See [Communications](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/communications.htm) later in this section for more information on the type-ahead feature.

## Masked Input Statements (INPUT @)

The **INPUT @** statement combines two invaluable enhancements to the **INPUT** statement: it accepts screen coordinates for the input string, and it allows a mask to be applied directly on the input.

The **INPUT @** statement takes screen coordinates as arguments, and places the cursor (printing the prompt character) at that position. Once the response has been supplied, the **INPUT @** statement compares it and translates it against the mask (if any). If the response does not match the mask, an error message is printed at the last line of the screen and the user is prompted again. The input data is then converted into internal format for storage.

For example, take the case of a screen-formatted program which prompts for a date and then translates it into internal format. Using the standard **INPUT** statement, the program would have to use a loop to place the cursor at the right position, prompt for the input, test for every way a date can be written (JUNE 4 1965, 4 JUN 1965, 6/4/65, 06/04/1965, 6-4-65, etc.), and then convert it to internal format. However, with the **INPUT @** statement, the programmer can simply write:

|  |
| --- |
| INPUT @(14,10) BIRTHDATE "D" |

|  |  |
| --- | --- |
| **D** | Specifies an internal date conversion is performed. |

See [Sending Output to the Screen and Printer](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sending_output_to_the_screen_and_printer.htm) for more information on data masking.

## INPUTTRAP, INPUTNULL and INPUTERR

Several statements were designed to be used concurrently with **INPUT @**.The **INPUTTRAP** and **INPUTNULL** statements were designed to provide escapes from the internal loop of **INPUT @**. The **INPUTNULL** statement specifies a character which will be interpreted as the null string by **INPUT** **@**. The **INPUTTRAP** statement allows the programmer to specify characters which, if supplied as answers to **INPUT @**, will branch to another statement label. Using **INPUTTRAP**, the operator can be told, in a menu program for example, that pressing ESC at any prompt will exit the program, and pressing CTRL+Z will return to the main menu.

In addition, the **INPUTERR** statement prints a message on the last line of the screen. This message will be cleared when correct input is taken by a succeeding **INPUT @** statement. The **INPUTERR** statement can be used to print a message about what sort of input is expected, or it can be used in a loop with **INPUT @** if a response requires further testing.

Example

If the programmer requires a date within the next year, the code might read:

|  |
| --- |
| TODAY=DATE( )  VALID = 0  INPUTERR "PLEASE ENTER A DATE WITHIN THE NEXT YEAR"  LOOP  INPUT @(14,10) RES.DATE "D"  BEGIN CASE  CASE RES.DATE < TODAY  INPUTERR "INVALID INPUT. PLEASE ENTER A  FUTURE DATE."  CASE RES.DATE > TODAY + 365  INPUTERR "PLEASE ENTER A DATE WITHIN THE NEXT YEAR."  CASE 1  VALID = 1  END CASE  UNTIL VALID DO REPEAT |

See [Sending Output to the Screen and Printer](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sending_output_to_the_screen_and_printer.htm) for more information both on data masking and on formatted screens.

## INPUT and the Data Stack

If the data stack is not empty, its contents will be supplied to any terminal input statement, and the user will not be prompted. The data stack is assigned with the **DATA** statement. The **SYSTEM(10)** function can be used to determine whether the data stack is empty.

See [External Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/external_program_control.htm) for more information about the data stack.

# Dynamic Array Processing

mvBase stores all data as a string. File items are separated by segment marks, and lines in file items are separated by attribute marks. For processing data in file items, therefore, mvBASIC supplies several powerful string functions. Using these functions, fields in a file item can be distinguished and processed separately.

There are two categories of string function: those which require a delimiter to be specified, and those which assume the dynamic array delimiters.

## File Items and Dynamic Arrays

Dynamic arrays are a powerful data structure in mvBASIC, since they can be used to represent the contents of a file item. A dynamic array is simply a string variable with attribute marks, value marks, and subvalue marks taken to be field delimiters.

When a file item is read into a string variable by a **READ** or **READU** statement, the fields are separated by attribute marks (CTRL+^, or **CHAR(254)**), and subfields are generally separated by value marks and subvalue marks (CTRL+] and CTRL+\ , or **CHAR(253)** and **CHAR(252)**). The string variable is thus in the form of a dynamic array and can be manipulated by the dynamic array functions.

|  |  |
| --- | --- |
| **NOTE** | When these delimiters are sent to the screen via a **PRINT** or **CRT** statement, they will not appear as expected: mvBASIC will subtract 127 from the ASCII value of a high-order character on output. Thus **CHAR(254)** will appear on output as **~**, **CHAR(253)** will appear as **}**, and **CHAR(252)** will appear as **|**. |

See [Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm) later in this section for more information on reading file items into an mvBASIC program.

## Dynamic Array Functions

To examine or alter the contents of a particular attribute, value, or subvalue of a dynamic array, mvBASIC provides the **EXTRACT**, **REPLACE**, **INSERT**, and **DELETE** functions.

The **EXTRACT** function returns the contents of the particular attribute, value, or subvalue. For example, if Attribute **6** of the dynamic array **CUST** contains the customer’s zip code, a variable **ZIP** can be assigned with:

|  |
| --- |
| ZIP = EXTRACT(CUST,6) |

The **REPLACE** function replaces the contents with new data. For example, if a customer had a new zip code **NEW.ZIP**, it can replace the old zip code in Attribute **6** of the array **CUST** with:

|  |
| --- |
| CUST = REPLACE(CUST, 6 ;  NEW.ZIP) |

The **INSERT** function inserts data as an attribute, value, or subvalue in the given position. For example, if Attribute **6** of **CUST** does not exist, it can be assigned to the customer’s zip code **ZIP** with:

|  |
| --- |
| CUST = INSERT(CUST , 6 ; ZIP) |

The difference between **REPLACE** and **INSERT** is that **REPLACE** will overwrite any data already in the given position, but **INSERT** will simply move it up. If Attribute **6** had already existed in the previous example, the new data will become Attribute **6**, the old Attribute **6** will become Attribute **7**, and so on.

The **DELETE** function deletes the specified attribute, value, or subvalue. For example, Attribute 7 of **CUST** can be deleted with:

|  |
| --- |
| CUST = DELETE(CUST , 7) |

The **DELETE** function does not perform the same function as using **REPLACE** with the null string. By deleting Attribute **7**, Attribute **8** becomes Attribute **7**, and so on. By replacing Attribute **7** with the null string, Attribute **7** becomes null and all other attributes remain unchanged.

## The LOCATE Statement

In addition to the four functions discussed in the preceding section, the **LOCATE** statement proves to be extremely powerful in manipulating dynamic arrays. The **LOCATE** statement searches for a particular attribute, value, or subvalue within a dynamic array string (or subset thereof). If the data has been sorted into an ascending or descending order, the order can be specified in the **LOCATE** statement. **THEN** and **ELSE** clauses are accepted by **LOCATE** to specify action if the string is or is not found.

If the string is found, the **LOCATE** statement sets a specified variable to the position where the data was found, and the statements of the **THEN** clause, if included, are executed. If the string is not found where expected, the variable is set to the current position plus one, and the statements in the **ELSE** clause are executed. In the **ELSE** clause, the variable can be used with an **INSERT** function to place the data in the proper position.

For example, if a dynamic array **LIST** contains names in alphabetical order separated by attribute marks, a new name **NAME** can be inserted with:

|  |
| --- |
| LOCATE(NAME, LIST; POSITION; "A") THEN  PRINT NAME : " ALREADY LISTED."  END ELSE  NAMELIST = INSERT(LIST , POSITION;  NAME)  END |

## Alternate Forms for Dynamic Array Processors

An enhancement of mvBASIC is an alternate form for each of the dynamic array processing functions. This form uses angle brackets in referencing a dynamic array field, thus simulating the syntax for referencing dimensioned arrays. Since the angle brackets tend to be more intuitive, they are generally preferred over the older syntax forms.

The preceding example lines might have read:

|  |  |
| --- | --- |
| **Old** | **New** |
| **ZIP = EXTRACT(CUST,6)** | **ZIP = CUST<6>** |
| **CUST = REPLACE(CUST, 6 ; NEW.ZIP)** | **CUST<6> = ZIP** |
| **CUST = INSERT(CUST , 6 ; ZIP)** | **INS ZIP BEFORE CUST<6>** |
| **CUST = DELETE(CUST , 7)** | **DEL CUST<7>** |
| **LOCATE(NAME, LIST; POSITION; "AL")...** | **LOCATE NAME IN LIST BY "AL"...** |

## Counting Delimiters and Substrings

The **COUNT** function returns the number of times a specified substring appears in a string. The function returns zero if the substring is not found. If the substring is the null string, the function returns the number of characters in the string minus one.

The **DCOUNT** function returns the number of fields separated by a given delimiter. **DCOUNT** can be very useful for processing dynamic arrays as well as other strings. For example, the number of attributes in a string **ADDRESSES** can be determined with:

|  |
| --- |
| NO.OF.ATTRS = DCOUNT(ADDRESSES, CHAR(254)) |

# Generalized String Processing

**EXTRACT**, **REPLACE**, **INSERT**, and **DELETE** are very powerful for referencing and adapting dynamic arrays in mvBASIC. However, they are dependent on the standard delimiters being used within the array. If given a string with different delimiters between its fields, the programmer is forced to use the more generalized string processing functions.

## Substring Assignment

A substring is specified by a starting character position and a substring length, separated by commas and enclosed in square brackets. The general syntax for a portion of a string is:

Format

|  |
| --- |
| *string-expression* [*n*,*m*] |

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Starting column position. |
| ***m*** | Length of the substring. |

Description

Using this syntax, substrings can be extracted and replaced like the fields of a dynamic array or the elements of a dimensioned array.

Example

If the string **NAME** contains "**SHAW, GEORGE BERNARD"**, the variable **FIRSTNAME** can be assigned **"GEORGE"** with:

|  |
| --- |
| FIRSTNAME = NAME [7,6] |

By using column positions, therefore, the **EXTRACT** function can be simulated for string variables. To simulate the **REPLACE**, **INSERT**, and **DELETE** functions, portions of a string can be assigned values directly. To substitute the string **"RALPH"** for **"GEORGE"**, for example, the code might read:

|  |
| --- |
| NAME[7,6] = “RALPH” |

The only thing which is not obvious in manipulating substrings is how to determine the column position and length of the substring. For this purpose, the **FIELD**, **COL1**, **COL2**, **LEN**, and **INDEX** functions are crucial to string processing.

## The FIELD Function

The **FIELD** function accepts any character as a delimiter and returns the specified field, thus acting as a generalized **EXTRACT**. For example, if the string **NAME** contains **"SHAW, GEORGE BERNARD"**, then the last name **"SHAW"** can be placed in the variable **SURNAME** with:

|  |
| --- |
| SURNAME = FIELD(NAME , "," , 1) |

## The COL1, COL2, and LEN Functions

The **COL1** and **COL2** functions, respectively, return the column positions immediately before and immediately after the last **FIELD** function. The **LEN** function returns the number of characters in a string.

For example, if the string **NAME** contained **"SHAW, GEORGE BERNARD"**, the first name **"GEORGE"** could be deleted with:

|  |
| --- |
| FIRSTNAME = FIELD(NAME, " " , 2)  NAME [COL1( ), COL2( )] = " " |

The string **NAME** now contains **"SHAW, BERNARD"**.

## The INDEX Function

The **INDEX** function returns the column at which a particular substring can be found in a string. This value can be used in a substring assignment statement to replace or delete the substring.

## Trimming Spaces

The **TRIM**, **TRIMB**, and **TRIMF** functions each serve to remove all extra spaces from a string. The **TRIM** function trims all multiple spaces and all spaces at the beginning and at the end of a string. The **TRIMB** function trims only spaces at the end of a string, and the **TRIMF** function trims only spaces at the beginning of a string.

### Converting Characters

The **CONVERT** statement can be used to convert every occurrence of a particular character in a string into a different character. The **CONVERT** statement is particularly useful for converting strings with nonstandard delimiters such as spaces, commas, or colons into dynamic array format. For example, if the string **NAME** contained **"SHAW, GEORGE BERNARD"**, this name could be incorporated into a file that had the last name and first name separated by attribute marks with:

|  |
| --- |
| CONVERT "," TO CHAR(254) IN NAME |

# Dimensioned Arrays

A dimensioned array is a one- or two-dimensioned structure for data. Elements of the array can be thought of as cells rather then fields.

The most significant difference between dimensioned and dynamic arrays is that the size of a dimensioned array must be assigned at compilation, whereas the size of a dynamic array can vary at run-time according to need. Dimensioned arrays require more storage allocation and are somewhat less flexible; they are, however, more efficient at run-time. Each element in a dimensioned array has a direct pointer; each field in a dynamic array, on the other hand, requires the entire string to be searched through from the start.

For example, suppose a dynamic array contains a thousand attributes. Attribute 999 actually refers to the data between the 998th and 999th attribute marks; therefore, to access attribute 999, the processor has to search through the string until 999 attribute marks are found. If several fields towards the end of the string need to be accessed this way, the run-time of the program can be drastically affected.

If a vector (a 1-dimensional array) were used instead of a dynamic array, however, the processor has to access only a single direct pointer to element number 999 in order to retrieve the data. Dimensioned arrays, therefore, provide a shortcut to the field.

See [Format, Data and Expressions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/format,_data_and_expressions.htm) for a more complete description of the structure of dimensioned arrays.

## Assigning Dimensioned Array Variables (DIM)

Before a dimensioned array can be used, it needs to be declared with its dimensions. The **DIMENSION** statement declares a dimensioned array. Since the **DIMENSION** statement is interpreted by the compiler, neither variables nor expressions can be used in a **DIMENSION** statement.

If the array is to be allocated space in the **COMMON** area, the **COMMON** statement can also be used to declare a dimensioned array. Arrays that have been declared with the **COMMON** statement, however, should not be declared again with a **DIMENSION** statement.

## Converting Strings to Dimensioned Arrays

The statements **MATPARSE** and **MATBUILD** were designed for placing the elements of a string into a dimensioned array with **MATPARSE** and later reading them back into a string with **MATBUILD**.

**MATPARSE** and **MATBUILD** are flexible enough to be applicable to several string formats. Their behavior, therefore, is dependent on the number of delimiters specified in the syntax. In the most common case, if the string is a dynamic array, each attribute can be placed into a separate element of the array by specifying an attribute mark as the single delimiter. For example, if **STRING** were a dynamic array, the attributes of **STRING** could be read as elements of dimensioned array **ARRAY** with:

|  |
| --- |
| MATPARSE STRING FROM ARRAY, CHAR(254) |

The attribute marks will not appear in any of the elements of the new array. The **MATPARSE** statement in this example might be thought of as a shortcut for:

|  |
| --- |
| FOR I = 1 TO MAXELTS  ARRAY(I) = STRING<I>  NEXT I |

with **MAXELTS** representing the dimensions of the array.

Once the string is read into the array, its elements can be freely read and updated. To write the dimensioned array back into the string, **MATBUILD** can be used with the same delimiter:

|  |
| --- |
| MATBUILD ARRAY FROM STRING, CHAR(254) |

Each element of the array will be placed into the string, separated by attribute marks.

**MATPARSE** and **MATBUILD** are meant to be used together. The same delimiters used for **MATPARSE** should be used with **MATBUILD** to reconstruct the string at the end of processing.

## MATREAD and MATWRITE

The **MATREAD** and **MATWRITE** statements allow file items to be read from and written directly into dimensioned arrays. **MATREAD** is equivalent to performing a **READ** statement and then using **MATPARSE** with an attribute mark as the delimiter. **MATWRITE** is equivalent to using **MATBUILD** with an attribute mark as the delimiter and then performing a **WRITE**. See [Reading and Updating File Items](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/reading_and_updating_file_items.htm) later in this section for more information on reading and writing file items.

## The MAT Statement

The **MAT** statement assigns all elements of a dimensioned array to a single value or to the values of another array. For example, to assign all elements of an array **ARRAY** to 6, the code might read:

|  |
| --- |
| MAT ARRAY = 6 |

This use of the **MAT** statement is a shorthand for:

|  |
| --- |
| FOR I = 1 TO MAXELTS  ARRAY(I) = 6  NEXT I |

With **MAXELTS** representing the maximum dimensions of the array.

To assign the elements of **ARRAY** to the elements of **ARRAY2**, the code might read:

|  |
| --- |
| MAT ARRAY = MAT ARRAY2 |

which is a shorthand for:

|  |
| --- |
| FOR I = 1 TO MAXELTS  ARRAY(I) = ARRAY2(I)  NEXT I |

# Reading and Updating File Items

Before an item in a file can be accessed, it must be assigned a symbolic name, called a file variable. The file variable is necessary to provide a pointer to the file that will be used by the program each time the file is accessed.

## File Variables (OPEN)

The **OPEN** statement assigns a file variable to a file, so that the program can read, write, select, or delete items in the file. All subsequent access of the file must reference the file variable and not the file name itself.

If a file is opened without a file variable specified, it uses the default file variable. Any subsequent file access statements that do not specify a file variable will use the default file variable. Only one file can be assigned to the default file variable at a single time.

## Reading and Writing a File Item (READ, WRITE, etc.)

Once the file is opened, any item can be directly accessed. The **READ** statement assigns the string value of a file item to a dynamic array variable. The fields of the array can then be accessed by the dynamic array processing functions **EXTRACT**, **REPLACE**, **INSERT**, and **DELETE**. The **WRITE** statement writes a new or updated dynamic array string into a file item. There are several variations to **READ** and **WRITE** provided by mvBASIC. The **READV** and **WRITEV** statements read and write only a single attribute of an item, as a shortcut for programs which are concerned only with a single attribute. In addition, the **MATREAD** and **MATWRITE** statements read and write items as dimensioned arrays, with each attribute corresponding to an element of the array.

The file-reading statements are each equipped with **THEN** and **ELSE** clauses. If the item cannot be found, the v statements are executed. If it can be found, the **THEN** statements are executed. See [Internal Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/internal_program_control.htm) for more information on the syntax of THEN and ELSE clauses.

## File Item Locks (READU, WRITEU, RELEASE, etc.)

Each of the statements for reading a file item have corresponding statements that place a lock on the file item as it is read. These statements are the **READU**, **READVU**, and **MATREADU** statements. (The **U** suffix stands for Update, declaring that the file item might be changed and rewritten.) The item lock is lifted either when the item is released with a **RELEASE** statement, deleted with a **DELETE** statement, written with a **WRITE**, **WRITEV**, or **MATWRITE** statement, or when the program is terminated. Until the lock is lifted, no other users will be able to access the same file item with a **READU**, **READVU**, or **MATREADU** statement.

File item locks only affect other **READU**, **READVU**, and **MATREADU** statements. While an item is locked, programs can access the file item with a normal **READ**, **READV**, or **MATREAD** statement, or they can even write it with any of the file writing statements. The success of a file item lock depends on its being respected by all other programs that access the same file.

If an item is to be written but the programmer does not want the lock removed, the **WRITEU**, **WRITEVU**, and **MATWRITEU** statements should be used in place of **WRITE**, **WRITEV** or **MATWRITE**. These statements will write the file item but retain the item lock for subsequent update. (Again, the U suffix stands for Update, declaring that further update might occur.)

## The LOCKED Clause

The item-locking statements **READU**, **READVU**, and **MATREADU** are each equipped with an optional **LOCKED** clause. Normally, when a program attempts to read and lock an item which is already locked, the program waits for the item to be released before continuing with execution. However, if the **LOCKED** clause is included, the program simply executes the **LOCKED** statements and continues with execution immediately. The **LOCKED** statements follow the syntax of **THEN** and **ELSE** clauses in mvBASIC.

The **LOCKED** clause helps to avoid the situation called a deadly embrace. A deadly embrace happens when two users both lock items, and before releasing their locks, each user then tries to read and lock the other item. Without the **LOCKED** clause, both users will be indefinitely stuck since neither is free to unlock its item. If the **LOCKED** clause is used, however, the deadly embrace cannot occur.

## Select-lists (SELECT, READNEXT)

Select-list variables can be created through the mvBASIC **SELECT** statement, or by using the **EXECUTE** statement to call one of the INFO/ACCESS select-list generators. The **SELECT** statement does not accept the selection expressions accepted by the INFO/ACCESS commands; however, the **SELECT** statement does allow a select-list to be created from the attributes of a dynamic array string. See [External Program Control](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/external_program_control.htm) for more information on using EXECUTE for generating select-lists.

A select-list can also be created external to the program by executing one of the INFO/ACCESS select-list generators and then immediately running the program. If the program is designed this way, the **SYSTEM(11)** function is recommended to test if there is an external select-list.

Once the select-list is created, it can be read with the **READNEXT** statement. **READNEXT** reads the next item- ID in the select-list. After selecting a file, the **READNEXT** statement is generally used in a loop to perform a procedure on all selected items.

### Deleting File Items (DELETE, CLEARFILE)

The **DELETE** statement is a statement that deletes a specific file item from an opened file. It should not be confused with the **DELETE** function or the **DEL** statement, which both delete a field from a dynamic array.

The **CLEARFILE** statement deletes all items in the data file.

# Reading and Writing Tapes or Floppy Disks

mvBASIC includes several statements for tape and floppy disk processing. For the purpose of this discussion a floppy disk functions as a tape and therefore, is included in any reference to a tape. For reading and writing strings on tape, there are the **READT** and **WRITET** statements. As expected, the **READT** statement reads the next record off the attached tape device, and the **WRITET** statement writes a record onto the tape. The **READTX** statement is designed for tapes which might contain segment marks. The **READTX** statement is identical to the **READT** statement, except that the data from the tape is translated into ASCII hexadecimal format before it is assigned to the string. The **ICONV** function can then be used to translate the string back into readable format. **READTX** is designed for reading segment marks (**CHAR(255)**) from a tape.

In addition, there are statements to simulate the **T-WEOF** and **T-REW** commands. The **WEOF** statement writes an End-Of-File mark at the current position of the tape, and the **REWIND** statement rewinds the tape to the beginning.

Each of the tape I/O statements includes **THEN** and **ELSE** clauses to specify action according to whether the tape statement was successful. The **ELS** clause is often used to produce a meaningful error message by calling the **SYSTEM(0)** function. The **SYSTEM(0)** function returns a number from 0 to 4, reflecting whether the latest tape I/O statement worked, and if it didn’t, what the problem was. See [Statement and Function Reference](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/statement_and_function_reference.htm) for more information.

# Communications

mvBASIC includes several statements designed for communication programs on the mvBase system. Data can be sent to or taken from any remote line.

## Unlinking and Attaching a Line

Before a program can communicate with a remote line, the line should first be attached with the **LINE-ATT** command. The **LINE-ATT** command can be executed only if the line is not already linked to another process. To unlink a line from a process, use the **UNLINK-LINE** command.

## Sending Data to a Line (SEND, SENDX, SENDBREAK)

The **SEND** statement sends data to a remote line. **SEND** does not require that the remote line be attached, but it does require that the line not be linked to another process. The syntax of the **SEND** statement is based on that of the **PRINT**, **CRT**, and **DISPLAY** statements, in that it accepts commas for tabulation, **@**-functions for screen formatting, and even trailing colons for carriage return and linefeed suppression.

A variation of the **SEND** statement is the **SENDX** statement, which sends data written in ASCII hexadecimal format to the remote line, translating it into standard ASCII characters. The **SENDX** statement was designed for sending a segment mark (**CHAR(255)**), which is normally taken as the end-of-data character. If the **SENDX** statement is used, a segment mark can be converted to hexadecimal (FF) and thus be successfully transmitted. To convert data into ASCII hexadecimal, use the **OCONV** function with the **MX** conversion code.

## Sending a BREAK

The **SENDBREAK** statement sends a break signal to the designated remote line. It might be used, for example, for hanging up modems that recognize a break as a hanging signal.

## Receiving Data from a Line

There are several ways of receiving data from an attached line in mvBASIC. The **INPUT** and **INPUTIF** statements include **FROM** clauses, which can be used to specify a remote line from which the input should be taken. In addition, the **GET** and **GETX** statements are designed specifically for taking characters from a remote line.

## The Type-ahead Buffer

The behaviour of input statements often depends on the status of the type-ahead feature. The type-ahead feature is generally enabled on mvBase. When type-ahead is enabled, characters are accepted from a line even when there is no prompt for input. These characters are placed in the type-ahead buffer. The type-ahead buffer accepts up to 127 characters on input. The **LIST-LINE-CHARS** command displays, among other things, whether the type-ahead feature is currently on for each process, and how many characters are currently in the type-ahead buffer for that line.

If type-ahead is enabled, the characters in the type-ahead buffer are taken as responses to the next input request. Thus an operator who is familiar with a program can type responses to prompts before they are printed, without having to wait for the program to catch up.

The type-ahead feature is turned on by the **TA-ON** command and turned off with **TA-OFF**. A user logged on to the SYSPROG account can turn type-ahead on or off for any process, defaulting to the current process. In addition, the **TA-ON** and **TA-OFF** statements in mvBASIC can be used to toggle the type-ahead feature for the current process or for all attached processes. A programmer might wish to turn type-ahead off during execution if there is danger of a user making mistakes by responding too quickly. If a programmer chooses to turn off the type-ahead feature in a program, however, it should be turned on again before the program is terminated.

An alternative to turning type-ahead off is to clear the type-ahead buffer periodically. The type-ahead buffer can be cleared in mvBASIC with the **TA-CLEAR** or **INPUTCLEAR** statement. However, only the type-ahead buffer for the current line is cleared. To clear the type-ahead buffer for another line, the SYSPROG command **PROTOCOL** with the **C** option should be used.

To determine the number of characters currently in the type-ahead buffer for the current process, or for any other process, use the **TA** function.

See the *TCL Reference Guide* for more information on the type-ahead feature.

## Input from an Attached Line (GET, GETX)

In general there are two ways to accept input from a remote attached line: through the **FROM** clauses of the **INPUT** and **INPUTIF** statements, or through the **GET** or **GETX** statement.

The **GET** statement is an all-purpose statement for taking input from an attached line. The **GET** statement will accept any character from the attached line, with the exception of segment marks (**CHAR(255)**), which are interpreted as the end of data. The **GET** statement includes many optional clauses, which determine the conditions at which the input is terminated, and whether the program should wait for input or just check the type-ahead buffer.

Among the conditions for termination supported by the **GET** statement are:

* Terminate input when a certain character is read.
* Wait only a specified number of seconds before timing out.
* Accept only a specified number of characters.
* Accept input only from the type-ahead buffer.

In addition to **GET**, mvBASIC supports a **GETX** statement, which is identical to **GET** except that the input characters from the remote line are translated into ASCII hexadecimal format before they are taken. The characters can then be converted back into normal ASCII characters with the **ICONV** function and the **MX** conversion code. The **GETX** statement was designed to accept segment marks from the remote line, which are normally interpreted as the end of data.

# Execution Locks

The **LOCK** statement sets an execution lock which establishes exclusive use by one process until the lock is removed. mvBase provides a set of 256 execution locks. These locks can be accessed only through mvBASIC. Execution locks should not be confused with file item locks, since they use a very different mechanism.

Execution locks are set with the **LOCK** statement by specifying a lock number. That lock number is determined only by local convention. mvBase establishes 256 slots (0 through 255), and keeps track of whether the slot is taken or not. It is up to the application programs to take advantage of this structure.

For example, suppose a particular subroutine tends to slow down the system each time it is used. If a **LOCK** statement is used at the beginning of the subroutine, then only one user will be able to execute the subroutine at any given time. The lock number should be unique.

Execution locks are released at the termination of the program, or at the encounter of an **UNLOCK** statement. The **UNLOCK** statement can be used to release a specific lock number or to release all locks set by the current program.

## The THEN and ELSE Clauses to LOCK

**LOCK** is supplied with optional **THEN** and **ELSE** clauses that have the same effect as the **LOCKED** clause for **READU**. Normally, when a **LOCK** statement is used on a lock number which is already locked, the program will wait for the lock to be lifted before continuing with execution. However, if the **THEN** or **ELSE** clause is included, the program will simply execute the **ELSE** clause, if present, and continue with execution immediately.

The **THEN** and **ELSE** clauses help to avoid the situation called a deadly embrace. A deadly embrace happens when, for instance, one user sets execution lock 1, and another user sets execution lock 2. The first user then attempts to access the procedure controlled by lock 2, and the second user attempts to access the procedure controlled by lock 1. Each user is now stuck, since the program will wait indefinitely for the lock to be released, and neither is free to release the lock that has already been set by their process. If the **THEN** or **ELSE** clause is used, however, the deadly embrace cannot occur.

For a list of all execution locks, use the WHAT command.

# Compiler Directives

There are four compiler directive statements. Each of these statements begins with a dollar sign (**$**).

## Comments in the Object Code

The **$\*** statement places a comment directly in the object code of a program when it is compiled. It is most useful for entering version numbers or copyright information before software is distributed.

## Reading External Source Code ($CHAIN, $INCLUDE, $INSERT)

Three statements tell the compiler to read source code from another file item: **$INCLUDE**, **$INSERT**, and **$CHAIN**.

**$INCLUDE** and **$INSERT** are identical statements. Either statement results in the program being compiled as if the external source code were written at the point where the **$INCLUDE** or **$INSERT** statement had been entered. Compilation will then continue at the line after the **$INCLUDE** or **$INSERT** statement. **$INCLUDE** and **$INSERT** are also useful for any code that might be used by several different programs. An example of such code might be a file item containing **COMMON** statements.

The **$CHAIN** statement is different from **$INCLUDE** and **$INSERT** in that the compilation will not return to the original program. The **$CHAIN** statement is not intended for code which might be shared by several programs, but for programs which may have source code longer than 32K bytes. The **$CHAIN** statement allows several different file items containing source code to be **CHAIN**ed together.

The only restriction to **$INCLUDE**, **$INSERT**, and **$CHAIN** is that the number of bytes in the resulting object code cannot exceed 248K.

# Miscellaneous Statements and Functions

The **INPUT CTRL** statement allows the programmer to toggle the acceptance of control characters in terminal input.

The **SLEEP** and **RQM** statements suspend program execution for a number of seconds, or until a specified time of day.

The **ECHO** statement toggles the echo feature for the attached process.

The **REM**, **\***, and **!** statements allow comment lines to be placed in the source code. This statement allows the programmer to document code and make it more accessible to future modification.

The **PROCREAD** and **PROCWRITE** statements allow the program to read to and write from the primary input buffer of the calling Proc.

The **LINESTATUS** function returns information on the DTR and RTS signals on a specified line. This information is important for communication programs that need to know if a device (such as a modem) is properly connected.

The **USER\_DEFINED** function allows the programmer to create mvBASIC functions that may be used in any expression in the mvBASIC program. This function accepts any number of input expressions and always returns a single value.

## Conversion Codes (ICONV, OCONV)

The **ICONV** function translates a string from external to internal format, according to the INFO/ACCESS conversion codes. The conversion codes supported are those for dates, time, hexadecimal, and table translation. The **OCONV** function translates back from internal to external format. See INFO/ACCESS for more information on conversion codes.

## The SENTENCE ( ) Function

The **SENTENCE( )** function returns the TCL or Proc statement used to invoke the mvBASIC program that contains the function.

## The SYSTEM Function

The **SYSTEM** function returns significant information about the system. Some information made available by **SYSTEM** follows:

* The command-line options used to the **RUN** command.
* The error code for a failed tape or floppy disk I/O statement, or the tape or floppy disk record length.
* Whether the program was called by a Proc, whether the external select-list variable has been set, whether the data stack is empty, or whether the program is cataloged.
* Whether output is being sent to the printer, the number of lines left on the current page, the current page number, and current open Spooler files.
* The operator’s terminal type, or the number of lines or columns on the operator’s terminal.
* The user’s account number, process number, or line number.
* The system identification number.

## Entering the Debugger

If the BREAK key is pressed during program execution, the user will be placed in the mvBASIC Debugger. This feature can be disabled and reinstated with the BREAK statement. The **BREAK** statement does not simply toggle the break feature, it also increments and decrements the Break Inhibit Counter.

Alternative ways of entering the Debugger are to use the **D** option to the **RUN** command or to place a **DEBUG** statement directly in the source code.

See [Using the mvBASIC Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_the_mvbasic_debugger.htm) for a full description of Debugger commands and how they are used.

# The Error Message Processor

mvBase permits powerful formatting control when printing error messages. The following are examples:

You may create an error message and display that message on the screen at a specified location. This is done by using **@(row,column)** and **@(column)** cursor addressing directives, and **@(-*n*)** screen functions. The parameters used to specify the location of the message may have the following formats:

Parameter(s)

|  |  |
| --- | --- |
| ***n*** | Where ***n*** is a literal numeric value. (of the error message) |
| **A** | Where **A** is the next error message parameter value. |
| **A+*n*** | Where the numeric value ***n*** is added to the next error message parameter value **A**. |
| **A-*n*** | Where the numeric value ***n*** is subtracted from the next error message parameter value **A**. |
| **A\**n*** | Where the next error message parameter value **A** is multiplied by the numeric value ***n***. |
| **A/*n*** | Where the next error message parameter value **A** is divided by the numeric value ***n***. |

Parameters 5 and 7 are passed to the error message **ERROR1A**:

|  |  |  |
| --- | --- | --- |
| ID: | ERROR1A |  |
| 001 | @(A,A) | Outputs at position (5,7) |
| 002 | H\*\* x=5 y=7 \*\* | Outputs \*\* x=5 y=7 \*\* |
| 003 | S1 | Resets pointer to parameter #1 |
| 004 | @(A+3,A) | Outputs at position (8,7) |
| 005 | H\*\* x=8 y=7 \*\* | Outputs \*\* x=8 y=7 \*\* |
| 006 | S1 | Resets pointer to parameter #1 |
| 007 | @(A,A-3) | Outputs at position (5,4) |
| 008 | H\*\* x=5 y=4 \*\* | Outputs \*\* x=5 y=4 \*\* |
| 009 | S1 | Resets pointer to parameter #1 |
| 010 | @(A\*3,A/1) | Outputs at position (15,7) |
| 011 | H\*\* x=15 y=7 \*\* | Outputs \*\* x=15 y=7 \*\* |

The parameters that can be passed to the error message (using mvBASIC **STOP**, **ABORT** or **PRINTERR** statements) may be accessed in any sequence and may be reused if necessary. The advantages of this are:

* The user has control over the order in which the parameters will be displayed in the final error message. Previously, the parameters were displayed only in the order in which they were passed to the error message.
* Parameters can be reused. This allows the use of windowing capabilities. For example, suppose parameters **X** and **Y** are passed to the error message. These parameters would be used as base coordinates **(X,Y)**. You could then use **@(row,column)** to reference all other desired cursor positions.

Parameters 12 and 17 are passed to the error message, **ERROR1**:

|  |  |  |
| --- | --- | --- |
| ID: | ERROR1 |  |
| 001 | @(A,A) | Outputs at position (12,17) |
| 002 | H\*\* x=12 y=17 \*\* | Outputs \*\* x=12 y=17 \*\* |
| 003 | S1 | Resets pointer to parameter #1 |
| 004 | @(A+3,A) | Outputs at position (15,17) |
| 005 | H\*\* x=15 y=17 \*\* | Outputs \*\* x=15 y=17 \*\* |

The enhanced **S** command used in the **ERROR1** example is now able to use numeric literal parameters without parenthesis **( )**. For example, instead of using **S(1)**, **S1** may be used.  is a numeric literal. In the previous example, the internal pointer is reset to parameter #1. Use the format:

|  |
| --- |
| S{*nnn* / (*nnn*)} |

Suppose the three parameters **X**, **Y** and **Z** are passed to the following error message, **ERROR2**:

|  |  |  |
| --- | --- | --- |
| ID: | ERROR2 |  |
| 001 | E | Outputs ID:ERROR2 |
| 002 | A(2) | Outputs X |
| 003 | A(2) | Outputs Y |
| 004 | A(2) | Outputs Z |
| 005 | S2 | Resets pointer to parameter #2 |
| 006 | A(2) | Outputs Y |
| 007 | A(2) | Outputs Z |

The following message will be displayed: **ERROR2,** **X,** **Y,** **Z** followed by **Y**, then **Z**.

Suppose parameters 10 and 15 are passed to the following error message, **ERROR3**:

|  |  |  |
| --- | --- | --- |
| ID: | ERROR3 |  |
| 001 | @(A,A) | Outputs at position (10,15) |
| 002 | H\*\*\*\*\*\*\*\*\*\* | Outputs \*\*\*\*\*\*\*\*\*\* |
| 003 | S1 | Resets pointer to parameter #1 |
| 004 | @(A+1,A) | Outputs at position (11,15) |
| 005 | H\* WINDOW \* | Outputs \* WINDOW \* |
| 006 | S1 | Resets pointer to parameter #1 |
| 007 | @(A+2,A) | Outputs at position (12,15) |
| 008 | H\*\*\*\*\*\*\*\*\*\* | Outputs \*\*\*\*\*\*\*\*\*\* |

The following message will be displayed at the base coordinate **@(10,15)**:

|  |
| --- |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* WINDOW \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

|  |  |
| --- | --- |
| **NOTE** | After parameters are referenced (using the **A**, **R** and **X** commands), the internal pointer advances to the next available parameter. For example, in **ERROR2** the **A** command references parameter **X** and the internal pointer advances to the next parameter **Y**. The error message ID is interpreted as parameter 0 (as with PROC). Therefore, specifying **S0** within your error message will allow the ID to be used and reused by the next parameter referenced. |

You may use ASCII codes in error messages. The Error Message Processor will translate the code from a decimal number to its ASCII equivalent. Functions previously impossible to perform from within an error message can be performed. For example, a peripheral escape sequence can be easily added to an error message as show below.

Format

|  |
| --- |
| C(*code*) |

Parameter(s)

|  |  |
| --- | --- |
| ***code*** | Decimal ASCII character that the Error Message Processor will translate. |

Example

|  |
| --- |
| C(27) |

The example above outputs the ASCII <ESCAPE> character.

Any command line in the error message beginning with a question mark (**?**) will require a keystroke from the terminal before allowing the program to continue. This ensures that the user has a chance to see the error message before it is cleared from view by the next output screen.

Any line beginning with an exclamation mark (**!**) will be interpreted as a comment and will be ignored by the Error Message Processor. This feature allows you to add documentation to your error messages.

# Statement and Function Reference

This is the reference section for mvBASIC statements and functions. Each statement or function has its own topic, and each topic contains a description of its purpose, an explanation of its syntax, and at least one example of its use.

Readers who are unfamiliar with mvBASIC should read [Format, Data, and Expressions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/format,_data_and_expressions.htm) and [Overview of mvBASIC Statements and Functions](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/overview_of_mvbasic_statements_and_functions.htm) before referring to this section.

The following statements and functions are presented:

|  |  |  |  |
| --- | --- | --- | --- |
| [$\* Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dollar_asterisk_statement.htm)  [$CHAIN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dollar_chain_statement.htm)  [$INCLUDE/$INSERT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dollar_include_dollar_insert_statement.htm)  [= Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/__statement.htm)  [[ ]= Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/brackets___statement.htm)  [@ Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/@_function.htm)  [ABORT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/abort_statement.htm)  [ABS Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/abs_function.htm)  [ALPHA Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/alpha_function.htm)  [ASCII Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ascii_function.htm)  [ASSIGNED/UNASSIGNED Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/assigned_unassigned_function.htm)  [ATTACH TAPE DEVICES/ CHANGE BLOCK-SIZE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/attach_tape_devices__change_block-size_statement.htm)  [AUX ON/AUX OFF Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/aux_on_aux_off_statement.htm)  [BLOCK/UNBLOCK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/block_unblock_statement.htm)  [BREAK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/break_statement.htm)  [CALL Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/call_statement.htm)  [CASE Construct](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/case_construct.htm)  [CHAIN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/chain_statement.htm)  [CHANGE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/change_function.htm)  [CHAR Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/char_function.htm)  [CLEAR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/clear_statement.htm)  [CLEARCOMMON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/clearcommon_statement.htm)  [CLEARDATA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/cleardata_statement.htm)  [CLEARFILE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/clearfile__statement.htm)  [CLEARSELECT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/clearselect_statement.htm)  [COL1 Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/col1_function.htm)  [COL2 Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/col2_function.htm)  [COMMON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/common_statement.htm)  [COMPARE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/compare__statement.htm)  [CONNECT/ DISCONNECT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/connect__disconnect_statement.htm)  [CONSOLE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/console_statement.htm)  [CONVERT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/convert_function.htm)  [CONVERT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/convert_statement.htm)  [COS Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/cos_function.htm)  [COUNT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/count_function.htm)  [CRT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/crt_statement.htm)  [CRT ON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/crt_on_statement.htm)  [DATA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/data_statement.htm)  [DATE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/date_function.htm)  [DCOUNT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dcount_function.htm)  [DEBUG Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/debug_statement.htm)  [DEL Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/del_statement.htm)  [DELETE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/delete_function.htm)  [DELETE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/delete_statement.htm)  [DETACH TAPE DEVICES Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/detach_tape_devices_statement.htm)  [DIMENSION Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dimension_statement.htm)  [DISCONNECT /CONNECT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/disconnect__connect_statement.htm) | [DISPLAY Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/display_statement.htm)  [DQUOTE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dquote_function.htm)  [DTX Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dtx_function.htm)  [EBCDIC Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ebcdic_function.htm)  [ECHO Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/echo_statement.htm)  [END Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/end_statement.htm)  [ENTER Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/enter_statement.htm)  [EQUATE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/equate_statement.htm)  [EREPLACE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ereplace_function.htm)  [ERROR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/error_statement.htm)  [EXCHANGE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/exchange_function.htm)  [EXECUTE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/execute_statement.htm)  [EXP Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/exp_function.htm)  [EXTRACT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/extract_function.htm)  [FIELD Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/field_function.htm)  [FMT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/fmt_function.htm)  [FOLD Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/fold_function.htm)  [FOOTING Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/footing_statement.htm)  [FOR...NEXT Construct](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/for_next_construct.htm)  [FUNCTION Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/function_statement.htm)  [GET Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/get_statement.htm)  [GETX Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/getx_statement.htm)  [GOSUB Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/gosub_statement.htm)  [GOTO Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/goto_statement.htm)  [HEADING Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/heading_statement.htm)  [ICONV Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/iconv_function.htm)  [IF Construct](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/if_construct.htm)  [IFR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ifr_statement.htm)  [IN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/in_statement.htm)  [INDEX Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/index_function.htm)  [INMAT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inmat_function.htm)  [INPUT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/input_statement.htm)  [INPUT @ Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/input_@_statement.htm)  [INPUTCLEAR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputclear_statement.htm)  [INPUT CTRL Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/input_ctrl_statement.htm)  [INPUT Statement with the ELSE Clause](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/input_statement_with_the_else_clause.htm)  [INPUTERR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputerr_statement.htm)  [INPUTIF Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputif_statement.htm)  [INPUTNULL Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputnull_statement.htm)  [INPUTPROMPT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputprompt_statement.htm)  [INPUTTRAP Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/inputtrap_statement.htm)  [INS Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ins_statement.htm)  [INSERT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/insert_function.htm)  [INT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/int_function.htm)  [LEN Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/len_function.htm)  [LINESTATUS Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/linestatus_function.htm)  [LN Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ln_function.htm) | [LOCATE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/locate_statement.htm)  [LOCK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/lock_statement.htm)  [LOCK-BEEP Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/lock-beep_statement.htm)  [LOOP Construct](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/loop_construct.htm)  [MAT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/mat_statement.htm)  [MATBUILD Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matbuild_statement.htm)  [MATPARSE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matparse_statement.htm)  [MATREAD Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matread_statement.htm)  [MATREADU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matreadu_statement.htm)  [MATWRITE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matwrite_statement.htm)  [MATWRITEU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/matwriteu_statement.htm)  [MAXIMUM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/maximum_function.htm)  [MINIMUM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/minimum_function.htm)  [MOD Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/mod_function.htm)  [NOT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/not_function.htm)  [NULL Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/null_statement.htm)  [NUM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/num_function.htm)  [OCONV Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/oconv_function.htm)  [ON ERROR Clause](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/on_error_clause.htm)  [OPEN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/open_statement.htm)  [PAGE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/page_statement.htm)  [PRECISION Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/precision_statement.htm)  [PRINT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/print_statement.htm)  [PRINTER Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/printer_statement.htm)  [PRINTERR Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/printerr_statement.htm)  [PROCREAD Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/procread_statement.htm)  [PROCWRITE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/procwrite_statement.htm)  [PROMPT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/prompt_statement.htm)  [PWR Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/pwr_function.htm)  [READ Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/read_statement.htm)  [READB Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readb_statement.htm)  [READF Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readf_statement.htm)  [READNEXT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readnext_statement.htm)  [READT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readt_statement.htm)  [READTX Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readtx_statement.htm)  [READU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readu_statement.htm)  [READV Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readv_statement.htm)  [READVU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/readvu_statement.htm)  [RELEASE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/release_statement.htm)  [REM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/rem_function.htm)  [REM Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/rem_statement.htm)  [REMOVE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/remove_statement.htm)  [REPLACE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/replace_function.htm)  [RETURN Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/return_statement.htm)  [RETURNING Clause](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/returning_clause.htm)  [REWIND Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/rewind_statement.htm)  [RND Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/rnd_function.htm)  [RQM Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/rqm_statement.htm) | [SELECT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/select_statement.htm)  [SEND Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/send_statement.htm)  [SENDBREAK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sendbreak_statement.htm)  [SENDX Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sendx_statement.htm)  [SENTENCE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sentence_function.htm)  [SEQ Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/seq_function.htm)  [SIN Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sin_function.htm)  [SLEEP Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sleep_statement.htm)  [SORT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sort_function.htm)  [SOUNDEX Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/soundex_function.htm)  [SPACE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/space_function.htm)  [SPOOLQ Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/spoolq_statement.htm)  [SQRT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sqrt_function.htm)  [SQUOTE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/squote_function.htm)  [STATUS Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/status_function.htm)  [STOP Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/stop_statement.htm)  [STR Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/str_statement.htm)  [SUBROUTINE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/subroutine_statement.htm)  [SUM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/sum_function.htm)  [SUMMATION Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/summation_function.htm)  [SWAP Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/swap_function.htm)  [SYSTEM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/system_function.htm)  [TA Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ta_function.htm)  [TA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/ta_statement.htm)  [TAN Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/tan_function.htm)  [TIME Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/time_function.htm)  [TIMEDATE Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/timedate_function.htm)  [TIMEOUT Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/timeout_statement.htm)  [TRIM Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/trim_function.htm)  [TRIMB Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/trimb_function.htm)  [TRIMF Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/trimf_function.htm)  [UNASSIGNED/ASSIGNED Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/unassigned_assigned_function.htm)  [UNBLOCK/BLOCK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/unblock_block_statement.htm)  [UNLOCK Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/unlock_statement.htm)  [WAKEUP Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/wakeup_statement.htm)  [WEOF Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/weof_statement.htm)  [WRITE Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/write_statement.htm)  [WRITEB Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writeb_statement.htm)  [WRITEBU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writebu_statement.htm)  [WRITEF Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writef_statement.htm)  [WRITEFU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writefu_statement.htm)  [WRITET Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writet_statement.htm)  [WRITEU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writeu_statement.htm)  [WRITEV Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writev_statement.htm)  [WRITEVU Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/writevu_statement.htm)  [XTD Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/xtd_function.htm) |

Tokens Supported For System Delimiters

mvBase supports run-time efficient tokens called **@AM**, **@FM**, **@VM** and **@SM** for the attribute delimiter, the field delimiter, the value delimiter, and the sub-value delimiter, respectively.

|  |  |  |
| --- | --- | --- |
| **Token Name** | **Delimiter** | **Value** |
| **@AM** | Attribute mark | **CHAR(254)** |
| **@FM** | Field mark | **CHAR(254)** |
| **@VM** | Value mark | **CHAR(253)** |
| **@SM** | Subvalue mark | **CHAR(252)** |

These new tokens may be used anywhere **CHAR( )** may be used, including **EQUATE** statements.

## $\* Statement

The **$\*** statement allows a comment to be embedded directly into the program’s object code at compilation.

Format

|  |
| --- |
| $\* " *text* "  $\* ' *text* '  $\* \ *text* \ |

Parameter(s)

|  |  |
| --- | --- |
| ***text*** | The comment text, enclosed within string delimiters. |

Description

The **$\*** statement directs the compiler to write the quoted text directly into the object code of the program. These comments are generally used to place copyright information or version numbers into source code before it is distributed.

Examples

To place the string **"HELLO"** directly into the object code of a program, the code might read:

|  |
| --- |
| $\* "HELLO" |

In the next application, the **$\*** statement is used to place a copyright into an adventure game:

|  |
| --- |
| $\* " DRAGONS -- Version 1.1 "  $\* " Copyright 1989 N. West, NBM Inc. "  EQUATE TRUE TO 1, FALSE TO 0,     BEL TO CHAR(7)  .  .  . |

## $CHAIN Statement

The **$CHAIN** statement allows source code to be read in from an external file item.

Format

|  |
| --- |
| $CHAIN[*filename*] *item-ID* |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the file containing the item. If ***filename*** is omitted, the current file is assumed. |
| ***item-ID*** | ***item-ID*** of the item containing the source code |

Description

The **$CHAIN** statement directs the compiler to read source code from the specified file item and compile it as if it were written directly in the current program. The **$CHAIN** statement differs from the **$INCLUDE** and **$INSERT** statements in that the compiler does not return to the main program after compiling the source code in the specified file item; any statements appearing after the **$CHAIN** statement are ignored.

PC/OS versions prior to Mentor PRO 4.0 limited the size of items to 32K. Therefore, the **$CHAIN** statement was designed for programs with source code larger than 32K. By using **$CHAIN** statements, very long source code may be broken up into several file items and **$CHAIN**ed together, as long as the object code does not exceed 248K.

Example

To transfer compilation to source code in item **PROG2** in file **BP**, the code might read:

|  |
| --- |
| $CHAIN BP PROG2 |

In the next example, a very long program is broken up into 3 different source file items, each one but the last ending with a **$CHAIN** statement that calls the next.

The source file item **CUST.ENTRY**:

|  |
| --- |
| EQUATE TRUE TO 1, FALSE TO 0,     AM TO CHAR (254), VM TO CHAR (253),     .     .     .  PRINT "PRINT REPORTS?"  $CHAIN CUST.ENTRY2  At the end of CUST.ENTRY, source file item CUST.ENTRY2 is called with a $CHAIN statement. CUST.ENTRY2 contains:  INPUT ANSWER     .     .     .  FOUND = FALSE  FOR I = 1 TO NO.OF.ELTS UNTIL FOUND     IF CUST.ARRAY (I) < 1, 3 > = NAME THEN  $CHAIN CUST.ENTRY3  CUST.ENTRY2, in turn, calls CUST.ENTRY3, which reads:        FOUND = TRUE     END  NEXT I  .  .  . |

Note in the example that a **FOR** loop and an **IF** construct are broken over two file items.

## $INCLUDE/$INSERT Statement

**$INCLUDE** and **$INSERT** statements allow source code to be read in from an external file item.

Format

|  |
| --- |
| $INCLUDE[*filename*] *item-ID*  $INSERT[*filename*] *item-ID* |

Parameter(s)

|  |  |
| --- | --- |
| ***filename*** | Name of the file containing the item. If ***filename*** is omitted, the current file is assumed. |
| ***item-ID*** | ***item-ID*** of the item containing the source code. |

Description

The **$INCLUDE** and **$INSERT** statements direct the compiler to read in source code from the specified file item and compile it as if it were written directly in the current program. The **$INSERT** statement is identical to the **$INCLUD**E statement.

The **$INCLUDE** statement differs from the **$CHAIN** statement in that the compiler returns to the main program and continues compiling with the statement following the **$INCLUDE**.

The **$INCLUDE** and **$INSERT** statements are particularly useful to read in file items containing **COMMON** and **EQUATE** statements, or any statements which a programmer might want to be consistent between several different programs. Be careful, however, of naming conflicts between file items.

**$INCLUDE** statements may be nested; that is, a program may **$INCLUDE** a file item which **$INCLUDE**s another file item, up to 40 levels. However, the total object code when compiled may not exceed 248K bytes.

If the source code read in through an **$INCLUDE** statement generates a run-time error message, the error message displays only the number of the line which contains the **$INCLUDE** statement. The line numbers from the external file item are not kept in the object code.

Examples

To read in the source code written in item **SETUP** in file **BP**, the code might read:

|  |
| --- |
| $INCLUDE BP SETUP |

In the next application, the **$INCLUDE** statement is used at the beginning of a program to read in common variables, equated variables, and the part of the program which opens the file.

|  |
| --- |
| $INCLUDE STARTUP  PRINT "ENTER THE CUSTOMER ID : ":  INPUT ID  MATREAD PHONE.ARRAY FROM CUSTFILE, ID ELSE     PRINT "CANNOT READ RECORD!"     STOP  END  The file item STARTUP contains the text:  COMMON PHONE.ARRAY(10), PHONEREC  EQUATE TRUE TO 1, FALSE TO 0, AM TO CHAR(254)  PROMPT " "  OPEN "CUSTOMERS" TO CUSTFILE ELSE     ABORT 201, "CUSTOMERS"  END |

## = Statement

The direct assignment statement assigns a value to a variable.

Format

|  |
| --- |
| *var* = *expr*  *var* += *expr*  *var* -= *expr*  *var* \*= *expr*  *var* /= *expr*  *var* := *expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***var*** | Variable to be assigned. |
| ***expr*** | An expression evaluating to the value to be assigned to ***var***. |

Description

The direct assignment statement (**=**) assigns the value of ***expr*** to the variable ***var***. In addition, there are several other forms of the assignment statement, of the form ***var op*=** ***expr***, which is equivalent to ***var*** **=** ***var op expr***.

|  |  |
| --- | --- |
| **= *expr*** | ***var*** takes the current value of ***expr***. |
| **+= *expr*** | ***var*** becomes ***var*** plus the current value of ***expr***. |
| **-= *expr*** | ***var*** becomes ***var*** minus the current value of ***expr***. |
| **\*= *expr*** | ***var*** becomes ***var*** multiplied by the current value of ***expr***. |
| **/= *expr*** | ***var*** becomes ***var*** divided by the current value of ***expr***. |
| **:= *expr*** | ***var*** becomes ***var*** concatenated with the current value of ***expr***. |

Examples

If the **SURNAME** were to be appended to the variable **NAME**, the code might read:

|  |
| --- |
| NAME : = SURNAME |

In the next application, assignment statements are used to assign to the variable **PROFIT** the value of **COST** minus **PRICE**, and then to subtract from **PROFIT** the value of the overhead allotted to that sale. An external subroutine **CALC.OVERHEA**D assigns the value **OVERHEAD** based on the value of **PROFIT**.

|  |
| --- |
| PRINT "ENTER COST OF ITEM: "  INPUT COST  PRINT "ENTER PRICE AT WHICH ITEM WAS SOLD: "  INPUT PRICE  PROFIT = COST - PRICE  CALL CALC.OVERHEAD (PROFIT,OVERHEAD)  PROFIT  - =  OVERHEAD  PRINT "WITH OVERHEAD TAKEN INTO ACCOUNT, THE  PROFIT IS:"  PRINT PROFIT |

## [ ]= Statement

The substring assignment statement replaces a part of a string.

Format

|  |
| --- |
| *string* [*expr1*,*expr2*] = *expr3* |

Parameter(s)

|  |  |
| --- | --- |
| ***string*** | String variable to be changed. |
| ***expr1*** | An expression evaluating to the starting character position. |
| ***expr2*** | An expression evaluating to the ending character position. |
| ***expr3*** | An expression evaluating to the replacement string. |

Description

The substring assignment statement allows any part of a string to be reassigned to another value. The substring assignment statement may be used with the **FIELD**, **COL( )**, and **COL2( )** functions to provide the same performance as the **REPLACE**, **INSERT**, and **DELETE** dynamic array functions.

The behavior of the substring assignment is dependent on the values of ***expr1*** and ***expr2***. The rules are as follows:

|  |  |  |
| --- | --- | --- |
| ***expr1* >= 0** | If ***expr1*** is nonnegative, it is taken as the starting character position of the string from left to right. If ***expr1*** evaluates to 0, the starting character position is 1. For example, if **STRING** is **"HI THERE"**, then   |  | | --- | | STRING [2,2 ] = " OW " |   produces **"HOWTHERE"**.  If ***expr1*** evaluates to a number greater than the length of the string, then the replacement string is appended at the end. |
| ***expr1* < 0** | If ***expr1*** is negative, it is taken as the starting character position from right to left. For example, if **STRING** is **"HI THERE"**, then   |  | | --- | | STRING [-2,2 ] = " OW " |   produces **"HI THEOW"**. If the absolute value of ***expr1*** is greater than the length of the string and ***expr1*** is negative, it behaves as if ***expr1*** were 0. |
| ***expr2* >= 0** | If ***expr2*** is nonnegative, it is taken as the length of the string to be replaced. Note that this length does not have to correspond with the length of the replacement string. If ***expr2*** evaluates to zero, then the replacement string should be inserted without replacing any characters in the string. (If ***expr1*** is negative, it is inserted to the left of that position; otherwise it is inserted to the right.) |
| ***expr2* < 0** | If ***expr2*** is negative, it is taken as the ending character position of the string portion to be replaced, counting from right to left. For example, if **STRING** is **"HI THERE"**, then:   |  | | --- | | STRING [2, -2 ] = "OW" |   produces **"HOWE"**. Similarly, **STRING [-2, -2 ] = "OW "** produces **"HI THERWE"**. |

If the positions specified by ***expr1*** and ***expr2*** overlap, characters are repeated in the resulting string. For example, if **STRING** is **"HI THERE"**, then:

|  |
| --- |
| STRING[7,-7] = "OW" |

produces:

|  |
| --- |
| "HI THEOW THERE" |

With the statement:

|  |
| --- |
| DIGITS[n , m] = "XX" |

and the variable **DIGITS** containing **"1234567890"**, the behavior of the substring assignment statement is summarized by this table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***n* > 0** | ***n* = 0** | ***n* < 0** |
| ***m* > 0** | Starting at position ***n***, replace the next ***m*** characters.   |  | | --- | | DIGITS [3,4]="XX" |   results in:   |  | | --- | | DIGITS= "12XX7890" | | Same as ***n*** **=1**: replace the first n characters.   |  | | --- | | DIGITS [0,4]="XX" |    results in:   |  | | --- | | DIGITS= "XX567890" | | Starting at the ***n***th position from the end of the string, replace the next ***m*** characters.   |  | | --- | | DIGITS [-6,4]="XX" |    results in:   |  | | --- | | DIGITS= "1234XX90" | |
| ***m* = 0** | Insert the replacement string at position ***n***, with no characters deleted.   |  | | --- | | DIGITS [3,0]="XX" |   results in:   |  | | --- | | DIGITS = "12XX34567 890" | | Same as ***n*** **=1**: insert the replacement string at the beginning of the original string, with no characters deleted.   |  | | --- | | DIGITS [0,0]="XX" |   results in:   |  | | --- | | DIGITS = "XX1234567 890" | | Starting at the ***n***th position from the end of the string, insert the replacement string with no characters deleted.   |  | | --- | | DIGITS [-6,0]="XX" |    results in:   |  | | --- | | DIGITS= "12345XX67890" | |
| ***m* < 0** | Replace all characters from the ***n***th position from the beginning of the string up to the ***n***th position from the end of the string.   |  | | --- | | DIGITS [3,-4]="XX" |   results in:   |  | | --- | | DIGITS= "12XX890" | | Same as ***n*** **=1**: replace all characters from the first position up to the ***m***th position from the end of the string.   |  | | --- | | DIGITS [0,-4]="XX" |   results in:   |  | | --- | | DIGITS= "XX890" | | Starting at the ***n***th position from the end of the string, replace all characters up to the ***m***th position from the end of the string.   |  | | --- | | DIGITS [-6,-4]="XX" |    results in:   |  | | --- | | DIGITS= "1234XX890" | |

Example

In this application, a full name in the string variable **NAME** is reduced to the first initial and last name. (**EQUATE** statements are used to differentiate a blank space from the null string for readability.)

|  |
| --- |
| EQUATE BLANK TO " ", NIL TO ""  NO. OF WORDS = DCOUNT (NAME, BLANK )  SURNAME = FIELD (NAME, BLANK, NO. OF WORDS)  POSITION = COL1( ) -1  NAME [2, POSITION] = BLANK |

## @ Function

The **@** function generates the screen format control sequences for a terminal.

Format

|  |
| --- |
| @(*col*)  @(*col*,*row*)  @(-*code*) |

Parameter(s)

|  |  |
| --- | --- |
| ***col*** | An expression to be taken as the column (x-coordinate) of the position desired. |
| ***row*** | An expression to be taken as the row (y-coordinate) of the position desired. |
| ***-code*** | An expression to be taken as a numeric code signifying a specific effect, such as clearing the screen. Codes are listed below. |

Description

All terminals have built-in command sequences which move the cursor to a particular position, clear the screen, place text in reverse video, etc. The **@** function returns the proper command sequence for performing many terminal control functions. When this function is within a terminal output statement (**PRINT**, **CRT**, **DISPLAY**, or **SEND**), the terminal is sent the command string and acts accordingly.

When used in a terminal output statement, the **@** syntax is treated as any block of text: it may be combined with other blocks of text (including other calls of the **@** function) with the concatenation operator (**:**), and the carriage return and linefeed may be suppressed with a trailing colon.

There are two forms of the **@** function.

* The first form, **@(*col*,*row*)**, returns the command string for moving to a specified column and row. Although both expressions should be within the ranges of the particular display screen (usually either 79 or 131 columns by 23 rows, with 0,0 as the upper left corner of the screen), this is not enforced. If the row expression row evaluates to a value greater than the number of rows on the screen, it defaults to the last row on the screen (usually row 23); if the row expression evaluates to a value less than zero, it defaults to the top row of the screen (row 0). If the row expression is omitted, the current row is assumed.

Similarly, if the column expression col evaluates to a value greater than the number of columns on the screen, the cursor goes to the last column of the screen. Note, however, that if the column expression evaluates to something less than zero, the leading minus sign causes it to assume one of the special codes of the **@** function listed below, and the row specification is ignored.

* The other form of the **@** function, **@(*–code*)**, uses special codes, each preceded by a minus sign. The codes are as follows:

|  |  |
| --- | --- |
| **Code** | **Description** |
| **@(-1)** | Clear screen and position cursor at home (upper left corner). |
| **@(-2)** | Position cursor at home (upper left corner). Same as **@(0,0)**. |
| **@(-3)** | Clear from cursor position to end of the screen. |
| **@(-4)** | Clear from cursor position to end of current line. |
| **@(-5)** | Begin blinking field. |
| **@(-6)** | End blinking field. |
| **@(-7)** | Begin protected field. Data in this field cannot be overwritten. |
| **@(-8)** | End protected field. |
| **@(-9)** | Backspace one character. |
| **@(-10)** | Move cursor up one line. |
| **@(-11)** | Begin protected field. |
| **@(-12)** | End protected field. |
| **@(-13)** | Begin reverse video mode. |
| **@(-14)** | End reverse video mode. |
| **@(-15)** | Begin underline field. |
| **@(-16)** | End underline field. |
| **@()** function codes 301 - 399 below support calling Windows Printer API methods. Two types of Windows Printer **@()** function codes are provided:   * **@()** function codes 301 - 339 perform setup tasks in preparation for a print job. * **@()** function codes 340 - 399 perform the actual printing.   **WARNING—**Although multiple printing **@()** function codes can be concatenated, do not concatenate a setup **@()** function code with another setup **@()** function code or a printing **@()** function code. Setup **@()** function codes should always be isolated in a single statement.  **NOTE**—See the Windows GDI topic in Microsoft MSDN Library for more information on the Windows Printer methods. | |
| **@(-301)** | Load object's x-coordinate position. |
| **@(-302)** | Load object's y-coordinate position. |
| **@(-303)** | Load object's selected point number. |
| **@(-304)** | Load object's selected rectangle number. |
| **@(-305)** | Load object's brush on draw rectangle. |
| **@(-311)** | Load font's height of character. |
| **@(-312)** | Load font's escapement angle. |
| **@(-313)** | Load font's orientation angle. |
| **@(-314)** | Load font's weight of font. |
| **@(-315)** | Load font's italic flag. |
| **@(-316)** | Load font's underline flag. |
| **@(-317)** | Load font's strike out flag. |
| **@(-318)** | Load font's character set. |
| **@(-319)** | Load font's output precision. |
| **@(-320)** | Load font's pitch and family. |
| **@(-321)** | Load font's index of font name string. |
| **@(-322)** | Load image’s path to bitmap file string. |
| **@(-324)** | Load brush's style. |
| **@(-325)** | Load brush's color. |
| **@(-326)** | Load brush's hatch style. |
| **@(-327)** | Load pen's style. |
| **@(-328)** | Load pen's width. |
| **@(-329)** | Load pen's color. |
| **@(-330)** | Load point's x-coordinate. |
| **@(-331)** | Load point's y-coordinate. |
| **@(-332)** | Load rectangle's left x-coordinate. |
| **@(-333)** | Load rectangle's top y-coordinate. |
| **@(-334)** | Load rectangle's right x-coordinate. |
| **@(-335)** | Load rectangle's bottom y-coordinate. |
| **@(-336)** | Load image's raster opcode. |
| **@(-337)** | Load image's stretch opcode. |
| **@(-340)** | Create font. |
| **@(-341)** | Create brush. |
| **@(-342)** | Create hatch brush. |
| **@(-343)** | Create solid brush. |
| **@(-344)** | Create pen. |
| **@(-345)** | Defines the default font. |
| **@(-346)** | Selects the default font. |
| **@(-347)** | Select font. |
| **@(-348)** | Select brush. |
| **@(-349)** | Select pen. |
| **@(-350)** | Set x-coordinate position. |
| **@(-351)** | Set y-coordinate position. |
| **@(-352)** | Set text color. |
| **@(-353)** | Set background color. |
| **@(-354)** | Set background mode. |
| **@(-355)** | Set text align. |
| **@(-356)** | Set text extra spacing. |
| **@(-357)** | Sets the page orientation.  For portrait orientation: **@(-357,1)**  For landscape orientation: **@(-357,2)**  Since the orientation affects the entire printed page, this function must be included at the beginning of your **print** statement. |
| **@(-358, *x*)** | Sets the number of lines on a print page where ***x*** is the number of lines. |
| **@(-361)** | Draw rectangle with brush. |
| **@(-362)** | Fill rectangle with brush. |
| **@(-363)** | Draw line. |
| **@(-364)** | Draw ellipse. |
| **@(-365)** | Draw arc. |
| **@(-366)** | Draw polygon. |
| **@(-367)** | Draw bitmap - start. |
| **@(-368)** | Draw bitmap with path - start. |
| **@(-369)** | Draw bitmap - end. |

The **@** function generates the command string for the terminal being used at run time, according to the current terminal type defined by the **TERM** command. The most important thing to grasp about the **@** function is that all it does is generate a string of control characters, which happen to trigger a unique response when they are sent to the screen.

In mvBASIC, there is a statement called **INPUT @**, which among other things allows input to be prompted for at a particular coordinate of the screen. The **INPUT @** statement, however, does not provide for any of the screen control codes listed above, only for moving the cursor. If the format masking properties of the **INPUT @** statement are not being taken advantage of, the same effect might be achieved by preceding a standard **INPUT** statement with a **PRINT** statement which uses the **@** function directly.

The **CALL** and **ENTER** statements also recognize the **@** sign in their syntax lines, to signify that the name of the program to be called or entered is kept in a variable. This use of the **@** symbol, however, should not be confused with its use in the **@** function.

Examples

To clear the screen for a program, the code would read:

|  |
| --- |
| PRINT @(-1) |

To print the words **QUIT?** at the bottom of the screen, the code would read:

|  |
| --- |
| PRINT @(0,23) : "QUIT?" |

In the next example, the **@** function is used within a **PRINT** statement to clear the screen and prompt the user at position **(30,10)** on the screen. If the user supplies an invalid answer, an error message appears on the bottom of the screen in blinking mode, and the previous invalid answer is erased. The operator is then prompted again until a valid answer is provided.

|  |
| --- |
| PRINT @(-1) :  CLEAR.ANSWER = @(30,10) : @(-4)  PRINT @(10,10) : " ENTER A NUMBER: " : @(30,10):  LOOP     INPUT NUMBER  :     IF NUM(NUMBER) THEN        VALID = 1        GOSUB COMPUTE     END ELSE        VALID = 0        PRINT @(0,23) : @(-5) : " NON-NUMERIC INPUT.":        PRINT "ENTER A NUMBER" : @(-6) : @(-4):        PRINT CLEAR.ANSWER :     END  UNTIL VALID DO REPEAT |

These techniques are used in the application:

* All **PRINT** and **INPUT** statements are followed by a colon to suppress the automatic carriage return and linefeed. This provides more control over the screen: in auto-scroll mode, a linefeed causes all lines above the cursor position to scroll up one line.
* The sequence to clear the previous answer is placed in a variable **CLEAR.ANSWER**, which is later sent to the screen with the **PRINT** statement.

The example below illustrates a program that prints text to a specified location (x,y coordinates) on the paper.

|  |
| --- |
| PRINT\_AT\_LOC  001 \*  002 \*\* Set output to printer  003 \*  004      PRINTER ON  005 \*  006 \*\* Set location (x,y coordinate)  007 \*  008      PRINT @(-350,600): @(-351,900):  009 \*  010 \*\* Print to specified location  011 \*  012      PRINT "PRINT AT SPECIFIED LOCATION"  013 \*  014 \*\* Reset output to printer  015 \*  016      PRINTER OFF  017 \*  018      STOP |

The example below illustrates a program that prints an ellipse graphic.

|  |
| --- |
| PRINT\_ELLIPSE  001 \*  002 \*\* Set output to printer  003 \*  004      PRINTER ON  005 \*  006 \*\* Load brush fields  007 \*  008      x = @(-324,0) ;\* style  009      x = @(-325,12237498) ;\* color  010      x = @(-326,0) ;\* hatch style  011 \*  012 \*\* Create brush and assign as brush number 3  013 \*  014      PRINT @(-341,3):  015 \*  016 \*\* Select brush number 3 as current brush  017 \*  018      PRINT @(-348,3):  019 \*  020 \*\* Set rectangle 2 (left, top, right, bottom)  021 \*  022      x = @(-304,2)  023      x = @(-332,900)  024      x = @(-333,800)  025      x = @(-334,2200)  026      x = @(-335,1100)  027 \*  028 \*\* Draw ellipse with rectangle 2  029 \*  030      PRINT @(-364,2):  031 \*  032 \*\* Set location (x,y coordinate)  033 \*  034      PRINT @(-350,1200): @(-351,900):  035 \*  036 \*\* Print line at specified  037 \*\* location within ellipse  038 \*  039      PRINT "PRINT ELLIPSE"  040 \*  041 \*\* Reset output to printer  042 \*  043      PRINTER OFF  044 \*  045      STOP |

The example below illustrates a program that prints text using a specific font.

|  |
| --- |
| PRINT\_CREATE\_SELECT\_FONT  001 \*  002 \*\* Set output to printer  003 \*  004      PRINTER ON  005 \*  006 \*\* Load font fields  007 \*  008      x = @(-310,265) ;\* character height  009      x = @(-311,100) ;\* character width  010      x = @(-312,0) ;\* escapement angle  011      x = @(-313,0) ;\* orientation angle  012      x = @(-314,700) ;\* weight  013      x = @(-315,0) ;\* italic flag  014      x = @(-316,1) ;\* underline flag  015      x = @(-317,0) ;\* strike out flag  016      x = @(-318,1) ;\* character set  017      x = @(-319,0) ;\* output precision  018      x = @(-320,0) ;\* pitch and family  019      x = @(-321,2) ;\* font name  020 \*  021 \*\* Create font and assign as font number 1  022 \*  023      PRINT @(-340,1):  024 \*  025 \*\* Select font number 1 as current font  026 \*  027      PRINT @(-347,1):  028 \*  029 \*\* Set location (x,y coordinate)  030 \*  031      PRINT @(-350,600): @(-351,900):  032 \*  033 \*\* Print line using selected font  034 \*  035      PRINT "PRINT LINE USING SELECTED FONT"  036 \*  037 \*\* Reset output to printer  038 \*  039      PRINTER OFF  040 \*  041      STOP |

## ABORT Statement

The **ABORT** statement terminates the current program and returns the user to the TCL prompt, regardless of the environment in which the program was executed.

Format

|  |
| --- |
| ABORT[*errmsg* [,*parameter1*, *parameter2*, ...]] |

Parameter(s)

|  |  |
| --- | --- |
| ***errmsg*** | An integer corresponding to an error message from the system message file (ERRMSG). The message is output upon termination of the program. See the *Command Reference* section for a list of these messages. |
| ***parameter1*, *parameter2*, ...** | Parameters to be passed to the error message. |

Description

The **ABORT** statement differs from the **STOP** statement in that a **STOP** statement returns control to the calling environment (often a Proc), whereas **ABORT** terminates all calling environments as well as the mvBASIC program. In general, the **ABORT** statement should be used for abnormal terminations of a program, whereas a **STOP** statement should be used for normal terminations.

Example

This example demonstrates how the **ABORT** statement may be used to terminate a program on failure to open a file:

|  |
| --- |
| OPEN 'CUSTOMERS' TO CUSTFILE ELSE     ABORT 201, 'CUSTOMERS'  END   .   .   . |

If the **CUSTOMERS** file is not found, the user receives this message and is returned directly to the TCL prompt:

|  |
| --- |
| [201] 'CUSTOMERS' IS NOT A FILE NAME  > |

## ABS Function

The **ABS** function returns the absolute value of the given expression.

Format

|  |
| --- |
| ABS(expr) |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | An expression evaluating to a numeric value. |

Description

The absolute value of any expression is defined mathematically as its positive value; that is, the difference between itself and 0. The absolute value of any positive expression is itself, and the absolute value of a negative expression is calculated by reversing the sign—that is; the absolute value of -*n*  is n. If ***expr*** does not evaluate to a numeric value, the **ABS** function returns 0.

Example

If the variable **NUMBER** contains -1.732, then

|  |
| --- |
| ABS(NUMBER) |

returns 1.732.

In the next application the **ABS** function is used to discover an error in bookkeeping. Note that in the printing of the discrepancy, a form of the **FMT** format function has been used to make the data more readable.

|  |
| --- |
| DIFF = ABS(PRICE - COST - PROFIT)  IF DIFF THEN     PRINT "ERROR OF  " : DIFF " 2,$ " : " . PLEASE CHECK."  END |

If the **PRICE** has been established as 9.99, the **COST** is 4.25, and the estimated **PROFIT** has been set at 5.75, the resulting output is:

|  |
| --- |
| ERROR OF  $0.01. PLEASE CHECK. |

## ALPHA Function

The logical function **ALPHA** evaluates an expression to determine if it is a string containing only alphabetic characters.

Format

|  |
| --- |
| ALPHA(*expr*) |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | String expression to be tested. |

Description

The **ALPHA** function determines whether the expression is an alphabetic or nonalphabetic string. If the expression contains the characters A through Z or a through z (ASCII 65 - 90, 97 - 122), it evaluates to true and a value of 1 is returned. If the expression contains any other character (such as numeric or special characters), it evaluates to false and a value of 0 is returned.

Examples

If the variable **NAME** contains **"HENRY FRENKL"**, then:

|  |
| --- |
| ALPHA(NAME) |

returns 1. However, if **NAME** contains **"HENRY FRENKL4th"**, the **ALPHA** function returns 0.

In the next application, airline reservations require the traveler’s starting point and final destination. The travel agent must enter these with the 3-letter code assigned to airports.

|  |
| --- |
| PROMPT " "  PRINT @(-1)  PRINT @(2,2) : "ENTER FLIGHT DATE: ":  INPUT @(30,2) DATE "D"  PRINT @(2,4) : "ENTER FLIGHT NUMBER: ":  INPUT @(30,4) FLTNO,3 "0"  PRINT @(2,6) : "STARTING POINT: ":  LOOP     INPUT @(30,6) START,3     IF NOT ( ALPHA( START )) THEN        INPUTERR "PLEASE ENTER 3-LETTER        AIRPORT CODE."     END  UNTIL ALPHA(START) DO  REPEAT  PRINT @(2,8) : "FINAL DESTINATION: ":  LOOP     INPUT @(30,8) DEST,3     IF NOT( ALPHA( DEST )) THEN        INPUTERR "PLEASE ENTER 3-LETTER        AIRPORT CODE."     END  UNTIL ALPHA(DEST) DO  REPEAT |

## ASCII Function

The **ASCII** function converts a string in EBCDIC code into ASCII code.

Format

|  |
| --- |
| ASCII(*expr*) |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | A expression evaluating to the string to be converted. |

Description

The **ASCII** function converts each character of the given expression from its EBCDIC representation value to its ASCII representation value. It is the inverse to the **EBCDIC** function in mvBASIC.

The **ASCII** function does not convert a character to its numeric ASCII value, or vice versa. For that purpose, the **SEQ** and **CHAR** functions should be used. See [SEQ Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/seq_function.htm) and [CHAR Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/char_function.htm) for more information.

See [Appendix B: List Of ASCII Codes](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/appendix_b_list_of_ascii_codes.htm) for a full listing of ASCII codes.

Example

In this application, data is read from a tape which has been written in EBCDIC code. The **ASCII** function converts it into ASCII code.

|  |
| --- |
| READT STRING ELSE   .   .   .  END  STRING = ASCII(STRING) |

## ASSIGNED/UNASSIGNED Function

The mvBASIC intrinsic function **ASSIGNED( )** returns 1 (true) if the variable enclosed between the parentheses is currently assigned and 0 (false) if it is not. **UNASSIGNED( )** returns 0 if the variable is currently unassigned, and 1 otherwise.

Description

One use of these functions is when an application attempts to open files only once, keeping the file variables in a **COMMON** block. It is much more effective to modify the **OPEN** statements in existing programs to say:

|  |
| --- |
| IF UNASSIGNED(*filevar*) THEN OPEN ... TO *filevar* ... |

rather than change the application to open all files as soon as the user logs on and remove all subsequent **OPEN** statements from all other existing programs.

## ATTACH TAPE DEVICES/ CHANGE BLOCK-SIZE Statement

Use these statements to attach tape devices or change tape attachment blocksizes.

Format

|  |
| --- |
| ATTACH *devunit*,*blocksize* THEN/ELSE ... |

Description

The **ELSE** clause is taken if the required **ATTACH** was not successfully completed. In such a case, **SYSTEM(0)** provides the reason for failure as follows:

1. Invalid tape type specified.
2. No such tape device/unit configured.
3. Someone else is using this device/unit.

## AUX ON/AUX OFF Statement

**AUX ON** selects the auxiliary printer as the output device for all printer output until an **AUX OFF** is issued.

Data which would have gone to the spooler without the **AUX ON**, instead comes out of the auxiliary port. Data which would have displayed on the terminal continues to do so.

## BLOCK/UNBLOCK Statement

mvBASIC provides facilities for converting between variable length data structures and fixed length data structures.

For example, **BLOCK** converts a dynamic array into a fixed length record, automatically handling issues such as adding leading zeros or trailing spaces. It also allows the user to convert decimal or hexadecimal numbers into binary fields within the fixed length data structure. **UNBLOCK** is the converse of **BLOCK** and creates variable length structures from fixed length data structures.

**BLOCK** and **UNBLOCK** may also be used with great effect to handle foreign tapes created with more than one record per tape block, or when **IMPORT**ing or **EXPORT**ing formatted data to/from UNIX or DOS.

Description

**BLOCK** and **UNBLOCK** statements use a pre-defined structure to control the way in which they operate. This structure must be created as a dynamic array, with one attribute per field within the structure. Each attribute specifies the starting position of the field within the fixed length record, its length, and its type, separated by commas.

Examples

If working with an 80 byte card image and the user wishes to make attribute 5 of the dynamic array replace a customer name stored in bytes 51 through 70 of the card image (or visa versa), attribute 5 of the structure would be:

|  |
| --- |
| 51,20,L |

Define the field as type **L** so that trailing spaces are automatically trimmed if unblocking the card image to a dynamic array and automatically appended if blocking the dynamic array to the card image.

If using an **R** type, this trims or inserts leading zeros as appropriate.

Type **B** is used only when working with binary numbers in the fixed length record. It converts between a binary number there and a normal numeric character string in the dynamic array.

Type **X** is similar to type **B**, but works with binary numbers in the fixed length record, and hexadecimal character strings in the dynamic array. If no type code is specified, the data is moved without conversion.

If the structure has errors in it, such as referencing byte 0 of the fixed length record, moving zero bytes, or trying to extract data from beyond the end of the fixed length record, the **BLOCK** and **UNBLOCK** commands which use the structure set **SYSTEM(0)** to a value other than zero.

A good way of testing a structure is to take the original data, **BLOCK** it, and immediately **UNBLOCK** it using the same structure, verifying that the original data is recovered. Then do the same again first **UNBLOCK**ing then **BLOCK**ing. Both tests should regenerate the original data, or at least, the user should be able to explain all differences.

Format of BLOCK

|  |
| --- |
| BLOCK *fixed* FROM *dynamic* USING *structure* |

Parameter(s)

|  |  |
| --- | --- |
| ***fixed*** | Fixed length structure that is being created. |
| ***dynamic*** | Contains the data being used to create it. |
| ***structure*** | Contains the template controlling where the data from dynamic are placed within fixed and what conversions (if any) are performed on it during this process. |

If the command works correctly, **SYSTEM(0)** is set to 0. If the command works incorrectly during this process (almost always because of a bad structure), **SYSTEM(0)** specifies the type of error.

Format of UNBLOCK

|  |
| --- |
| UNBLOCK *dynamic* FROM *fixed* USING *structure* |

Parameter(s)

|  |  |
| --- | --- |
| ***dynamic*** | The dynamic array that is being created. |
| ***fixed*** | Contains the fixed length data being used to create it. |
| ***structure*** | Contains the template controlling where the data from fixed are placed within dynamic and what conversions (if any) are performed on it during this process. |

If the command works correctly, **SYSTEM(0)** is set to 0. If the command works incorrectly during this process (almost always because of a bad structure), **SYSTEM(0)** specifies the type of error.

## BREAK Statement

The **BREAK** statement allows the Break Inhibit Counter of a program to be incremented or decremented, thus controlling access to the Debugger.

Format

|  |
| --- |
| BREAK[KEY] ON | OFF |

Parameter(s)

|  |  |
| --- | --- |
| **ON** | Decrements the Break Inhibit Counter by 1. |
| **OFF** | Increments the Break Inhibit Counter by 1. |

Description

While a program is being executed, pressing **BREAK** normally transfers control from the program to the Debugger. The **BREAK** statement gives the programmer control over this feature.

The **BREAK** statement does not directly toggle the **BREAK** feature on and off, but increments and decrements the Break Inhibit Counter. The counter is usually set to 0, meaning that the **BREAK** feature is enabled. Each **BREAK OFF** statement increments the counter by 1, and each **BREAK ON** statement decrements the counter by 1. When the counter is set to any number other than 0, the **BREAK** feature is off. Thus if two **BREAK OFF** statements have been used in a program, the counter is set to 2, and two **BREAK ON** statements are necessary to return the counter to 0 and re-enable the **BREAK** key.

Using a counter instead of directly turning **BREAK** on and off simplifies situations where a program calls another program or external subroutine. Using a counter ensures that the status of the **BREAK** key in the calling program is maintained. This, of course, is dependent on each **BREAK OFF** statement being paired with a **BREAK ON** statement before the end of the program or subroutine.

During a **SLEEP** or **RQM** statement, pressing **BREAK** not only enters the Debugger but also disables the sleep. If the Break Inhibit Counter is set, the Debugger is not entered, but the sleep statement is still interrupted. Thus the **BREAK** key may also be used to terminate unwanted sleeps during the execution of a program.

The Debugger may also be entered through a run-time error or by encountering a **DEBUG** statement. See [Using the mvBASIC Debugger](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/using_the_mvbasic_debugger.htm) for more information.

Examples

To turn the break feature off for a program, the code would read:

|  |
| --- |
| BREAK OFF |

At the end of the program, the break feature may be reinstated with:

|  |
| --- |
| BREAK ON |

In the next application, a quiz program gives 60 seconds for the user to answer a question. To answer the question before the time is up, the user is allowed to press **BREAK**. A **BREAK OFF** statement is used to turn off the debugging feature of the **BREAK** key during the sleep, so that pressing **BREAK** interrupts the sleep but does not enter the Debugger. The **INPUTIF** statement is used after the sleep to examine the type-ahead and determine if a response was entered.

|  |
| --- |
| ITEM = RND(99)  READ QUESTION FROM QUESTFILE,ITEM ELSE     PRINT "ERROR IN READING " : ITEM     STOP  END  PRINT @(0,23) : "ENTER ANSWER, PRESS ENTER AND  BREAK." :  PRINT QUESTION <1> :  BREAK OFF  SLEEP 60  BREAK ON  INPUTIF ANSWER THEN     GOSUB EVAL  END ELSE     PRINT @(-1) : "NOT ANSWERED IN TIME. -3     POINTS."     POINTS - = 3  END |

## CALL Statement

The **CALL** statement transfers control from a main program to an external subroutine.

Format

|  |
| --- |
| CALL *name* (*expr1*, *expr2*, *expr3*, ...)  CALL @ *var* (*expr1*, *expr2*, *expr3*, ...) |

Parameter(s)

|  |  |
| --- | --- |
| ***name*** | Name of the subroutine to be called. |
| ***expr…*** | Values to be passed to the cataloged subroutine. If one of the values is an array variable, it must be preceded by the **MAT** keyword. |
| **@ *var*** | ***var*** is a variable which has been assigned the item-ID of the cataloged subroutine to be entered. |

Description

The **CALL** statement may be used to enter an external subroutine. An external subroutine is a subroutine that is compiled and cataloged separately from the programs that call it. When the ending **RETURN** statement of the subroutine is encountered, program control is returned to the original program at the line following the **CALL** statement.

The subroutine to which the **CALL** statement branches must be cataloged, unless the subroutine is in the same file as the main program. The first line of the subroutine must contain the **SUBROUTINE** statement. Control is returned to the main program when a **RETURN** is encountered in the subroutine which does not correspond to a previous **GOSUB** within the same external subroutine. If there is no **RETURN** statement, control does not return to the main program.

Each of the parameters listed in the **CALL** syntax line is passed into the corresponding variable list on the **SUBROUTINE** syntax line. Other than their positions on the **CALL** and **SUBROUTINE** syntax lines, there is no correspondence between variable names in the calling program and subroutine.

An alternative way of passing variables between programs and subroutines is by using **COMMON** statements in both program and subroutine. See [COMMON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/common_statement.htm) for more information.

The **DATA** statement may be used to supply input that the subroutine might request. See [DATA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/data_statement.htm) for more information.

### Passing Arrays

When arrays are being passed from the main program to a subroutine, the array name must be preceded by the **MAT** keyword and there must be a one-to-one correspondence to the elements being passed. For example, to pass the 3x4 matrix **MATRIX**, type:

|  |
| --- |
| CALL SUBR( MAT MATRIX ) |

The **MATRIX** array must be previously dimensioned in the program with the **DIM** (dimension) statement.

In the subroutine **SUBR**, the corresponding dimensioned array must also be dimensioned. Note, however, that the corresponding arrays do not need to have the same dimensions, as long as they have the same number of elements. The first 2 lines of the subroutine **SUBR** might read:

|  |
| --- |
| SUBROUTINE SUBR( MAT ARRAY1)  DIM ARRAY1(6,2) |

If the 3x4 matrix **MATRIX** in the main program contains:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| RED | BLUE | GREEN | YELLOW |
| A | B | C | D |

then when it is passed to **ARRAY1**, the 6x2 matrix contains:

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| RED | BLUE |
| GREEN | YELLOW |
| A | B |
| C | D |

Examples

To call the subroutine **ADDTHEM**, passing variables **A**, **B**, and **C**:, the calling line in the main program would read:

|  |
| --- |
| CALL ADDTHEM(A, B, C ) |

The first line of the source code for **ADDTHEM** might then read:

|  |
| --- |
| SUBROUTINE ADDTHEM(X, Y, Z ) |

Variable **A** is passed to variable **X**, **B** is passed to **Y**, and **C** is passed to **Z**. When the subroutine has finished, these values are passed in the opposite direction.

## CASE Construct

The **CASE** construct performs a conditional selection of a sequence of statements.

Format

|  |
| --- |
| BEGIN CASE     CASE *expr*  *statements*     CASE *expr*  *statements*        .        .        .  END CASE |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | An expression to be evaluated for its logical value. |
| ***statement*** | Statements to be executed if the previous ***expr*** had been tested to be logically true. |

Description

A **CASE** construct must begin with a **BEGIN CASE** statement and end with an **END CASE** statement. The **CASE** construct evaluates a series of conditions until one is true and executes a set of statements accordingly. The expressions in the **CASE** statements are evaluated sequentially for their logical value until a value of true is encountered. When an expression evaluates to true, the statements between the **CASE** statement and the next **CASE** statement are executed, and all subsequent **CASE** statements are skipped. Execution continues with the next sequential statement following the **END CASE** statement.

If none of the expressions evaluate to true, no action is performed, and program execution continues with the statement after the **END CASE** statement.

The **CASE** statement can usually replace multiple nested IF constructs: it is much more readable and easier to implement.

Example

To test a variable **NUMBER** for positive or negative value, the source code might read:

|  |
| --- |
| BEGIN CASE     CASE NUMBER > 0     PRINT "POSITIVE"     CASE NUMBER < 0     PRINT "NEGATIVE"     CASE 1     PRINT "ZERO"  END CASE |

Note that the third and last condition reads **CASE 1** instead of **CASE NUMBER = 0**. In this situation the two conditions are equivalent since the last condition would only be tested if the first two failed. **CASE 1** is often used as the last condition of a **CASE** statement, as a catch-all condition.

## CHAIN Statement

The **CHAIN** statement terminates execution of a program and executes a TCL command.

Format

|  |
| --- |
| CHAIN *command-expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***command-expr*** | Any command to be passed to TCL. |

Description

Like the **EXECUTE** statement, the **CHAIN** executes a TCL command. The **CHAIN** statement differs from the **EXECUTE** statement, however, in that it does not support any of **EXECUTE**’s features (such as capturing output or error messages), and it does not return to the program, but returns directly to the environment which called the program.

If the **CHAIN** statement is used to execute another program, parameters cannot be directly passed to the second program. However, if the I option (which suppresses initialization of all values) is used with the **RUN** command, the **COMMON** area may be used to pass parameters from one program to the next. See [COMMON Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/common_statement.htm) for more information.

The data stack may be used to supply input which the TCL command might request. See [DATA Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/data_statement.htm) for more information.

Example

To end a program by running another program, **WRAPUP**, the code might read:

|  |
| --- |
| CHAIN "RUN BP WRAPUP" |

## CHANGE Function

The **CHANGE** function swaps one string of characters for another.

Format

|  |
| --- |
| NEW=CHANGE(*Orig*,*Str1*,*Str2*{,*Occurs*{,*Start*}}) |

Description

**CHANGE( )** returns ***Orig*** with all occurrences of ***Str1*** changed to ***Str2***. If ***Occurs*** is omitted, all occurrences are assumed. If ***Start*** is omitted, swapping starts with the first occurrence.

It is perfectly acceptable for ***Str2*** to be null or a different length from ***Str1***.

|  |  |
| --- | --- |
| **NOTE** | **SWAP( )**, **CHANGE( )** and **EREPLACE( )** are synonyms. |

## CHAR Function

The **CHAR** function returns the character with the given ASCII decimal code.

Format

|  |
| --- |
| CHAR(*expr*) |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | An expression evaluating to a numeric value. |

Description

The **CHAR** function converts a decimal value to its corresponding ASCII character. It is particularly useful to access characters like attribute marks (**CHAR(254)**) and error bells (**CHAR(7)**). The **CHAR** function is commonly used in **EQUATE** statements.

The **SEQ** function acts as an inverse for the **CHAR** function, producing the ASCII value of a given character. See [SEQ Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/seq_function.htm) for more information.

See [Appendix B: List Of ASCII Codes](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/appendix_b_list_of_ascii_codes.htm) for ASCII character codes.

Example

To send an error bell to the screen, the code might read:

|  |
| --- |
| DISPLAY CHAR(7) |

## CLEAR Statement

The **CLEAR** statement assigns a value of 0 to all variables throughout the program.

Format

|  |
| --- |
| CLEAR |

Description

The **CLEAR** statement is generally used at the beginning of a program to set all previously assigned and unassigned values of variables to zero. This procedure avoids run-time errors for unassigned variables. If the **CLEAR** statement is used later in the program, any values that have already been assigned to variables (including array variables) are lost.

The **CLEAR** statement cannot be used to initialize only selected variables. If it is used, all variables in the program are initialized to 0.

|  |  |
| --- | --- |
| **NOTE** | The **CLEAR** statement is often used to prevent the run-time warning message which normally ensues when an unassigned variable is used. This practice, however, is not always desirable, since the unassigned variable message may be useful in detecting programmer errors (such as misspelled variable names). |

Example

In this application the **CLEAR** statement is used at the beginning of the program to initialize variables. Thus, when the previously unused variable **STOPNOW** is used as the loop control, no error message ensues.

|  |
| --- |
| CLEAR     .     .     .  LOOP UNTIL STOPNOW DO     .     .     .     PRINT "DO YOU WANT TO STOP (Y OR N)" :     INPUT ANSWER,1     IF ANSWER = "Y"  THEN STOPNOW = 1  REPEAT |

## CLEARCOMMON Statement

The **CLEARCOMMON** statement clears **NAMED COMMON** blocks.

Format

|  |
| --- |
| CLEARCOMMON |

Description

**NAMED COMMON** blocks are active from the time the user first references them until the process logs off. Usually this works well, but occasionally the user may wish to re-initialize them without logging the process off. **CLEARCOMMON** resets all variables in **NAMED COMMON** blocks to unassigned.

## CLEARDATA Statement

The **CLEARDATA** statement clears stacked input. **CLEARDATA** resets the DATA stack to empty.

Format

|  |
| --- |
| CLEARDATA |

## CLEARFILE Statement

The **CLEARFILE** statement empties the opened section of a file.

Format

|  |
| --- |
| CLEARFILE[*filevar*] |

Parameter(s)

|  |  |
| --- | --- |
| ***filevar*** | File variable to which the file had been opened. If ***filevar*** is not specified, the default file variable is used, which is the last file opened without an assigned file variable. |

Description

The **CLEARFILE** statement deletes all items in a previously opened data file. This statement does not delete the file itself, but it empties the opened section of a file completely.

|  |  |
| --- | --- |
| **NOTE** | The **CLEARFILE** statement cannot be used to empty the file dictionary, nor to delete individual data file items. |

Examples

To clear the data from a file opened to **DATAFILE**, the code might read:

|  |
| --- |
| CLEARFILE DATAFILE |

In the following application, the file **TRANS.LOGS** contains logs for all transactions during the week. The program fragment shown clears all logs at the request of the operator.

|  |
| --- |
| OPEN " TRANS.LOGS " TO LOGFILE ELSE     ABORT 201, " PRINTLOG "  END  PRINT " EMPTY ALL TRANSACTION LOGS (Y OR N) " :  INPUT ANSWER  IF ANSWER = " Y " THEN     CLEARFILE LOGFILE     PRINT " ALL TRANSACTION LOGS EMPTIED. "  END ELSE     PRINT " TRANSACTION LOGS UNTOUCHED. "  END |

## CLEARSELECT Statement

The **CLEARSELECT** statement clears the specified select-list.

Format

|  |
| --- |
| CLEARSELECT  CLEARSELECT *select.list* |

If no select list is specified, the default select-list is used.

Description

**CLEARSELECT** resets the specified select-list to empty, causing the next **READNEXT** command issued with that list to take the **ELSE** clause.

## COL1 Function

The **COL1** function returns the column position of the character immediately preceding the string returned by the most recent **FIELD** function.

Format

|  |
| --- |
| COL1( ) |

Description

After the execution of a **FIELD** function, the **COL1** function returns the column position immediately preceding the selected substring. Although the **COL1** function takes no arguments, the parentheses are required to identify it as a function.

If no **FIELD** function precedes the **COL1** function, a value of zero is returned. If the delimiter expression of the **FIELD** function is null or if the string is not found, the **COL1** function returns a zero value.

The **COL1**, **COL2**, **INDEX**, and **FIELD** functions, with the substring assignment statement (**[ ]=**), may be used to perform array processing for strings with delimiters other than the attribute mark (**CHAR(254)**), value mark (**CHAR(253)**), and subvalue mark (**CHAR(252)**).

Example

To determine the column position before the third word in a string **STRING**, the code might read:

|  |
| --- |
| WORD = FIELD(STRING, " ", 3)  POS = COL1( ) |

If **STRING** contains **"IT WAS TWENTY YEARS AGO TODAY"**, **WORD** contains **"TWENTY"** and **POS** contains **"7"**. With this information, the string may be cut off after the second word with:

|  |
| --- |
| STRING = STRING[1,POS] |

and **STRING** contains **" IT WAS "**.

In the next application, the **NAMES** string contains a list of names separated by commas (**,**). To replace a name, the substring assignment statement is used, but the **COL1** function is necessary to determine where the replacement should start.

|  |
| --- |
| CURR.NAME = FIELD( NAMES, ",", 2)  IF NEW.NAME <> "" THEN     BEG.COL = COL1( ) + 1     LENGTH = LEN( CURR.NAME )     NAMES[BEG.COL, LENGTH] = NEW.NAME  END |

## COL2 Function

The **COL2** function returns the column position of the character immediately after the string returned by the most recent **FIELD** function.

Format

|  |
| --- |
| COL2( ) |

Description

After the execution of a **FIELD** function, the **COL2** function returns the column position immediately after the selected substring. Although the **COL2** function takes no arguments, the parentheses are required to identify it as a function.

If no **FIELD** function precedes the **COL2** function, a value of zero is returned. If the delimiter expression of the **FIELD** function is null or if the string is not found, the **COL2** function returns a zero value.

The **COL2**, **COL1**, **INDEX**, and **FIELD** functions, with the substring assignment statement, may be used to perform array processing for strings with delimiters other than the attribute mark (**CHAR(254)**), value mark (**CHAR(253)**), and subvalue mark (**CHAR(252)**).

Examples

To determine the column position before the third word in a string **STRING**, the code might read:

|  |
| --- |
| WORD = FIELD(STRING, " ", 3)  POS = COL2( ) |

If **STRING** contains **"IT WAS TWENTY YEARS AGO TODAY"**, then **WORD** contains **"TWENTY"** and **POS** contains **"14"**.

With this information the string may be cut off after the third word with:

|  |
| --- |
| STRING = STRING[ 1, POS ] |

and **STRING** contains **" IT WAS TWENTY "**.

## COMMON Statement

The **COMMON** statement is used to specify the sequence in which the listed variables are allocated space. It allows programs and external subroutines to access the same variables.

Format

|  |
| --- |
| COM[MON] [/*name*/] *var1* [, *var2* , ...] |

Parameter(s)

|  |  |
| --- | --- |
| ***name*** | Name (enclosed in **/**) of the common block where the variables are to be temporarily stored. A maximum of five named **COMMON** statements may be included in an mvBASIC program. Note that multiple **COMMON** statements in mvBASIC programs must be contiguous, and cannot be separated by executable statements. |
| ***var...*** | Names of the variables to be shared. ***var*** can be a simple variable, file variable, or array variable. |

Description

The **COMMON** statement provides a storage area for the listed variables which is accessible by other programs and by external subroutines. The variables may be defined using different names in separate programs and subroutines, but they must be defined in the same exact order. The **COMMON** statement must precede any use of the variables that it names during compilation.

Simple variables that have not been declared with a **COMMON** statement are allocated space as they appear, and array variables are allocated space after simple variables. By using a **COMMON** statement, the sequence in which they are allocated space is explicitly set, and other programs using the same **COMMON** area can access the same variables by position. By using **COMMON**, variables do not have to be supplied in **CALL** and **SUBROUTINE** statements, and programs runs more efficiently. **COMMON** may also be used for programs that have been linked via the **CHAIN** statement, as long as the I option is used with the **RUN** command to prevent reinitialization.

|  |  |
| --- | --- |
| **NOTE** | It is crucial that the number and order of variables which are listed in **COMMON** statements be consistent between programs and subroutines. It is not necessary, however, that the variables have the same definitions (see second example). Once a **COMMON** statement is changed, all other subroutines and programs using the same **COMMON** area need to be recompiled with the same change in **COMMON**. For that reason it is suggested that if the **COMMON** area is used, the same variable names be used in programs and subroutines and that the **COMMON** statement be placed in a library, to be read via an **$INCLUDE** or **$INSERT** statement: this way, a change needs to be made only once, although all related programs and subroutines still need to be recompiled. |

Arrays may be declared in a **COMMON** statement, with the same syntax as in a **DIMENSION** statement. If an array is declared by a **COMMON** statement, it should not also be declared in a **DIMENSION** statement, or an error will occur at compile-time.

Examples

If **PROGRAM1** contains the line:

|  |
| --- |
| COMMON A, B, ADDRESSES(3,3) |

and **SUBR2** contains:

|  |
| --- |
| COMMON X, Y, MATRIX(3,3) |

then, if **PROGRAM1** calls **SUBR2** with the line:

|  |
| --- |
| CALL SUBR2 |

variables **A** and **X** will be equivalent,  variables **B** and **Y** will be equivalent, and elements of the dimensioned arrays **ADDRESSES** and **MATRIX** will be equivalent.

Named Common Example

If **PROGRAM2** contains the line:

|  |
| --- |
| COMMON /FILES/ CUST, INV, ORDERS |

and **SUBR3** contains:

|  |
| --- |
| COMMON /FILES/ FVAR(3) |

then, if **PROGRAM2** calls **SUBR3** with the line:

|  |
| --- |
| CALL SUBR3 |

variables **CUST**, **INV**, and **ORDERS** will be equivalent to the elements of the dimensioned array **FVAR**, respectively.

|  |  |
| --- | --- |
| **NOTE** | The state of all variables declared in a named common statement are retained until the user logs off the system. |

## COMPARE Statement

The **COMPARE** statement enables the comparison of two strings.

Format

|  |
| --- |
| COMPARE *String1* TO *String2* {PRESENT *sameones*} {MISSING *diffones*} |

Description

**COMPARE** allows the comparison of the contents of two variables to find the matches and mismatches. This statement works its way along the elements (components) of ***String1***, seeing if they are present in ***String2***. If the **PRESENT** option was specified, the variable associated with it contains all the elements that appear in both lists. If the **MISSING** option was specified, the associated variable contains all the elements which appear in ***String1*** and not in ***String2***.

Example

|  |
| --- |
| COMPARE String1 TO String2  {PRESENT string 3 | MISSING String} |

## CONNECT/ DISCONNECT Statement

**CONNECT** and **DISCONNECT** may be used to direct terminal I/O to a specific line.

Format

|  |
| --- |
| CONNECT *line.no* THEN/ELSE ...  DISCONNECT THEN/ELSE ... |

Description

The user may connect to another line providing that line is currently logged off, that no one else is currently connected to that line, and that the process was not already connected to a different line.

Once a line has been connected to the process, all issued **INPUT**s and **PRINT**s affect that line and not the user’s. **CRT**s and **INPUT@**s continue to work with the user’s own line as they did previously.

**SYSTEM(14)**, which returns the number of characters in the type-ahead buffer, returns it for the connected line, not the user’s. **SYSTEM(13)** always returns the number of characters in the user’s type-ahead buffer.

**DISCONNECT** releases the connected line, allowing the user to once again use **PRINT** and **INPUT** to issue terminal I/O on the user’s process line. Terminating the user’s program automatically performs a **DISCONNECT**.

If the **CONNECT** fails, the **ELSE** clause is taken, and **SYSTEM(0**) gives the reason for the failure.

## CONSOLE Statement

Use **CONSOLE** to write messages to the operator's console.

Format

|  |
| --- |
| CONSOLE *expression* |

Parameter(s)

|  |  |
| --- | --- |
| ***expression*** | Appears on the console port |

Description

An operator's console displays activities such as users logging on and off, who uses the tape units, run-time aborts, etc. mvBASIC now allows the user to write messages to that console.

## CONVERT Function

The **CONVERT** function converts individual characters.

Format

|  |
| --- |
| NEW=CONVERT(*Original*,*String1*,*String2*) |

Description

**CONVERT( )** returns a modified version of the original string, with each character found in ***String1*** converted to the character in the same position in ***String2***.

It is perfectly acceptable for ***String2*** to be null or a different length from ***String1***. Where there is no corresponding character in ***String2***, the character in ***String1*** is deleted. Additional characters in ***String2*** are ignored.

## CONVERT Statement

The **CONVERT** statement may be used to replace characters in a string variable.

Format

|  |
| --- |
| CONVERT *expr1* TO *expr2* IN *var* |

Parameter(s)

|  |  |
| --- | --- |
| ***expr1*** | List of original characters to be converted. |
| ***expr2*** | List of characters to replace original characters. |
| ***var*** | String variable to be converted. |

Description

The **CONVERT** statement replaces every occurrence of each of the specified characters with the corresponding replacement character. It treats each expression as a list of characters, not as a string: the first character in ***expr1*** is replaced with the first character in ***expr2***, the second character in ***expr1*** is replaced with the second character in ***expr2***, and so on.

Every time a character listed in ***expr1*** appears in the string, it is replaced by the corresponding replacement character, regardless of how many times it appears. If ***expr1*** contains more characters than ***expr2***, the extra characters are deleted from the converted string. If the second expression contains more characters than the first, the extra characters in ***expr2*** are ignored. If a character is repeated in the first expression, only the first assignment is made and all subsequent assignments of that character are ignored.

Examples

If the variable **STRING** contains **"I LIKE IT"**, all **K**s may be changed to **V**s with:

|  |
| --- |
| CONVERT "K" TO "V" IN STRING |

The resulting string is **"I LIVE IT"**. However,

|  |
| --- |
| CONVERT "LIKE" TO "LOVE" IN STRING |

produces the string, **"O LOVE OT"**.

In the next application the **CONVERT** statement is used to turn a comma-separated list of names, **NAMES**, into a dynamic array, by converting each comma into an attribute mark.

|  |
| --- |
| EQUATE AM TO CHAR(254)     .     .     .  CONVERT "," TO AM IN NAMES |

As a comma-separated list, fields of the **NAMES** array may be deleted, inserted, or arranged only through a sequence of statements involving **FIELD,** **COL1( )**, etc. By converting commas to attribute marks, however, fields may be manipulated using the more powerful (and more intuitive) dynamic array functions.

## COS Function

The **COS** function returns the trigonometric cosine of the expression.

Format

|  |
| --- |
| COS(*expr*) |

Parameter(s)

|  |  |
| --- | --- |
| ***expr*** | Treated as an angle expressed as a numeric value in degrees. Values outside the range of 0 to 360 degrees are interpreted as modulo 360. |

Example

In this application the **COS** function is used with a standard trigonometric formula to calculate the sin of an angle without using the **SIN** function.

|  |
| --- |
| SINE = SQRT(1 - COS(ANGLE) \* COS(ANGLE))  PRINT " THE SINE IS CALCULATED AS : " : SINE |

## COUNT Function

The **COUNT** function determines how many times a character or string of characters occur within a specified string.

Format

|  |
| --- |
| COUNT(*string*,*chars*) |

Parameter(s)

|  |  |
| --- | --- |
| ***string*** | An expression evaluating to the string to be searched. |
| ***chars*** | An expression evaluating to the substring to be searched for and counted. |

Description

The **COUNT** function returns the number of times a substring is repeated within a string. The **COUNT** function returns zero if the substring is not found. If the substring is the null string, the function returns the number of characters in the string minus one.

The **COUNT** function actually counts the number of starting points for the specified substring within the string. That is, for each character in the string, it determines whether an occurrence of the specified substring begins at that character. If it does, its return value is incremented by one. This means that if there are overlapping occurrences of the substring within the string, **COUNT** returns as many occurrences as it can find, regardless of whether the starting character is a part of a previous occurrence.

The **DCOUNT** function returns the number of fields separated by a given 1-character delimiter. See [DCOUNT Function](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/dcount_function.htm) for more information.

Example

To assign the variable **NUMS** to the number of times the substring **"ANA"** occurs in the string **"BANANA"**, type:

|  |
| --- |
| NUMS = COUNT("BANANA","ANA") |

Note that the two occurrences of **"ANA"** overlap.

## CRT Statement

The **CRT** statement sends data to the terminal display screen. It is identical to the **PRINT** statement except that it writes only to the terminal. The **DISPLAY** statement is identical to the **CRT** statement.

Format

|  |
| --- |
| CRT *print-expr*  DISPLAY *print-expr* |

Parameter(s)

|  |  |
| --- | --- |
| ***print-expr*** | A print expression, optionally combined with commas and colons to designate the format of the output. If ***print-expr*** is omitted, a blank line is output. See [DISPLAY Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/display_statement.htm) for information on the format for a print expression. |

Description

The **CRT** and **DISPLAY** statements cause data to be output to the terminal screen, regardless of whether a **PRINTER ON** statement has been executed. See [DISPLAY Statement](http://www3.rocketsoftware.com/rocketd3/support/documentation/mvb/32/refman/mvbasic/display_statement.htm) for more information on **CRT** and **DISPLAY**.

Example

To print the string **"HELLO…"** to the screen, the code might read:

|  |
| --- |
| CRT "HELLO…" |

or

|  |
| --- |
| DISPLAY "HELLO…" |

## CRT ON Statement

The **CRT ON** statement allows display characters on other lines.

Format

|  |
| --- |
| CRT ON *line#* *message* {ELSE ...} |

Parameter(s)

|  |  |
| --- | --- |
| ***line#*** | Valid line number that is not running a printer or background process. |
| **{ELSE ...}** | (Optional) If present, is taken if another process is attached to the specified line. |