GFK-0256

GE Fanuc Manual Series 90-30

MegaBasic Language Reference and Programmer's Guide Reference Manual

GE Fanuc Automation

Programmable Control Products

MegaBasic™ Language Reference and Programmer's Guide

Reference Manual

GFK-0256D *September 1994*

Warnings, Cautions, and Notes as Used in this Publication

Warning

Warning notices are used in this publication to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist in this equipment or may be associated with its use.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where equipment might be damaged if care is not taken.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware or software, nor to provide for every possible contingency in connection with installation, operation, or maintenance. Features may be described herein which are not present in all hardware and software systems. GE Fanuc Automation assumes no obligation of notice to holders of this document with respect to changes subsequently made.

GE Fanuc Automation makes no representation or warranty, expressed, implied, or statutory with respect to, and assumes no responsibility for the accuracy, completeness, sufficiency, or usefulness of the information contained herein. No warranties of merchantability or fitness for purpose shall apply.

Preface

MegaBasic is a powerful implementation of the BASIC language, which runs under twelve different operating systems and a host of different hardware configurations. One of the strengths of MegaBasic is that the language can be extended to support the underlying hardware.

Content of this Manual

Chapter 1. Introduction to MegaBasic: Provides an introduction to the MegaBasic language.

Chapter 2. MegaBasic Commands: Describes all the MegaBasic commands. It is organized into five sections: introduction, program entry and retrieval, editing and alteration, execution control and debugging, and information and control.

Chapter 3. Representing and Manipulating Numbers: Describes the concepts and use of numeric constants, variables, arrays, expressions, operators, functions, vector processing, and floating point systems.

Chapter 4. Representing and Manipulating Strings: Describes strings and how to represent and manipulate them in your programs.

Chapter 5. Data Definition and Assignment Statements: Describes statements which define data structures and move computational results between variables.

Chapter 6. Program Control Statements: Describes program control statements which allow you to change the course of execution to suit your processing requirements.

Chapter 7. I/O and System Interaction: Describes statements for accessing data files, for character device input and output, and for interacting with external system processes and services.

Chapter 8. User-Defined Subroutines: Describes concepts and techniques for building and using subroutines.

Chapter 9. MegaBasic Built-in Function Library: Describes the built-in functions in MegaBasic.

Chapter 10. Multiple Module Programs: Describes MegaBasic package concepts and supporting statements.

Appendix A. Error Messages: Describes error types and messages reported by MegaBasic.

Appendix B. Other Operating Systems: Describes how MegaBasic under other operating systems differs from the MS-DOS implementation described in chapters 1 through 10 of this manual.

Appendix C. Utilities and Other Software: Describes programs external to MegaBasic that perform functions useful to the development process.

Appendix D. Miscellaneous Information: Describes MegaBasic enhancements, reserved words and characters, code conversion tables, converting non-integer programs to use integers, and loading earlier programs.

Related PCM Publications

For more information, refer to these publications:

Series 90™ Programmable Coprocessor Module and Support Software User's Manual (GFK-0255): provides a general overview of the capabilities and operation of the Series 90 PCM modules.

Series 90™ PCM Development Software (PCOP) User's Manual (GFK-0487): describes how to use the PCM development software (PCOP) to develop applications for the PCM.

Series 90™ Quick Reference Guide (GFK-0260): outlines the steps involved in installing and operating the PCM.

Series 90™ PCM Support Software (TERMF) Quick Reference Guide (GFK-0655): outlines the steps involved in installing and operating TERMF.

Series 90™ PCM Development Software (PCOP) Quick Reference Guide (GFK-0657): outlines the steps involved in installing and operating PCOP.

Series 90™ -70 PCM Important Product Information (GFK-0351).

Series 90™ PCM Programmer Important Product Information (GFK-0352).

Series 90™ -30 PCM Important Product Information (GFK-0494).

Related Series 90 Publications

For more information, refer to these publications:

Series 90™ -70 Programmable Controller Installation Manual (GFK-0262).

Logicmaster™ 90-70 Programming Software User's Manual (GFK-0263).

Series 90™ -70 Programmable Controller Reference Manual (GFK-0265).

Series 90™ -30 Programmable Controller Installation Manual (GFK-0356).

Series 90™ -30/90-20 Programmable Controllers Reference Manual (GFK-0466).

Logicmaster™ 90 Series 90-30 and 90-20 Programming Software User's Manual (GFK-0467).

We Welcome Your Comments and Suggestions

At GE Fanuc Automation, we strive to produce quality technical documentation. After you have used this manual, please take a few moments to complete and return the Reader's Comment Card located on the next page.

Chapter 1 Introduction to MegaBasic

MegaBasic is a state-of-the-art high-performance **BASIC** that is specifically designed to support large applications, real-time processing and fast execution. The MegaBasic compiler further advances the speed of your programs so that they perform like optimized **PL/1,** C or **PASCAL** programs, without giving up the string processing, vector handling and other integrated high-level capabilities of extended **BASIC**. The primary distinguishing features of MegaBasic can be summarized as:

- Full access to all available memory while imposing few artificial limitations to its use. Program space, array space and string space independently have no fixed limits and can change *dynamically* during execution.
- 80286/386 protected-mode version available that provides up to 16 megabytes of memory for general program and data space under **MS-DOS**, with or without the presence of a *DOS-Extender.*
- Integrated *mode-less* programming development environment requiring little in the way of **CPU** and memory resources, while providing extensive *built-in* testing, analysis and debugging support. Compiler available for accelerated execution speed and global syntax verification.
- The small size of MegaBasic makes it ideally suited for integration into **ROM**s of small machines in *real-time* applications. Custom versions for proprietary applications are available by special arrangement.
- A rational syntax for commands, statements and functions that is easy to remember, making reference to the manual less frequent.
- Provides all the expected program control structures, including **FOR** loops with *multiple* ranges, **WHILE** and **REPEAT** loops, **CASE** statements, multi-line **IF** statements, multi-line procedures and functions with argument lists and varying numbers of parameters during execution, local variables within subroutines, etc.
- Dynamic linking (at *run-time)* of user-routine sets called *packages*, specifically designed to support very large modular applications that can exceed the capacity of available memory (see Chapter 10, Section 1). Packages can access other packages as needed.
- Support for true *asynchronous event-driven* processes, designed specifically for multi-tasking, background processing, instrumentation and other real-time process control applications (Chapter 7, Section 4). Full access to machine-level resources, such as I/O ports, absolute memory addresses and **INT**errupt calls with access to **CPU** registers.
- Support for *IEEE/80x87 binary* and *14-digit decimal* (**BCD** representation) floating point, 32-bit integers, extended arithmetic and mathematical operations with

automatic selection of *software vs. 80x87 math coprocessor* in IEEE versions (Chapter 3, Section 8). Multi-dimensional arrays of integers and real numbers have no set limit on how much memory they can use (Chapter 3, Section 1).

- Extended numeric assignment statements letting you assign values to variables *within* numeric expressions and to perform increment, decrement or other arithmetic operation on variables (e.g., X**+=Z,Q/=D,Y*=M**, etc.).
- A complete family of integrated arithmetic and mathematical *vector* operations for *dramatic reduction* in both execution time and notational complexity for matrix processing and other general sequential processing of integer and real numbers (Chapter 3, Section 7).
- Field structures let you assign *names* and *data types* to specific regions *within* string variables or other fields so you can later refer to these fields with *pathnames* and access them as variables for any purpose (Chapter 5, Section 3).
- Supports *pointer* extraction and resolution on variables, arrays, strings, fields, procedures and functions, similar to ***C*** or **PASCAL** pointer capabilities but with better *dynamic* support (Chapter 5, Section 4).
- Extended, integrated library of character string and bit-string operations, including pattern matching and search, re-ordering and rotation, format conversion, character translation, set searching, enumeration, union, intersection and exclusion. Large strings and string arrays supported and no *garbage collection* penalties (Chapter 4, Section 1).
- True *multi-level* error trapping that lets you trap errors at any level or pass errors on to higher level as needed (Chapter 6, Section 4).
- Supports *shared/exclusive* open files and file region locking in network and multi-user environments (Chapter 7, Section 2)

If you are reading this section for instructions on how to **RUN** a MegaBasic program and have no interest in the details of actual programming, skip this section and move on to Section 2 in this chapter.

Section 1: MegaBasic Components and Installation

The MegaBasic software system comes with one user's manual and a diskette containing all the software components. Some of the more important files are described below:

BASIC	MegaBasic development system for creating, testing, debugging and **RUN**ning programs. Several different floating point **BCD** precisions and **IEEE** binary real formats are available. Standard precision is 14-digit **BCD** or 16-digit **IEEE** binary.
RUN	MegaBasic **RUN** time system for optimized execution and reduced memory requirements, but without program development support. Several different floating point precisions are available.
CRUNCH	Program compaction utility, with an option for code protection of finished programs using a ciphering or scrambling technique (Appendix 2, Section 2).
CONFIG	Utility for altering various MegaBasic internal parameters (see Appendix C, Section 3).
LIBRARY	MegaBasic program containing many useful general purpose subroutines for use in your programs.
PCBASLIB	MegaBasic program containing special purpose subroutines for the **IBM-PC** environment (**MS-DOS** versions only).
README	Documentation file containing additional information not yet available in the MegaBasic manual. This file may or may not appear and its contents will vary depending on when the MegaBasic system was purchased.
CONTENTS	Documentation file containing a complete list and description of everything on the diskette. See this file for specific information about any files on the diskette that are not described above.

There may be some slight variation in the precise disk contents and spellings of the file names; the above list is intended to be a rough guide rather than an exact table of contents. This is because MegaBasic is supported on a wide variety of machines and operating systems and the disk contents are much more likely to change over short periods of time, as compared with the printed documentation.

Installing MegaBasic on Your Computer

Before you install MegaBasic on your computer, be sure that the capabilities of the machine satisfy the minimum requirements listed below:

- 8088, 8086, 80186, 80286, 80386 or 80486 Microprocessor running **IBM PC-DOS**, **MS-DOS**, **CP/M-86, TURBODOS**, Concurrent Dos, Xenix 386 or other operating system supported by MegaBasic. **MS-DOS** versions require **MS-DOS** revision 3.0 or later.
- At least 128k bytes of free memory before loading MegaBasic.
- CRT Console Screen and keyboard (a hard-copy console is not recommended). VGA systems are recommended.
- One or more disk drives (two or more recommended, one or more hard disks are highly desirable).

Additional equipment to further enhance MegaBasic capabilities includes larger disk drives, up to 16 Megabyte of extended memory (accessible to Extended MegaBasic), a high-speed printer and a letter quality printer.

If your computer has only floppy disk drives and no hard disk, MegaBasic installation consists of the making working copies of the MegaBasic release disk(s) and then using MegaBasic from those copies. However, hard disks make your life much easier and most microcomputers now come equipped with hard disks providing anywhere from 20 to 1000 *megabytes* of storage.

The specific installation steps to follow will vary with the operating system you will be using. The vast majority of MegaBasic users will, however, **RUN** MegaBasic under **MS-DOS.** Some releases of MegaBasic include a file named **INSTALL** that performs all necessary **MS-DOS** MegaBasic installation tasks. These tasks can also be done manually in releases without the **INSTALL** program, as follows:

- Create a new directory under the *root* directory named PGM, and copy all files from your MegaBasic release diskette(s) into it.
- Place the PGM directory into the default search path, by modifying the *PATH command* that should be in the *autoexec.bat* file in your root directory. This step lets you use MegaBasic and your MegaBasic programs from whatever directory you happen to be in when you **RUN** or work on them.
- Modify, as needed, the *FILES command* in your *config.sys* file to increase the open-file capacity of the system up to at least 40.

You could perform all these tasks yourself, but the **INSTALL** program handles all the details, which could take a while for a newcomer to **MS-DOS**. All you have to do to RUN **INSTALL** is insert the diskette into a drive and type:

```
INSTALL
```

After **INSTALL** completes, remove the diskette(s) and re-boot your computer, which applies any new configuration to the running system. From then on, you can load MegaBasic and access program files from the PGM directory without having to specify any directory *path names* (regardless of the current default directory).

About This Manual

We have confined this manual to only one purpose: a complete and accurate description of all MegaBasic facilities, in which you can quickly find the material you need and get on with your work. To this end, this manual has been organized into useful logical sections as shown in the table of contents. To answer your questions on specific subjects, an extensive index, with over 3800 entries, will direct you from the phrase you think of to the pages you need to read. The subject matter is covered in depth, with all its nuances, so that questions are answered rather than raised. We think you will appreciate this approach, over an *alphabetic* organization that breaks up and scatters related information arbitrarily about the manual.

This manual is a reference guide to the facilities provided by MegaBasic for creating, modifying, debugging and running programs written in the MegaBasic programming language. It is not intended to be a tutorial manual and its emphasis is on your daily needs over the long run, rather than your short term needs when you begin using MegaBasic for the first time. Because of this, people unfamiliar with general **BASIC** programming may wish to select a beginning **BASIC** programming guide to supplement this manual for further clarification of **BASIC** structures and usage. A working knowledge of your computer system and its operating system is assumed.

No book or set of documentation can do the learning for you. Computer software, more than most subjects, is difficult at best to get across in print because it is a *dynamic* activity. You would not attempt to learn to play the piano or ride a bicycle out of a book and you should not expect to use any complex software tool by the manual alone. Try everything, make *lots of mistakes*, play with each new feature that you are learning. After all, you *do* have a powerful computer system sitting in front of you which you can use to experiment with each of the facilities in MegaBasic.

Section 2: Running Programs from the Operating System

MegaBasic is an *executable file* which you **RUN** by typing its file name from the console as a direct command to the operating system. It must reside on one of the system disk drives installed on your computer system in order to be executed. The specific command to invoke MegaBasic merely consists of the file name containing MegaBasic followed by the name of the file containing the **BASIC** program you wish to execute. File names are not fixed entities and particular applications may have file names assigned which differ from those stated in this manual. Assuming MegaBasic is contained in a file named **BASIC** and your program is named **MYPROG,** the command to execute your program from the operating system is as follows:

```
BASIC MYPROG
```

This command causes the operating system to load MegaBasic which in turn loads your program file and then begins its execution. At that point your program takes over the computer and proceeds with whatever it is programmed to do. The **MYPROG** program file contents must have previously been created by MegaBasic.

The MegaBasic Development Version

The standard distribution disk of MegaBasic includes several different configurations of MegaBasic which can be divided into two fundamentally separate forms. The first form is your primary development version that supports all phases of program development such as program entry, saving to files, debugging, testing, etc. All of your time spent developing software under MegaBasic is spent under this version, usually named **BASIC** on the disk. The development environment provided by **BASIC** is entered using the same command shown above but without the additional program file name, as follows:

```
BASIC
```

This operating system command puts you into the MegaBasic command level from which you can enter program lines and MegaBasic commands. To leave **BASIC** and get back to the operating system, type the MegaBasic command: **BYE**

The Runtime Version of MegaBasic

The second form of MegaBasic is a subset of the first which can **RUN** programs, but *does not* support any *program development* facilities. This second form, usually named **RUN,** is designed for the production environment in which only finished and debugged programs are executed. **RUN** is about 30% smaller than the development version (saving about 24k bytes), and up to 50% faster. **RUN** provides even greater memory savings because it compacts all programs (but not their data) it executes down to 50-80% of their original size whenever they are loaded into memory. This compaction process consumes less then a tenth of a second and is totally invisible to the user and the program.

RUN is executed exactly as described above except that **RUN** must be typed in the command instead of **BASIC**. Since it contains no development facilities and cannot even list program source code, **RUN** is ideal for so-called *turn-key* systems which are sold to end-users or distributed throughout an organization in executable form only. Programs are therefore secure against unauthorized alteration and source code access. As a *licensed purchaser* of MegaBasic, you can distribute **RUN** with your programs to third parties without any royalty or other licensing fees.

As an additional security measure, a separate utility is provided to scramble the contents of a program file. Such files may be executed using **RUN**, but the development version cannot even load them for listing, execution, or any other purpose. The scrambling process is irreversible, making the program file useless for anything except its intended use. Needless to say, scrambling your only copy of a particular program is not recommended. The utility that performs this process, called **CRUNCH,** is described in Appendix C, Section 2.

The **PGMLINK** utility provided with the **RUN** system can produce a stand-alone program that combines your program with a copy of **RUN** into one file so that it becomes functionally indistinguishable from other utility programs or compiled software. This utility is described in Appendix C, Section 1.

The MegaBasic Compiler

A compiler for creating execute-only versions of finished MegaBasic programs is available as a separate option. It analyses your program for errors in syntax, argument list formation and data type consistency and, if no errors are found, produces a program that executes from 200% to 800% faster than the original. If any errors are found, the compiler describes them with sufficient detail for you to correct them and re-compile the program. Except for the much faster execution speed, compiled MegaBasic programs operate identically to their interpreted counterparts with little or no increase in memory requirements. The compiler operates on programs that have already been prepared under the MegaBasic development system, and must be used in conjunction with that system. For further information, see the documentation supplied with the MegaBasic compiler.

Section 3: Program Development Overview

To use the MegaBasic program development environment, type the **BASIC** command described earlier, but omit the program file name. Without a program name, you immediately **ENTER** into the command mode of MegaBasic which under your direction provides facilities to create and test programs. Only **BASIC** (the *development* version) provides this command mode, while **RUN** (the *runtime* production version) does not.

The command mode provides a selection of over 20 commands, which you choose and **ENTER** from the keyboard. Each command specifies a single task which MegaBasic carries out immediately after accepting the command. The command set can be divided into four logical groups:

- **Program Entry & Retrieval**

  entering programs from the keyboard or from files, listing your programs on the console or other devices, saving your programs to files.

- **Editing & Alteration**

  Sequential line editing, global search and replace, identifier renaming, line renumbering, line range deletion, rearranging program sections, merging program modules from files into your current program.

- **Execution Control & Debugging**

  Running and testing, debugging by breakpoint and single-step debugging, interrupting and continuing execution, interactive examining and setting of program data structures.

- **Information and Control**

  Displaying program statistics, listing file directories, exiting back to the operating system command level, switching between multiple programs in memory, displaying execution state.

After entering the MegaBasic command mode, the first thing you do is either key in a program from the console or load an existing program from a file. To type new program lines from the console, enter a line number (an integer from 0 to 65535), followed by a sequence of program statements separated by semi-colons and terminated with a carriage return. Lines may be up to 255 characters long. The line number tells MegaBasic where to insert the new line into the current program. Therefore new lines may be entered in any order, providing a simple way to insert changes at a later time. See Chapter 1, Section 4 for further details on MegaBasic program format.

Any line typed with a valid line number is *always* inserted into the current program; if there is no current program the line becomes the first program line. If the line number duplicates a previously existing line number, that line is replaced with the new line. All lines entered *without* line numbers are assumed to be *commands or direct statements* that MegaBasic attempts to execute immediately regardless of their actual contents. MegaBasic will inform you of lines which contain improper statements or commands when they are typed for immediate action.

After entering or loading a program and making any desired changes, you can then run the resulting program under interactive control of execution to check its correctness. If errors

are found, you can alter the program to correct the errors, and then repeat the process until you are satisfied with program operation. At any stage of the development phase, the current program may be saved on a disk file to safeguard your work from system failures or your own blunders (e.g., power failures, mistaken revisions), or so that you may continue work at a later time. On completion of your working program, save the final version on a file to be executed as described in Section 2 of this chapter.

MegaBasic always maintains a file name in connection with your program. This file name is the one used to load the program from the disk or the one used to save the current program onto the disk. A program entered from scratch at the keyboard is assigned the name *unnamed.pgm*.

MegaBasic keeps track of this file name for two reasons. First, you can save your development work out to the file without having to remember its name yourself or to type it correctly each time, which saves time and eliminates potentially destructive mistakes. Secondly, MegaBasic lets you have as many as 64 programs in memory simultaneously and the file name associated with each provides a name through which they may be accessed at random. Each program source has its own workspace (in memory) in which development activities may take place. This capability is extremely powerful for large scale program development and execution purposes, but its detailed description is beyond the scope of this section and will be covered later on in Chapter 10.

Section 4: Lines, Statements and Program Form

MegaBasic programs consist of a series of typed lines beginning with a line number and ending with a carriage return. Line numbers must be in the range 0 to 65535 and serve a dual purpose. First, since MegaBasic continually keeps the program lines arranged in ascending order, you can easily insert additional lines by typing them with appropriate line numbers. Secondly, some MegaBasic statements refer to program steps by line number, perhaps to repeatedly execute some group of statements or skip over undesired statements. The simple example program below illustrates some of the building blocks used to form programs:

```
100 REM *** This is a sample Program ***
110 INPUT "ENTER a number -- ",N; If N<=0 then Stop
120 Print N, N*N, Log(N), Tan(N), Sqrt(N), Atn(N)
130 Goto 110
```

Line 100 contains a remark which describes the program to a human reader and is ignored by MegaBasic when executed. Such remarks may appear anywhere in a program to document program operation. Line 110 contains two statements, separated from each other with a semicolon (;). The first statement causes the computer to display the request *ENTER a number*—and accept a number from the user when he/she is ready to type it in. The second statement on line 110 stops the program if the number entered is less-than-or-equal-to zero. Line 120 goes on to display various computations on the value entered, but only if the value is greater than zero. Line 130 causes the computer to go back to line 110 and ask for another number, which repeats the whole process until the number entered is not greater than zero.

Besides being numbered, the lines themselves may be up to 255 characters long and consist of one or more statements (i.e.. you cannot have a line with no statements on it). Statements are separated from one another in the line with semi-colons (;) and represent the fundamental building blocks of MegaBasic programs. Statements in general begin with a specific keyword followed by additional data parameters separated from one another with commas (,). For example the **PRINT** statement above begins with the keyword **PRINT** and it is followed by a list of things to be printed.

By themselves, statements perform simple and easily understood operations, but in combination they can express procedures of unlimited complexity. MegaBasic statements are grouped into six Chapters (Chapters 5 through 10), each beginning with a summary of the statements they contain, followed by detailed descriptions of each MegaBasic statement.

Program Line Continuation

Extra long lines, longer than 80 characters, will *wrap-around* to the next line on your console. Usually this will break the line at an arbitrary and undesired place. To break up your long lines anywhere you choose, type a line-feed (Ctrl-J or Ctrl-**ENTER**) and continue your line. Line-feeds are like carriage returns except that they do not terminate the line, thus permitting one numbered program *line* to be folded into several *physical* lines. Line-feeds are also useful as the last character of a program line (before the

carriage return) to insert empty blank lines for visually separating successive sections of your program. A line-feed may be typed anywhere a space is permitted. No line may be longer than 255 characters, regardless of line-feeds.

Program Line Numbers

In recent years there has been a move away from line numbers in **BASIC** programs. Although programs without line numbers *look* cleaner, in some ways they tend to be more difficult to develop and maintain. MegaBasic uses line numbers not because they *look great*, but because they provide real functionality in the following areas:

- Traditional **GOTO, GOSUB** and **DATA** statement references.
- Reporting locations of errors in program execution and syntax.
- Identifying program locations of error recovery routines.
- Discriminating between program lines to be inserted and commands to be acted upon right away.
- Identifying line locations for program editing, e.g., insertions, deletions, replacements and merging.
- Identifying line ranges for block operations, e.g., block search, display, text replacement, etc.
- Reporting program source locations in cross-reference listings and analyses.
- Facilitating program development on *minimal* terminals and over modem communications lines to remote terminals.
- Communicating program locations during conversations and exchanges between programmers and software support people, e.g., over the telephone.

Line numbers do have there shortcomings, however. They take up valuable screen space; they *do* look ugly; they make every line a potential target of a **GOTO, GOSUB** or error trap. But, as the list above illustrates, line numbers provide functional capabilities that cannot easily be duplicated by line labels or other more *modern* or *high-tech* solutions to the same problems.

Section 5: Names and Identifiers

A key feature in MegaBasic is the way it lets you to assign meaningful names to any program line, variable, function or subroutine. For example, the name **CUBE_ROOT** is certainly more descriptive than FNR3 for a user-defined function that computes cube roots. Names must conform to certain rules in order to be properly recognized. The syntax of user-assigned names in MegaBasic is simple, reasonable and easy to remember:

- Names must begin with a letter (A-Z).
- Characters after the first must be letters (A-Z), digits (0-9), or underscores (_).
- The last character of a name may be a dollar sign (**$**), a percent sign (%) or an exclamation mark (!) to force the data type of the name to string, integer or floating point, respectively. Other methods exist to declare the data type of a name without such characters.
- Names may be from 1 to 250 characters in length and all characters participate in the spelling and must be present in all references.
- Upper and lower case letters in names are treated identically.
- MegaBasic reserved words (e.g., **FOR**, **NEXT, READ**, etc.) cannot be used for user-assigned names. See Appendix D-2 for a complete list.

Examples of valid names are *TOTAL!*, *X3*, *THIS_IS_A_NAME*, and *STRING***$**. Examples of illegal names are *3X LABEL#*, **$***VAR*, *X***$***STR*, and *THIS&THAT*. Underscores are useful for breaking up longer names since spaces are not permitted. All characters in a name are significant in recognizing the name, i.e., two names are different unless they match exactly**.** Upper and lower case letters are treated identically so that you can type names with or without the **SHIFT** key.

Line-labels are names which may optionally be typed at the beginning of any program line (after the line number). Such lines may be referred to either by line number or by name. For example, the following one line program prints all the integers from zero to one hundred:

```
10 AGAIN: Print C; C = C+1; If C<101 Then AGAIN
```

Notice the colon (:) after the AGAIN line-label. A colon must always follow each line-label definition immediately without intervening spaces. Line label references are never followed with a colon. The colon is required to clearly distinguish line-labels from other named objects used in the program.

This example uses a variable named C which is displayed and incremented by the program. Regardless of how you type in a program, when it is **LISTed** user-assigned names always appear capitalized and MegaBasic reserved words appear in lower-case so that you can see which are which. This is important because reserved words cannot be employed as user-assigned names. Hence when you see one of your assigned names spelled with any lower case letters, you will know that it is a reserved word, an error that must be rectified by editing the program. This kind of editing is best performed using the **CHANGE** command (Chapter 2, Section 3).

Variables and functions with names ending in a dollar sign (**$**) are automatically string variables and string functions. A percent sign (%) ending names of variables and

functions gives them an integer data type and an exclamation mark (!) forces a real floating point type. You can assign data types to various letters so that variables and functions with names beginning with those letters will automatically be defined with the data type specified. This subject is covered further in Chapter 3, Section 1.

The **NAMES** command (Chapter 2, Section 3) displays the user-assigned names in your program. It is sometimes useful for finding occurrences of names which have been misspelled or mistyped during the course of editing your program. Since the **NAMES** display is alphabetically ordered, names which are similar tend to be together in the list and it is generally a simple matter to visually scan the list to find similar but different *spellings.*

If you do not correct such misspellings, each different spelling will refer to a different program variable, function or procedure, and your program will not operate correctly. Another way to detect such errors is by displaying a cross-reference listing of your program, using the **XREF** command (Chapter 2, Section 5). This command finds all references to each user-assigned name throughout your program. Since virtually all names will be used in more than one place, any names that are only referred to once are likely misspellings of other names. **XREF** should be used for this purpose after you make any major additions or alterations to your program, so that you can correct any misspellings before you even begin testing your program again.

Section 6: The MegaBasic Line Editor

Whenever you **ENTER** data or program lines from the keyboard you are actually using the MegaBasic line editor. This line editor lets you **ENTER** lines of text, and provides editing services ranging from simple typing corrections to text insertion, searching, block deletion and rearrangement. It provides a visually complete presentation of the line you are modifying at every key stroke, while supporting virtually all video screens (i.e. IBM-PC screens and generic terminals) without any configuration. This makes it suitable for use over modem communication lines and a wide variety of hardware configurations.

All editing functions are invoked by typing special control or function keys. Not all keys perform editing functions and if accidentally struck will be rejected by the computer with a warning *beep*. For the purpose of notation *Ctrl- ?* will denote a control character where ? is some key.

If you don't make any mistakes while typing an input entry, then all you have to do when your input line is finished, is type the **ENTER** (or **RETURN**) key. You can easily correct simple typing errors by backing up over the error with the **BACKSP** key, type the correct characters, then continue the input entry. In the pages that follow, we will explain how to use other line editor control keys to insert text, delete and rearrange text blocks, move the cursor, and search for characters.

Inserting Text

Some editors provide two different ways to input characters: insert mode and replacement (or overwrite) mode. This forces you to remember at all times what mode you are in. To make things easier, the MegaBasic editor is *always* in insert mode. This means that whenever you type characters while inputting or editing a line, the characters you type are always inserted into the line at the cursor location. To replace characters in your line with a new sequence of characters, you have to delete the old sequence then type the new sequence.

The cursor is the special screen symbol that indicates the location where the next character will appear. Normally, this will be at the end of the line you are typing. However, you can move the cursor to any point within the line you are editing, so that subsequent characters you type will be inserted into the line instead of appended to the end of it. Cursor repositioning is summarized on the next page.

When the cursor reaches the right margin of the screen and you continue to type more characters, the cursor will *wrap around* to the next screen line below it and continue on. This will generally break up your input entry in an arbitrary place. You can insert you own line break anywhere in the line by typing a line-feed (down arrow or Ctrl-J). This breaks the line, moving all text past the cursor down one line, and positions the cursor at the beginning of the next screen row and enters a line-feed code (an ASCII 10) into the input line.

Most input entries will be less than 80 characters and will generally fit completely on one screen line. However, MegaBasic lets you type a line of up to 255 characters. Once this limit has been reached, MegaBasic prevents you from entering any more characters and *beeps* at you each time you try to insert a character. At that point you either have to delete characters from the line to make room for more input, or enter the line the way it is.

Cursor Positioning

MegaBasic provides a variety of ways to move the cursor to a different location within the current input line. Changing the cursor position does not alter the line in any way, nor does the position affect the input entry when you type the **ENTER** (or **RETURN**) key to terminate it. The only reason to move the cursor is so that a subsequent insertion or deletion can take occur at the right place.

Two controls let you move the cursor left and right by one character (the left and right arrow keys or Ctrl-L and Ctrl-A). Two other controls let you move the cursor left and right by one word (Ctrl-left and Ctrl-right arrow keys or Ctrl-W and Ctrl-Q). A *word* in this context is any sequence of letters and digits containing no other characters. Moving left or right a word always leaves the cursor on the first character of the word. By typing these keys repeatedly you can *walk* through the line to quickly locate the position where you want to make a change. Two other controls let you move to the beginning of the line (Home or Ctrl-F) or to the end of the line (End or Ctrl-G).

Another control, F2 (or Ctrl-S), lets you advance the cursor to the next occurrence of any single character. After you type it, you must then type the character you wish to find. If it exists, the cursor moves to that character in the line, if it does not exist, the cursor does not move and a warning *beep* sounds. If you type this control twice, it will search for the same character that it searched for the last time. When you search for a letter, you can type it in upper or lower case regardless of the case of the letter sought.

An important aspect of entering and editing program lines is making sure that all your parentheses and brackets are properly balanced. In complicated lines containing many levels of parentheses, it can be difficult to see where each parenthetical sequence begins and ends. Therefore, MegaBasic provides two keys to move the cursor between opening and closing parentheses. F9 (or Ctrl-O) backs up the cursor to the preceding parenthesis, bracket or brace. If the cursor is already on a closing parenthesis, bracket or brace, it backs up to the opening parenthesis that matches it. F10 (or Ctrl-P) is the reverse of F9, advancing the cursor to the next parenthesis, bracket or brace in the line. If the cursor is already on an opening parenthesis, bracket or brace, it advances to the closing parenthesis matching it. If no matching parenthesis exists in the line, the cursor does not move and a warning *beep* sounds.

In order to promote the widest possible console compatibility, MegaBasic relies on only the minimum possible set of console controls to position the cursor. Only one operation requires any configuration: backing up the cursor to the previous line. This is controlled by the Console Mode byte, which you can configure using the CONFIG utility program, described in Appendix C, Section 3. If you have trouble with the line editor maintaining the proper cursor position or observe any erratic behavior, consider trying a different configuration.

Deleting Text

Deletion is always relative to the cursor position. BACKSP deletes the character to the left of the cursor; DEL deletes the character at the cursor location. F6 (or Ctrl-V) deletes all the characters from the cursor to the *next word.* F4 (or *Ctrl-X) followed by* a *character* deletes from the cursor up to that character, or *beeps* if the character is not found in the line. Typing this F4 twice deletes up to the next occurrence of the *previous* search character. Ctrl-HOME deletes all characters to the left of the cursor; Ctrl-END deletes all characters from the cursor position to the end of the line.

Text Recovery and Rearrangement

MegaBasic provides a limited mechanism to recover text *that you* have deleted without forcing you to type it back into the line. Every *time you* delete one or more characters from the line, MegaBasic remembers those characters. If you make several deletions from *the* same place in *the* line, MegaBasic remembers the entire sequence as one deletion. You can recover this sequence of deleted characters by typing *the Ctrl-U* key. The deleted characters are inserted into the line at the cursor location in effect when you type Ctrl-U, leaving the cursor positioned after the insertion.

You can recover only the most recent sequence of contiguous deleted characters. For example, if you delete 10 characters from the beginning of the line, and then move to the end of the line and delete 5 characters, typing the Ctrl-U key recovers only the 5 characters; the 10 characters deleted from the beginning are lost. However, if you delete the preceding 4 characters, then you delete the next 3 characters, all 7 characters are remembered and may be recalled by typing Ctrl-U.

In addition to simple recovery from accidental deletion, you can also use this operation to rearrange text within the line, or to move text from one line to another. First, delete the text sequence you wish to move. Second, move the cursor to the location in the line where you want to move the character sequence (being careful not to perform *any other deletions* along the way). Third, type the Ctrl-U to insert the deleted characters back into the line at the cursor location. If you are editing a MegaBasic program, you can use this capability to delete a portion of one program line and insert it back into another program line (as long as there are *no other* intervening deletions) . You can also type Ctrl-U repeatedly to insert the same string into the line as many times as the line capacity permits (255 characters maximum).

Accessing The Previous Input Line

To simplify entry of repetitive or similar input lines, you can access the previous input entry by typing F5 (or Ctrl-R). This abandons any input you have already typed, displays the previous line (called the *old line)* and positions the cursor in front of it. This saves time when the computer requests successive entries that are identical or differ only slightly. Furthermore, if you are editing the old line and make some irrecoverable editing errors, you can type F5 (or Ctrl-R) to restore its original form so that you can start over with the least amount of effort.

If the *very first key* typed to an input or command line entry is an editing control key (rather than an *ordinary* input character), MegaBasic automatically restores the previous input entry as the current entry before acting on the control typed. This implicit restore operation makes the previous input entry easier to access, but you can only get it on the first key typed.

Restoring previous input is frequently useful when you are entering commands and program lines in the MegaBasic command level. You will find yourself typing successive commands which differ from the previous command (the old line) by only one or two characters, or to correct a mistake in a command just entered. Similarly, instances of nearly identical sequential program lines are common. Your program may already contain lines which nearly match a new line about to be entered into the program, and by editing the old one and changing its line number, you can construct the new line with minimal effort.

Accessing Any Prior Input Line

In addition to just the prior input line, MegaBasic also remembers all the most recent lines of text entered through the console keyboard so that you can retrieve them

whenever you are entering a command or entering keyboard input. This is particularly useful when you find yourself entering several different complicated commands or inputs repeatedly, since you can avoid having to retype them each subsequent time. MegaBasic only remembers one instance of each line entered and keeps them in a most-recently-used order for convenient access. Lines that differ only in upper / lower case and number of spaces are treated as the same line and only the most recent *rendition is* remembered. Null lines (i.e., those without any characters) are never retained.

You access previously entered lines by typing one of several control keys at any time while you are entering a text line into MegaBasic (or into a MegaBasic program). PgDn and PgUp keys move forward and backward through the line list; F5 returns to the original line and Ctrl-D deletes the current line from the list. Once a line is accessed, you can immediately begin editing it without any further keystrokes. At any time you can discard your current line and start over on a different line by simply accessing another line and continuing.

When accessing previous entries with PgUp and PgDn keys, the characters *to the left* of the cursor are used as a matching criteria, selecting only the entries that begin with those same characters. As each line is accessed, the cursor is left in the same position so that you can step through different lines beginning with that sequence. A *warning beep* indicates no entry begins with such a sequence. If the cursor is at the front of the line (i.e., no characters to match), PgUp and PgDn keys step through every line.

The number of lines retained depends on how many lines fit into the previous line buffer. This buffer defaults to 512 bytes, but you can change its size to any value from 0 to 4096 bytes by setting **PARAM**(24) to the desired size at any time Setting the buffer size to zero disables the previous line list capability altogether (except for the standard *old line* buffer). Setting **PARAM**(24) always clears the buffer of all lines, except for the most recently entered line. Defining a larger or smaller buffer size causes the total available memory space to decrease or increase accordingly.

If there is not enough room in the previous line list buffer for the next line being added to it, MegaBasic makes room for it by deleting the *oldest* lines in the buffer until sufficient room becomes available. If the line length exceeds the entire buffer capacity, the line will not be added to the list. Therefore to use this capability effectively, your buffer size (as defined by **PARAM**(24) should be at least as large as the longest line you will ever want to retain.

When you are modifying your program under the **EDIT** or **ENTER** command modes, the entered source lines can quickly fill up the previous line buffer and displace some or all of the prior command lines that you have typed. Therefore MegaBasic only remembers the single most recent program source line that is entered while in these modes. If you want to be able to access other such lines in later editing or input, you can always force the current line into the buffer by typing Ctrl-B just before typing **RETURN** to enter the line.

The **EDIT$** function always returns the most recent line so far entered. Setting **EDIT$** (e.g., **EDIT$** = *string)*, adds a new most-recent line to the line list. Setting **EDIT$** several times in succession adds several lines to the list, which can be useful for pre-loading the buffer in preparation for a subsequent input entry.

Editing Control Characters

The preceding discussion provides a complete explanation of the MegaBasic line editor, it capabilities and the editing process in general. The table below summarizes all of the editing control keys provided by MegaBasic.

For convenience, alternate keys are provided for most editing operations. In particular, the editing and cursor controls provided by the **IBM-PC** and **PC BASIC** are represented along with a generic control-character set that will work with any console terminal. Control characters are typed by pressing a specific character while holding down the key labelled **CTRL** on the left of the keyboard (the **SHIFT** key may be up or down). The **IBM-PC** set consists of function keys F1 through F10, the **HOME, END, TAB** keys and the cursor direction arrows (denoted Left, Right, Up and Down). These keys are supported for the editing functions below only for **IBM-PC** compatible keyboards. Other keyboards may appear to have these keys but the actual codes they generate may not be the same. If the indicated action for a editing key cannot be completed by MegaBasic for any reason, a warning beep is sounded.

The controls described below are line-oriented and their actions are confined solely to the current line being input or edited. When you are editing program source code, each line you are editing is under control of this line editor as a stand-alone line. There are currently no controls that provide a *full-screen* editing facility within MegaBasic (e.g., you cannot move the cursor freely between separate program lines). A line may be broken up into more than one *screen* line with line-feeds or by entering characters past the end of the screen to cause a *wrap-around* to the next screen row. Although such a line appears to be multiple lines, you should treat it as the single line that it is.

Character Operation	
Right **F1** **Ctrl-A**	Moves the cursor one column to the right or to the next line if a line-feed is encountered. This does not modify the current line. A warning beep will sound if you are at the end of the line when you type this control.
Left **Ctrl-L**	Backs up the cursor one column to the left. This can be repeated to backspace all the way back to the beginning of the line. It also backs up through line-feeds embedded in the line.
Backsp **Ctrl-H** **Rubout**.	Deletes the character to the left of the cursor. All remaining characters in the line that follow are shifted left one column to close the gap. Line-feeds and TABS can be deleted just like any other character
Del **Ctrl-Z**	Deletes the character from the line at the cursor position and shifts all characters that follow it over one column to close the gap. The cursor does not move. Line-feeds and TABS can be deleted just like any other character.

Word Operations	
Ctrl-Right **F8** **Ctrl-Q** **Ctrl-E**	Advances the cursor forward to the beginning of the next word in the current line, where *word is* defined as any contiguous sequence of letters and /or digits. This key is useful for quickly skip ping through the line to some point of interest.
Ctrl-Left **F7** **Ctrl-W**	Backs up in the line to the beginning of the previous word, where a word is defined as a contiguous sequence of letters and /or dig its. If the cursor is in the middle of a word, it backs up to the be ginning of that word.
F6 **Ctrl-V**	Deletes all characters up to, but not including, the first character of the next word, where *word is* defined as any contiguous sequence of letters and /or digits. The text to the right of the deletion moves over to the left to close the gap.

Searching Operations	
F2 **Ctrl-S**	Advances the cursor up to the character that you type immediate ly after this key. Upper and lower case letters are equivalent when searching. If the specified character is not in the remainder of the line a warning *beep is* sounded and the cursor does not move. This is a two stroke sequence and typing F2 twice will repeat the previous F2 search sequence.
F4 **Ctrl-X**	Deletes all characters from the cursor position up to, but not in cluding, a specified character. Like F2 above, F4 is a two-stroke sequence and typing F4 twice will repeat the previous F4 deletion sequence.
F9 **Ctrl-O**	Backs up the cursor to the preceding parenthesis, bracket or brace. If the cursor is already on a closing parenthesis, bracket or brace, it backs up to the opening parenthesis that matches it.
F10 **Ctrl-P**	Advances the cursor to the next parenthesis, bracket or brace in the line. If the cursor is already on an opening parenthesis, bracket or brace, it advances to the closing parenthesis matching it.

Line Operations	
End **F3** **Ctrl-G**	Advances the cursor to the end of the current line. Further input after this control will append to the end of the line.
Home **Ctrl-F**	Repositions the cursor to the beginning of the line, regardless of its current location.
Ctrl-End **Ctrl-N**	Deletes all characters from the cursor position all the way to the end of the line.
Ctrl-Home	Deletes all characters *to the left* of the cursor all the way back to the beginning *of* the line.

Edit Control	
ENTER **RETURN**	Terminates the edit, moves the cursor to the end of the input line, adds the line to the previous line list and returns the entire line to process requesting the input.
Ctrl-C **Esc**	Erases the line from the screen, abandons the line edit and terminates whatever process is currently underway. This key does nothing during program execution if Ctrl-C is disabled.
Up **Ctrl-K**	When you are in the MegaBasic program **EDIT** mode, Ctrl-K will abandon the current line you are editing and begin editing the line that immediately precedes it in the program. When you are in the **ENTER** mode (automatic line numbers), Ctrl-K will abandon the current line being entered and go back to the previous line and let you edit it.
Ctrl-U	This is an *undelete* key. It inserts the last contiguous sequence of deleted characters back into the line at the cursor position. It is useful recover deleted characters or to move or copy char acter sequences from one place to another, even between *separate* entries.

Line Formatting	
Down **Linefeed** **Ctrl-J**	Forces a line break during an input entry without terminating it. In MegaBasic, an edited input entry can be up to 255 characters long. Therefore this key lets you break long input entry into several physical lines by entering line-feeds into the input line.
TAB **Ctrl-l**	Advances the cursor and any text that follows it over to the next column position divisible by 8 (i.e. 8,16, 24,...). The key enters a single character into the input string (an ASCII 9 code), rather than a series of spaces. Tabs are permitted in program lines anywhere that spaces are permitted or as separators between numeric inputs. They are useful in program lines for indentation and other significant *whitespace* without eating up the line capacity (255 characters maximum) the way spaces do.

Previous Line Access	
PgUp **Ctrl-T**	Replaces the current line with the most recent entry matching the characters to *the left of the cursor. Typing this* key repeatedly accesses earlier *and* earlier lines. Once the *oldest* line *has* been accessed, typing this key cycles back to the *newest* line again. If the cursor is at the front of *the* line, every line is accessed.
PgDm **Ctrp-Y**	Once you have sequenced through one or more lines using PgUp, this key lets you go back the other way (i.e. to the line *more* recently entered than *the* one you have). Typing this key repeatedly accesses later and later lines. Once the *newest* line has been accessed, typing *this* key cycles back to the *oldest* line again.
Ctrl-D	Deletes *the* currently selected line from *the* previous line list and accesses the next most recent line in the list. If the line list be comes empty a *null* line is presented for editing. Ctrl-D does nothing but *beep* until a prior line has been selected with one of the previous line access keys.
F5 **Ctrl-R**	Restores the original most-recent line as the current line being edited no matter where you are in the previous line list. The cursor is repositioned to the beginning of the line, allowing you to resume editing.
Ctrl-B	Adds the current line in its present form to the previous line list, making it the most-recently entered line. If *the* line was already in the line list, Ctrl-B merely moves it to the front of the list. This is the only way to add a line *to* the list without terminating the input entry and is useful for saving the current line at some stage *that* might be useful to recall at a later time.

Chapter 2 MegaBasic Commands

In a sense, MegaBasic supports two languages: the underlying programming language and the MegaBasic *command* language. The command language lets you control what MegaBasic does in the *command mode*, while the programming language controls what MegaBasic does in *execution mode*. This section describes all the MegaBasic commands and it is organized into the following five subsections:

Introduction	Explains formation and syntax of commands, their arguments and their use within MegaBasic workspaces.
Program Entry and Retrieval	Entering programs from the keyboard or from files, listing your programs on the console or other devices, saving your programs to files.
Editing and Alteration	Sequential line editing, global search and replace, renaming identifiers, renumbering lines, deleting line ranges, rearranging program sections, merging program modules from other files or workspaces into your current program.
Execution Control and Debugging	Running, testing and debugging programs. Execution breakpoints and be set and cleared. Single-step debugging lets you interrupt and `continue` execution and interactively examine and modify program data structures.
Information and Control	Displaying program statistics, listing file directories, cross reference reports, exiting back to the operating system command level switching between multiple programs in memory, displaying execution state

Chapter 2 gives information about the MegaBasic commands in general and the ideas common to several or all of them. This includes such topics as the multiple workspace environment, the notational conventions used to describe MegaBasic statement and command syntax, device numbers, search strings, etc.

BASIC *[<program> [<command tail>]]*

BYE

CHANGE *[<line range>],<search string>,<replacement>*

CLEAR *[{ **DATA FREE** }]*

CONT

COPY *<starting line> [,<stepsize> [,<line range>]]*

DEL *<line range>*

D U P L *<starting line> [, <line range>]*

EDIT *[<starting line>] [, <search string>]*

ENTER *[<starting line> [,<stepsize>]]*

LIST *[#<device >,] [<line range>] [, <search string>]*

LOAD *<program file name list>*

MERGE *<program file name> [<source/dest specs>]*

MOVE *<starting line> [,<line range>]*

NAME *[#<device>] [<selector list>]*

NAME *<old label>, <new label>*

REN *[<starting line> [, <stepsize> [,<line range>]]*

RUN *[<line number or command tail>]*

SAVE *[<program file name>]*

SHOW *[#<device>][{ **ACCESS OPEN SIZE**}]*

STAT *[#<device>]*

TRACE END

TRACE RET

TRACE *[#<device>,][<line>]*

TRACE *[#<device>,]* **IF** *<logical exprn>*

TRACE: *<executable line of statements>*

USE *[<workspace name>]*

XREF *[#<device>]G<line range>][,<selectors>][**by** <mode>]*

You can abbreviate several of the above command keywords to a specific two or three character sequence for convenience. These abbreviations are as follows: **CHANGE** as **CH, EDIT** as **ED**, **ENTER** as **ENT, LIST** as **Ll** and **TRACE** as **TR**.

Section 1: Introduction To MegaBasic Commands

The command mode provides a selection of over two dozen commands, which perform such things as loading and saving program files, modifying programs, displaying information about the program state, running programs, etc. Each command specifies a single task which MegaBasic performs after you type in the command. You can perform any complex task, such as developing and debugging a MegaBasic program, by typing individual commands, one by one, until there is nothing left to do.

Before describing the various MegaBasic commands, we will first explain the concepts involved in forming commands and how to use them within the MegaBasic workspace environment. Some of the things discussed in this introduction include specifying program line ranges, output device channels, string search patterns, understanding command and statement syntax notation, and program file names.

The Workspace Environment

MegaBasic permits up to 64 programs to reside in memory simultaneously. This unique feature exists to support large-scale programs composed of a collection of independently developed libraries which have controlled access to the subroutines and data defined within the others. Chapter 10 describes all aspects of designing, implementing and using program modules. During program development however, it is important to understand the multi-program environment because it arises in a number of the commands presented in this Chapter (**LOAD**, **SAVE**, **USE**, **STAT**, **SHOW**, and **TRACE)**. Understanding how you can work on, or just refer to, more than one program at the same time can save you considerable time.

When you are working on your program, the kinds of activities that you do includes things like entering program lines, editing program lines, testing and debugging, loading and saving programs, etc. MegaBasic provides an environment for such activities by maintaining your program as you change and mold it into whatever you desire. In order to have a way of talking about this environment, we shall refer to it as a *workspace*.

We would not have to draw a distinction between an *environment* and a *workspace* if you could only deal with one program at a time. However MegaBasic supports more than one workspace simultaneously within the entire environment that it provides. Supporting more than one workspace involves the following set of capabilities:

- Create the initial environment
- Create new workspaces by name
- Delete workspaces no longer needed
- Select a workspace by name for subsequent operations
- Show name and status information of each workspace

In order for you to create, delete or select a workspace you need a way to refer to a particular workspace. Since we already associate file names with every program, those same file names can also serve as workspace names. The act of loading a program from a file gives its workspace its name. MegaBasic assigns the default workspace name *UNNAMED.pgm to* the original workspace present when you begin a MegaBasic session. It keeps this name until you type in a program and save it onto a file of a different name, or load a program into it from a file.

A multiple workspace environment is much like sitting in a swivel-chair inside a circle of consoles, each one a window through which a separate program is accessible for development work. At any given instant, you are facing one screen and your actions are all directed toward the particular program source it contains. But at any time you can easily turn to any other screen to work independently on its contents. MegaBasic of course supports such activities using only one screen and is certainly more feasible than all that hardware.

The **USE** command selects existing workspaces and creates new, empty workspaces. The **LOAD** command can also create a new workspace in the process of loading a program from a file. The slow command lets you see what workspaces are currently present and the nature of their contents. The **CLEAR** command deletes the current workspace (and its contents) or deletes all workspaces. MegaBasic automatically deletes workspaces which do not contain any program lines.

Most MegaBasic commands will refer to workspace concepts to some degree and they are therefore important to understand. As with many computer tools, their actual use is much simpler than a description of their use, and you really need to try each of the various commands out on your computer to get *a feel* for show they can support your activities.

The user's view of a multiple workspace environment hides all details which are not immediately necessary for accomplishing the task at hand. Hence in the simple case, where you are only interested in a single workspace, MegaBasic does not burden you with extra options and other details relevant only in a multiple workspace context.

Syntactic Notation Used in This Manual

This manual uses special notation for specifying the syntax of MegaBasic commands, statements and functions. Each command (or statement) consists of a sequence of typed symbols. The symbols are of two varieties: those that you type exactly as specified, and those that describe a generic item that can vary from one instance to another. Special brackets are used to denote items that vary:

Notation In Syntax Descriptions	
<...>	Encloses a description of the item to be typed. For example, *<line number>* describes an item for which you substitute a specific program line number, and *<file name>* is an item for which you type the actual name of a file.
[...]	Encloses an item that is optional. For example *[#]* means that you may type an optional Ib-sign (#) in that part of the statement or command, and *[<sfring expression>]* means you type an optional string expression. The *[...]* brackets may contain several items, in which case you either type all of them or omit them all. You will also encounter bracketed items inside of outer brackets to indicate optional items within larger items which are themselves optional.
{...}	Encloses a list of items from which you choose one item. For example {STOP END ON} means you type one of the words **STOP, END** or **ON**. The {...} braces may include <...> items as well.

All letters, digits and punctuation are otherwise typed exactly as they appear in the command description. The actual bracket characters themselves (i.e., < >, *{ }* and *[ ]*) are not typed into a command or statement, as they are shown simply to help describe their syntax. However, there are a few places where brackets *[ ]* are specifically used in

statements (e.g., **IF** and **STRUCT** statements, vectors, etc.), but each case is specifically documented to avoid confusion. When any of the special brackets (and any *descriptions* they contain) are used to delineate syntax, they are shown in *italics*, otherwise they are shown in normal or **boldface** to indicate *literal usage*. The following examples should clarify how to type specific commands from their syntactic descriptions:

Syntax Description	Example Use
RUN ***[<starting line>]***	**RUN** **RUN 200**
EDIT ***[<starting line> [,<string>]]***	**EDIT** **EDIT 1000** **EDIT 150, find this**
ENTER ***[<starting line.> [,<step>]]***	**ENTER** **ENTER 100** **ENTER 300, 20**
CLEAR ***[DATA]***	**CLEAR** **CLEAR DATA**

Command and Statement Form

Most MegaBasic commands (and statements) that require multiple arguments have the form: *<keyword> <argument list>*, where the *keyword is* the name of the command (or statement), and the *argument list* may consist of strings, numbers, other keywords, etc., appropriate to that command. You must separate the listed arguments from one another with commas, but *no comma* separates the keyword from the argument list. You may insert any number of spaces or line-feeds within your entries to make commands (and statements) more readable. MegaBasic ignores all such characters not enclosed within quotes.

You must separate command keywords from their arguments with at least one space. This is because you can name any program entities with arbitrary names, and running command or statement keywords together with numbers or other identifiers creates new names that MegaBasic cannot recognize. For example, *LIST* 1,$ is a command which lists the entire program on the screen, but typing it as *LIST1,$* is not valid because the sequence *LIST1* is not a command word, so MegaBasic thinks it is a user-assigned word for some variable or procedure. Chapter 1, Section 5 describes the rules for naming programming constructs (Chapter 1, Section 5).

Specifying I/0 Devices

You can re-direct command output to somewhere other than the console screen by specifying a optional # *<device>* in the command . The **#** is necessary to indicate the presence of a device (or open file) number. For example, **#1** refers to the printer and you can type the command **LIST#1** to output your program on the printer. See Chapter 7, Section 1 for additional details about the devices.

Referring to Program Lines

Commands often refer to specific program lines or to line ranges. A program line location can be specified in several ways. An unsigned integer from 0 to 65535 refers to the program line with that number appearing in front of it (rather than its absolute line

sequence number). A dollar sign ($) refers to the *last* line of the program. A dot (.) refers to the *most recent* line displayed by MegaBasic or edited. You can specify a line using any of these three forms in line ranges or in any command where MegaBasic expects to see a line number.

In executable program statements that refer to lines (e.g., **GOTO** statements), you may optionally refer to the intended line by line-label, if that line contains a line-label. You cannot specify line-labels in MegaBasic commands for any purpose. Line-labels make the program much more readable and easier to develop and maintain. Chapter 1, Section 5 shows how to define and use line-labels and other named program entities.

Specifying Program Line Ranges

Many MegaBasic commands (e.g., **LIST DEL, REN, DUPL, MOVE, CHANGE,** etc.) can operate on a *subrange* of program lines, instead of acting upon the entire program. You can specify a line range in any of the following ways:

- Omitting the line range altogether implies the range of *all program lines* (where omitting it is allowed).
- A single line number to indicate a *one-line* range.
- Two line numbers separated by a dash (e.g., 100-999) specifies all lines with line numbers at or above the first line number and on or below the second line number.
- A single line number followed by a dash (e.g.,100-) to indicate all lines from that line to the end of the program.
- A single line number preceded by a dash to indicate all lines from the beginning of the program up to the line number specified (e.g., -450).
- Two dots .. to indicate the previously specified line range or the range last modified by MegaBasic. This line range is known as the *current line range*, a topic covered greater detail below.
- The name of a subroutine (i.e., **FUNC** or **PROC**) preceded by a dot (.) indicates the entire range of lines containing that subroutine. For example the command **LIST.SUBR** would list all the lines of a function or procedure named **SUBR**. A subroutine line range consists of all lines of the subroutine starting with its initial **DEF** statement, along with any *immediately preceding* **REMark** and **DEF** lines, up to and including the line containing its terminating **FUNC END** or **PROC END** statement. **REMark**s that follow subroutines are not included. Incorrectly formed subroutines (e.g., missing **FUNC** or **PROC ENDS**) or errors encountered in other **DEF** statements along the way may abort the command with an error message.

If you specify a line range in a command and no actual program lines fall within that range, MegaBasic immediately terminates the command and displays the message: *No lines.* See the **LIST** command (Chapter 2, Section 2) for more examples of specifying line ranges.

Often, you may repeatedly specify the same line range for several commands in a row. To make this easier, MegaBasic maintains a concept of a *current line range* and lets you refer to this range using the dot-dot notation (..). The current line range is always either the last <*line range*> that you specified with two line numbers (i.e., 100-199), or the range of lines just created or modified by MegaBasic. You can type dot-dot in any context that requires a <*line range*> to specify this range. For example the command **LIST..** will display the current line range on the console, the command **EDIT..** will edit only the lines within the current range, and **REN100.5,..** will renumber the lines in the current range by 5's starting from 100.

The **REN, MOVE, COPY** and **DUPL** commands set the .. range to the range of program lines that they affected. You can specify the .. notation in all commands that act on line ranges. By experimenting with the .. notation you will find ways of using it to streamline the process you go through to develop and maintain your program source.

Specifying Search Strings

Some commands (**LIST, EDIT, CHANGE**) can restrict their scope to lines which contain a user-supplied character pattern called a search string. You can specify a search string in such a command by simply typing the characters you wish to match. You only need to surround a search string with quotes if the search string begins with a digit (0-9), dash(-), lb-sign (#) or period (.), or it contains commas, spaces or quotes. MegaBasic supports two kinds of quotes (“ and ”) which lets you include either quote character within a search string (but not both). The search process excludes the line-number part of a line from the search. Numeric searches will, however, pick up line number references.

Letters in a search string may be in upper or lower case and still match the same set of strings. Question marks (?) act as *wild-card* characters when used within search strings (except as the first character). For example, the string *TH??E* matches words like *There, those, tHeSe, Three, therefore,* etc. A question mark (?) at the beginning of a search string will only match a question mark, not any character.

Search String Option Switches

You can append an additional parameter string to an **EDIT, LIST** or **CHANGE** command to enable or disable additional search capabilities. This optional argument consists of a comma followed by one or more single-character option switches. Each option switch character turns on or off a different feature. If you never specify an option string argument, all options remain off (i.e., disabled). Once you switch an option on, it stays on for all three commands until you explicitly turn it off in a subsequent option string (using the minus (–) option described below). All available option switch characters are individually described below:

Search String Option Switches	
W	Modifies the search so that the string patterns found must appear as complete words, i.e., matches within words or numbers are not considered a match. For example with this option on, the string X only matches lines that contain X all by itself, ignoring words such as *XOR* or MAX$. The W option is normally off when MegaBasic starts up.
***** **&**	Defines the asterisk (*) or ampersand (&) as a special *multi-character wild-card* symbol matching any number of characters when it appears in search strings. For example, *this*that* matches any substring beginning with *this* and ending with *that.* More than one asterisk may appear in a search string to match arbitrary substrings in more than one place. When this option is *on, you* cannot use this *any-string* symbol character as an *ordinary* character in either search or replacement strings. Only one of these special characters is used for the above purpose: whichever one was most recently selected is the one in effect. In the **CHANGE** command, MegaBasic substitutes the text that matched each asterisk in the replacement string (i.e., macro parameter substitution).
(	A left-parenthesis turns on an option that causes the *any-string* wild-card character (i.e., * or &) to include the entire contents of any parentheses encountered, i.e., without ever breaking parenthesized expressions, function arguments or array subscript expressions. For example with this option enabled, the search string (*,*,*) matches any parenthesized argument list containing three or more arguments, even if the outer parentheses contain other items inside parentheses.
#	A lb-sign enables an option that expands the parameter substitution provided by the any-string character (t or &) when used in the search and replacement strings of a **CHANGE** command. When an enabled, you can follow the any-string in the replacement string with a digit to specify which any string character of the search string it corresponds to. For example, the command *CHANGE X(*,*,*),X(*2,*1,*3)* has the effect of swapping the 1st and 2nd subscript expressions of array X( ). In other words, you can refer to the strings matched by the *1st, 2nd* and *3rd* asterisk (or ampersand) in the search string in the replacement string as *1, *2* and *3* (or &1, &2 and &3). You can access up to nine such *parameters* (i.e., digits 1 through 9) in this manner. This option affects only the **CHANGE** command and only if the any-string option is also enabled.
L	L turns on an option that displays every line altered by the **CHANGE** command on the console screen after modifying each line. MegaBasic lists each altered line only once, even if several changes were made to it. This option is particularly useful when you are not *verifying* the changes. The L-option has no effect on **LIST** or **EDIT** commands.
–	A minus sign in the option string causes all option characters that follow it to turn-off their corresponding options instead of turning them on. For example the option string WL-(*# turns on the W and L options and turns off the (,* and # options. Turning off * or & turns off the any-string feature regardless of which of the two characters was in effect.

Option strings are never quoted when they are actually typed. The example above shows them quoted only for descriptive clarification. MegaBasic reports an *Argument Error* if you type any other characters in an option string, except spaces. To further enhance your understanding of the any-string character in search and replacement string, a number of example **CHANGE** commands (abbreviated **CH**) follow below. Each example assumes that some previous command has turned on the (* # options.

Example Change Command	Result Accomplished
CH rem*,rem	Deletes the text of all program remarks.
CH "X(*,*)","X(*,S,*)"	Inserts an additional subscript between the first and second existing subscripts of all references to array X( ). Notice that quotes are needed to allow commas to be part of the string.
CH "X(*,*,*)","X(*1,*3)"	Deletes the second subscript expression from all references to array X( ).
CH "fn(*,*)","fn(*2,*1)"	Moves the leading parameter of function fn( ) to the end of the list in all references.
CH"*;*;*;","*3;*2*1;"	Swap the first statement with the 3rd statement on every line.

Specifying Program File Names

MegaBasic stores programs on disk files managed by the operating system. These files may be given any name which is legal in the host operating system. File names have two parts: a primary name and a secondary name. Both are necessary for the file to be properly identified. For example, you can save a program on a file named *PROGRAM1.XYZ*, where *PROGRAM1 is* the primary name and XYZ is the secondary name. To simplify matters however, you do not need to specify the secondary name of a program file. When omitted, MegaBasic always supplies the default secondary name of *.pgm*. Hence if you supply the name *PROG* (without a secondary name), you are really specifying by default the file name *PROG.pgm*. Therefore you will not normally specify the secondary portion of program file names, although doing so is possible for special purposes.

In MegaBasic commands, program file names are always typed exactly as spelled and without any quotes around them, although you must separate file names from the other items typed in the command with spaces. In MegaBasic statements however, you specify file names with string expressions and string constants in such expressions require quotes around them, for example:

LOAD PROG1	You do not quote the *PROG1* file name because **LOAD** is a command (not a statement).
ACCESS "PROG1"	You must quote the program name here because **ACCESS** is an executable statement and you give the file name as a string expression.

If the file is not on the default drive, you must include the appropriate drive letter in the file name. You specify the drive letter in front of the file name, separated by a colon (:). File names and drive letters can be in upper or lower case with the same effect. For example the following file names all refer to the same program file on drive B:

B:PROGRAM.pgm B:program.pgm b:pRoGrAm.PGM

Under operating systems that support them, you can specify file names with their directory path. This provides access to files in directories other than the currently selected directory. As with file names, you can spell path names in upper or lower case, but MegaBasic converts any lower case characters to upper case internally. Path names consist of a series of directory names, separated by slashes (/ or \), and with no intervening spaces. MegaBasic converts forward slashes to backslashes before using the name internally.

Any legal **MS-DOS** pathname is acceptable to MegaBasic. Hence the file ..\x refers to the file named X in the directory just above the current directory. See your **MS-DOS** operating system users manual for complete information about file pathnames and how to specify them. If the last character of a pathname is a slash (/ or \), then MegaBasic treats the string as the name of a directory instead of a file. Whenever MegaBasic cannot find a pathname on the drive specified or implied, MegaBasic generates a *Directory Not Found Error*. For more information about **MS-DOS** pathnames, consult your operating system manual.

Controlling Command Output

Since you cannot usually read console listings and other displays as they fly by on the screen, several keys may be struck to **STOP** the display, step through it a line at a time, re-start it, and terminate the listing process prematurely. These controls are summarized below:

Ctrl-S ***or*** ***Space-Bar***	Alternates between *pausing* the display and *re-starting* it. When paused, you can abort the current display process by simply typing another command.
Carriage RETURN ***or*** **Line-feed**	Displays successive output lines, one for each key stroke. Effective only during a display *pause*. An output line *may* take up more than one *physical* screen line.
TAB	Displays the next 10 output lines of the current display on each keystroke. Effective only during a *pause*.
Ctrl-C ***or*** **ESC**	Immediately terminates the listing.

You should, in particular, be prepared to press the space-bar immediately after giving the **LIST** command if you are using a fast console screen.

Section 2: Program Entry, Storage and Retrieval

This section describes the commands for entering your own program from the keyboard and listing it back again on the screen or on the printer, and other commands for saving your program on a disk file and loading it back again. The summary below provides a brief synopsis of each command:

ENTER	Automatic line number generation for program entry from the keyboard. You may abbreviate **ENTER** as **ENT**.
LIST	Generates listings of your program and outputs to the console screen, the printer, a text file, or other I/O device. You may abbreviate **LIST** as **LI**.
ENTER KEY	Displays a block of program source code which immediately precedes the last line entered, listed, changed or interrupted for quick reference purposes.
LOAD	Loads a program from a file into the current workspace or into a new workspace. The file must contain a program in binary format, as created by the **SAVE** command, or a program in ASCII format, as created by an editor program.
SAVE	Saves your work onto a file for backup or later access using the **LOAD** command. **SAVE** writes your program in a memory-image binary format.

ENTER *[<starting line number>][,<stepsize>]*

Although you can enter a *single program line* into the program by just typing its line number and contents, the **ENTER** command provides *automatic line numbers for a series of new lines* that you enter. You may optionally specify a *starting number* and *stepsize*, arguments which default to the prior **ENTER** arguments or default to 10 on the first **ENTER**. The following examples illustrate the various options:

ENTER	10, 20, 30, 40, 50,...
ENTER1200	1200, 1210, 1220, 1230,...
ENTER340,2	340, 342, 344, 346, 348,...

After you type the **ENTER** command and press carriage **RETURN**, MegaBasic presents the first line number and waits for you to type a program line. After you finish typing the line and terminating it with a carriage return, MegaBasic gives the next line number in the sequence and you enter another line, and so on. To terminate the process, type a **CTRL-C** or **ESC** at any point or a carriage return *immediately after* the automatic line number appears. Since the last line you entered is always in the editing buffer, you can use editing controls to use all or part of that line in constructing the current line, potentially saving a significant amount of work.

You can backspace over the automatically generated line number and change it into any number you desire. After you enter a line into the program, the next automatic line number will be the *number just entered plus the step-size* specified by the **ENTER** command. You can edit the current line number to re-direct the sequence of automatic line numbers during program entry without typing additional **ENTER** commands.

If the automatically generated line number matches a program line that already exists, MegaBasic displays its contents, positions the cursor on the first non-blank character after the line number and lets you edit the line using the editing control keys described in Chapter 1, Section 6. At that point you can edit the line, skip it by typing a carriage **RETURN**, or get out altogether by typing **CTRL-C**. If you edit or skip the line, MegaBasic resumes the **ENTER** process with the next line number in the series. To correct previously entered lines without leaving the **ENTER** mode, type a **CTRL-K** during the **ENTER** process to go back to the line preceding the one you are on.

MegaBasic does not perform any syntax checks on lines you enter into a program. However, if you forget the *closing quote* on a string constant, MegaBasic automatically adds one to the end of the line. Since this can potentially enclose unwanted characters within the string (e.g., subsequent statements on the same line), MegaBasic provides a warning message to indicate this action. Also, MegaBasic removes *trailing* semicolons from any line that you enter.

LIST *[#<dev>,][<line range>][,<search$>][,<options>]*

Provides a display or printout of your program. You can specify a variety of arguments with the **LIST** command to direct the program **LISTing** to files or different output devices and to restrict the listing to only a portion of the entire program. All arguments are optional and MegaBasic assumes specific default values when you omit them. Each argument and its default value is summarized below:

<dev>	Specifies where the program listing is to be sent. MegaBasic uses the console (device #0) if you omit the <device>. You may supply a device number to send the listing to the printer (#1), to an open file, or to another output device. Be sure to type a pound sign (#) in front of a device number to distinguish it from a line number.
<line range>	Specifies the range of line numbers to **LIST**. The entire program is **LISTed** when no line range is given. See the discussion about specifying program line ranges on Chapter 2, Section 1.
<search$>	Specifies a string of characters which must be present in each line in the program listing. MegaBasic excludes all lines from the **LISTing** that do not contain the <*search$*> specified. When you omit the <*search$*>, MegaBasicincludes all lines in the **LISTing.** See the discussion on specifying search strings in Chapter 2, Section 1.
<options>	Specifies zero or more single-character switches that alter the way that MegaBasic conducts subsequent program line searches. See the discussion on specifying option strings back in Chapter 2, Section 1.

LIST is extremely flexible because of the many combinations possible. The following examples illustrate possible **LIST** commands along with a description of what they do.

LIST	List the entire program on the console.
LIST #1	List the entire program on the printer.
LIST450	List line number 450. A line number all by itself specifies a line range of one line.
LIST300-675	List all lines with numbers in the range from 300 to 675, inclusive.
LIST-500 **LIST0-500**	List all lines numbered 500 and lower. Omitting the leading line number of a range but including the dash indicates a range that begins with the first line of the program.
LIST225- **LIST225-$**	List all lines numbered 225 and higher. By omitting the ending line number of a range but including the dash indicates a range ending with the last line of the program.
LIST.-499	Lists all lines from the *current line up* line number 499.
LIST100-.	List all lines from line number 100 up to and including the *current line.*
LIST..	List all lines of the *current line range.*

MegaBasic uses the optional string argument to search through the line range given and list only those lines containing that string. You can include question marks (?) in the search string to act as *wild card* characters that match any character (see the information about this feature under the **EDIT** command in Chapter 2, Section 1). Upper and lower case letters match as the same letters.

LISTREM	List all the **REMark**s in the program.
LISTDEFPROC	List all the procedure definitions in the program.
LIST#1,0-99,write	List to the printer all lines below 100 containing **WRITE** statements.
LIST M?$=	List to the console all lines containing assignments to string variables having names 3 characters long that begin with the letter M.
LISTTOT&SUM,&	List all lines containing TOT followed later in the same line by SUM. The & option switch turns & into a symbol that matches zero or more characters.

You only need to specify the <*device*> number to direct the program listing to an output device other than the console (device #0). Usually this would be the printer (device #1), but may also be an opened file number. The resulting file contains pure text suitable for subsequent **LOADing** and also for processing by other text file utilities (e.g., text editors and formatters) which cannot handle the coded format of *normal* MegaBasic program files. See Chapter 7, Section 1 for further details about text file processing.

MegaBasic maintains your programs in an encoded representation for highspeed execution. The **LIST** command reconstructs readable program lines from this format as the program **LISTing** progresses. Therefore, the lines **LISTed** may not appear exactly the way you typed them originally. In particular, MegaBasic always displays reserved words (e.g., **FOR, NEXT, READ, WRITE**, etc.) as capitalized with trailing lower case (e.g., For, Next, Read, Write, etc.), and displays all variable, function, procedure and label names in all upper case. **REMark**s and quoted strings are always **LISTed** exactly the way you typed them.

Carriage Return

Typing a carriage return all by itself in the command mode has a special purpose. It will display a block of program source which precedes the last line **LISTed**, edited, entered or interrupted during program execution. For example if your program encounters an error and aborts with one the various built-in error messages, you can immediately view the region of the error by typing a carriage return. Likewise, if you have been editing a group of lines for a while, you can view your work by getting out of the edit mode (using **CTRL-C** or **ESC**) and typing a carriage return. Typing additional carriage returns will display successive lines that follow the initial group displayed.

To determine how many lines to display, MegaBasic scans backwards through the source from the current line back to a line beginning with a **REMark** or preceded by a line-feed, up to a maximum of 12 lines. In this manner, you can view the most recent *logical group* of lines at the touch of a button (**RETURN** key). When MegaBasic reports an execution error message with a line number, it sets the *current line* to that line number. This permits rapid review of the source region leading up to the error.

SAVE *[<program file name>]*

Saves a program from memory out onto a file. If you omit the file name, the program in the current workspace is written to a file bearing the name of that workspace. This is the preferred way to **SAVE** a file because it avoids having to remember, type and spell correctly the destination file name. In so doing, you can avoid spelling errors and back up your work so easily that you will tend to save it more often, rather than put it off.

If you supply the **SAVE** command with a file name, MegaBasic first compares it with all the currently defined workspace names. If it matches any one of them, MegaBasic aborts the **SAVE** to prevent multiple workspaces with the same name. Otherwise, MegaBasic saves the program in the current workspace to the file specified and renames the workspace to that new name. The following examples illustrate each of the possible forms of **SAVE**:

SAVE filename	Writes the current program onto the file specified. If the file name extension is **.PGM** then you need not type it. You must specify the file name extension if it is not a **.PGM** file.
SAVE	Writes the current program onto the same file that it was most recently **LOADed** from or **SAVEd** to. MegaBasic supplies a default name of **UNNAMED.pgm** if you have not yet assigned a name to the program.
SAVE d:	Writes the current program onto the same file name, but to the specified drive. Notice the colon after the drive letter, indicating the letter is a drive code, not a file name.
SAVE d:\path	Writes the current program onto the same file name of a different directory and / or drive. Notice that the pathname ends in a slash to indicate it is a directory, not a file.
SAVE d:\path;\file	Saves the program on the file specified by the complete file pathname given. You can specify pathnames in any form allowed by the operating system.

Regardless of how you specify the **SAVE** command or what workspace contents you are saving, MegaBasic asks you to confirm your request with a yes / no response after displaying the entire file name and indicating whether or not the file already exists in the

file directory. Answering N (for no) aborts all further **SAVE** action. Answering Y (for yes) saves the program to the file indicated, which is automatically created if not already present.

When there are modified programs in other workspaces and you give a **SAVE** command without specifying any file name, MegaBasic asks you if you want to save all modified workspaces. Answering *no* (N) causes the usual **SAVE** of the current workspace. Answering *yes (Y)* causes MegaBasic to sequence through each unsaved, modified workspace while letting you confirm or deny a **SAVE** on each one. This automatic **SAVE** option is only requested if other workspaces containing modified programs are present. To **SAVE** an unmodified program, you have to specify a file name in the **SAVE** command; **SAVE** by itself does nothing if no programs in memory have been modified.

The **SAVE** command detects when another user (in a multi-user operating system or network) has modified a program file you are about to **SAVE** and issues a warning that you are about to overwrite their changes. You are then given the opportunity to abort the **SAVE** or go ahead with it. This check is performed ONLY when you are saving to the file from which you **LOADed** the program, i.e., a **SAVE** with no arguments that uses the previous **LOAD** name.

Files written with the **SAVE** command are exact memory images of the program in its internally encoded form. Therefore other programs such as editors and other text processing software unaware of the program structure within the file cannot process MegaBasic program files. Furthermore, earlier **Z80** versions of MegaBasic cannot execute these files as programs. Whenever you **SAVE** to an existing file, any program that was loaded and converted from a text or other non-pgm format, MegaBasic informs you that you are about to write your program in Binary Format, and requests your confirmation. By answering N (for no), the **SAVE** is aborted. This extra confirmation is not requested if the destination file is new.

The **MS-DOS** and Xenix 386 operating systems organizes files in a hierarchical structure of files and subdirectories. To access a file, you must therefore specify a path of names from the top of the hierarchy down to the desired file. MegaBasic supports pathnames in any form acceptable to the host operating system. For example the file ../x refers to the file named X in the directory just above the current directory. Consult your operating system user's manual for detailed information about how to use and specify pathnames. You should also read the material in Appendix B, Section 1 of this manual for some differences between **MS-DOS** and Xenix regarding the formation of correct file and directory pathnames.

One word of caution is in order here. There have been various pathname management utilities for **MS-DOS** operating system to allow programs which were never designed to work in the pathname environment to use files in some or all subdirectories. Such programs may make files in those subdirectories appear as if they exist in the current directory. This can cause problems with MegaBasic or other programs that have been properly designed to take full advantage of the pathname environment, as they can be *fooled* into thinking that such files really do reside in the current directory. Therefore avoid such programs when using MegaBasic. Although it may work, MegaBasic is in no way guaranteed to work in systems that have such programs installed.

LOAD *<program file name* list>

Loads one or more program files into memory and displays their size and the date/time of most recent modification. If no program is present in the current workspace, MegaBasic loads the file into memory without further attention. If a program is already

present, MegaBasic lets you choose to either replace it with the incoming file, or else preserve it and load the file into a new workspace. In either case, MegaBasic selects the receiving workspace as the *current workspace* and assigns it the name of the file just **LOADed**. Using successive **LOADs**, you can bring into memory, one at a time, up to 64 programs simultaneously, limited of course to the amount of memory actually available in your machine.

When you specify more than one program file in the **LOAD** command, they are each **LOADed** into separate workspaces from the one you are in, and leaves you in the same workspace from which you started. The file names must be separated from one another in the **LOAD** command with spaces.

Before erasing the contents of a workspace prior to loading another program into it, MegaBasic looks to see if it contains original work which would be lost. If so, MegaBasic informs you and gives you the opportunity to abort any further **LOAD** action. MegaBasic never lets you destroy original, unsaved work without confirmation.

If the program file is not found in the directory implied in its name, MegaBasic searches each of the subdirectories specified in the **MS-DOS** alternate **PATH**= list maintained by the operating system, in order to find the program. See Chapter 10 for further details about the file lookup order.

Wherever the program came from, MegaBasic retains its full drive and pathname so that any subsequent **SAVE** commands can write the program back to its original file and directory no matter what the currently selected directory happens to be.

Loading Programs Stored in Text Format

MegaBasic programs are normally stored in a special coded form on the file. Occasionally, you may have a text file containing program text from another system or different dialect of that you wish to convert to the MegaBasic system. You can store MegaBasic programs in ASCII text format, by simply **LISTing** the program source to an open file number. This format can be processed by any text processing facilities present on your computer system, and then **LOADed** back again as needed. To be acceptable, a text file must conform to the following rules:

- Each line ends with a carriage return (ASCII 13). When a line-feed and a carriage return appear in pairs, MegaBasic *ignores the second character* of the pair (discarding it) and uses only the first character.
- MegaBasic ignores empty lines, i.e., lines consisting of only spaces and a carriage return.
- MegaBasic reads text files to the end or until it reads an end of file mark (normally an ASCII 26 code). You can specify a different code using **PARAM**(9) if the normal ASCII 26 value is inappropriate.
- Lines do not have to begin with line numbers, but MegaBasic accepts them if they appear. To lines without line numbers, MegaBasic automatically assigns line numbers one greater than the previous line loaded. Hence a text file without any line numbers will be automatically numbered: 1, 2, 3, and so on. MegaBasic uses line numbers to decide how to order the lines as it loads them.
- Lines longer than 255 characters are broken into two or more lines of no more than 255 characters each. These resulting lines will likely require editing, due to the arbitrary divisions imposed. You should therefore try to avoid such long lines whenever possible.

- The text file must not exceed 65535 bytes in length. Attempting to **LOAD** files longer than this will result in a *Length Error*.
- Upper and lower case do not matter, but MegaBasic will impose its own upper / lower case conventions on the resulting program.

Program Version Control

To help you manage the various versions of your MegaBasic program, MegaBasic automatically maintains a count of the number of times that you **SAVE** your program. This count is incremented each time you **SAVE** your program and the program has changed since the last time you saved it (i.e., redundant **SAVE**s do not count). You can access this count by opening the program file and reading the count word (16-bits) from position 14 in the file:

```
Open #5,"program.pgm"; filepos(5) = 14;
Read #5,@count; close #5
```

Given two versions of a program, their counts can tell you which version is the most recent one, regardless of their dates in the file directory. At any time, you can alter the count field directly (using a **WRITE** statement) for special purposes (but be sure you alter nothing else around it). **SAVE** *counts will* wrap around to zero after 65536 **SAVES**, but such a number is not likely.

Section 3: Editing and Alteration Commands

These are all the commands provided by MegaBasic to revise your program by editing source lines, to make global substitutions, to merge programs from other files into the current program, and other modifications. The following list summarizes them:

EDIT	Selects program lines by range and optional search string for rapid editing and display. You can abbreviate **EDIT** as **ED**.
CHANGE	Replaces one string for another throughout a range of program lines. One-at-a-time user-verify option, and wild-card characters within the target string are available. You can abbreviate **CHANGE** as **CH**.
NAME	Instantly renames any user-assigned name (identifiers, labels, etc.) as any new name. It is also able to display all existing names ordered alphabetically. You can define various selectors to restrict the names displayed to those with certain attributes.
DEL	Deletes any range of program lines.
REN	Renumbers any range of lines to any other range with a fixed increment. You can rearrange program blocks by renumbering them to the desired destination sequences. MegaBasic updates all line number references to renumbered lines.
MOVE	Moves any range of program lines to another starting line number. Preserves the increments between the lines moved. MegaBasic maintains line number references accordingly.
COPY	Creates a second copy of a range of program lines, renumbers it, then inserts it at another line number in the program. MegaBasic updates line number references within the copied lines.
DUPL	Duplicates any range of program lines and inserts them into the program at any line number. **DUPL** preserves the increments between line numbers and maintains local line number references.
MERGE	Merges selected lines from a program file or workspace into the current program by line number. You can specify the starting line number where you want the merged lines to go and source line ranges to merge. **MERGE** replaces existing lines with new lines wherever their line numbers match.

EDIT [<line range>][,<search string>][,<options>]

Lets you edit any sequence of lines in your program. MegaBasic displays each line and positions the cursor at the beginning of the line (on the line number). You can then edit this line using the editing control keys described in Chapter 1, Section 6. Only when you type the **ENTER** key (or carriage return) will MegaBasic accept this line and proceed to the next one in the line range specified or implied. You may skip over a line, leaving it unchanged, by typing a carriage return without making any changes. You can specify several optional arguments to focus your efforts on the exact area of interest:

<line range>	Specifies the range of lines that you wish to sequence through for editing. Unlike all other commands that use line ranges, a single line number implies the range of lines from the number specified to the end of the program.
<search strings>	Specifies a search pattern that MegaBasic uses to filter out lines that you do not wish to edit. MegaBasic only presents you with the lines that contain the search string and excludes all others.
<options>	Specifies zero or more single-character switches that alter the way MegaBasic conducts program line searches. See the earlier discussions in this Section for a complete explanation of <*options*> (Chapter 2, Section 1) and <*search strings*> (Chapter 2, Section 1).

The editing process steps from line to line until you have edited the last line of the line range, or until you type a **CTRL-C** or **ESC**, or until you enter a new command. Anything without a line number is considered a command and therefore if the first key you type (after a line is presented) is not a digit or editing control character, edit mode exits automatically and the character becomes the first character of the next command. Whenever you exit the edit mode while a line is presented on the screen, MegaBasic erases the line shown before accepting the next command. If you want to bring it back and continue editing, use **F5** or **Ctrl-R**.

At any time during the **EDIT** mode you can type a **Ctrl-K** (up-arrow on some terminals) to edit the line preceding the current line being shown. Repeated use of **Ctrl-K** sequences backward through the program one line at a time. See Chapter 1, Section 6 for a complete explanation of this key. To get the most out of the **EDIT** process, you should understand the material presented in Chapter 1, Section 6.

If you specify the <*search string*>, MegaBasic will present only those program lines (within the given line range) that contain the string specified. Upper and lower case letters in the search string are equivalent. You must enclose the string with quotes (“ or ”) if it contains any commas, significant leading or trailing blanks, or it begins with a period (.), digit or dollar sign ($). When MegaBasic finds a line that contains a <*search string*>, the entire line is displayed and the cursor is positioned in the line where the <*search string*> was found. At that point you can edit the line or skip it by typing a carriage return, after which MegaBasic skips to the next line containing the <*search string*>.

For flexibility, your search string may contain special *wild card* characters that match any character. This special character, a question mark (?), may appear anywhere in the search string (except as the first character) and as many times as desired. With this concept, the string A??= will match all assignment statements with variable identifiers 3

characters long beginning with the letter A. The following examples illustrate and describe each of the various forms of the **EDIT** command:

EDIT	Edit the program sequentially from the first line.
EDIT175	Edit sequentially from line 175 and on up.
EDIT200-299	Edit the lines in the 200 range and **STOP**.
EDIT.subr	Edit the lines in the function or procedure named *SUBR* and **STOP**.
EDIT"i,j"	Edit all lines containing the string i,j.
EDIT""	Edit all lines containing a double quote (")
EDIT850,rem	Edit lines containing *rem* numbered 850 and up.
EDIT300-499,read	Edit lines containing the string *read* in the lines numbered from 300 to 499.
EDIT M???=	Edit all lines containing a five character sequence beginning with *M* and ending with =.

After editing a line and entering it, MegaBasic automatically presents you with the next line that follows it in the line number sequence. Because of this, if you edit the line number the edit will continue from that point in the program. You can re-start the editing sequence anywhere in the program by simply typing an unused line number at the desired starting point (followed by a carriage return). This normally deletes the line there, so be sure that the number you select is not in use.

You can edit an executing program after you interrupt it with a **CTRL-C**. Afterward, you can usually continue its execution from where you interrupted it. This may be desirable when, during debugging your program, you discover a programming error requiring a small correction. However, there are certain program lines which you cannot alter without disrupting program continuability. When you edit such a line, MegaBasic will inform you with the message: *Program continuation no longer possible.* Consult the **CONT** command in Chapter 2, Section 4 for complete information about the effect of program alteration on execution.

When an error occurs in a running program, MegaBasic places a copy of the line in which the error occurred into the editing buffer. This lets you immediately examine and modify the offending line after MegaBasic reports an error in it. MegaBasic does not automatically display an erroneous line, but you can access it by typing the appropriate previous line access control keys.

Whenever you enter new or edited lines into a program, MegaBasic does not, in general, perform any syntax checks on the line. However, there are two corrective actions MegaBasic takes automatically. If you forget the closing quote (' or ") on a string constant, MegaBasic automatically adds one to the end of the line. Since this can potentially enclose unwanted characters within the string (e.g., subsequent statements on the same line), MegaBasic provides a warning message to indicate this action. Also, since some users have a tendency to place a semicolon (;) at the end of a program line, MegaBasic removes trailing semicolons from any line that you enter.

CHANGE *[<range>,] <search$>,<replace> [,<opts>]*

Global *search-and-replace* may be done with the **CHANGE** command, which replaces one string with another everywhere or selectively within a line range. After you enter a **CHANGE** command, MegaBasic will request a yes or no response to the question: *Verify?*. An *no* response causes an immediate replacement of all occurrences found. A yes

response tells MegaBasic to request confirmation of each replacement before actually making any change. This allows you to individually control each replacement as it happens. Upon completion of a **CHANGE** command, MegaBasic displays the number of changes made. The paragraphs below summarize the various **CHANGE** command arguments:

<range>	Specifies the range of lines that you wish to search and change. When you omit the <*range*>, **CHANGE** acts upon the entire program.
<search$>	Specifies a string of characters that you want replaced by another string. Lines which do not contain the search string remain unchanged. Special characters can be used to match any character or string of characters.
<replace>	Specifies the string that you want to substitute for each instance of the <*search$*>.
<opts>	Specifies zero or more single-character switches that alter the way MegaBasic conducts program line searches.

These arguments are fully described in Chapter 2, Section 1 and you should understand this material in order to take full advantage of the **CHANGE** command. When you include *wild-card* characters in the <*search$*>, *you* should use the verify option to avoid unintentional replacements. The following examples show various ways you can type **CHANGE** commands:

CHANGE this, that	Change the string *this* to the string *that* throughout the program.
CHANGE "A(i,j)", V(j)	Change *A(i,j)* to *V(j)* throughout the program.
CHANGE th??e, those	Change sequences that match *th??e* to *those.*
CHANGE 100-199, x, y	Change x to y in the line range 100 to 199.
CHANGE 560, 23, -23	Change 23 to -23 throughout line 560.

It is wise to use the verify option when you specify a numeric search string in a **CHANGE** command. Short numbers can easily occur within longer numbers and unintentional replacement can cause considerable work to repair. To a lesser degree, unintentional replacement of sub-strings can occur with any search string, and for that reason you should be careful using **CHANGE.** When in doubt about what a search string will match, you can always try it first in a **LIST** command to see what matches before changing your program.

CHANGE is a very general purpose tool that you can apply in a wide variety of situations. However if you are changing line numbers or renaming user assigned names (e.g., variable or function names), you should employ the **REN** and **NAME** commands for these purposes instead of using the **CHANGE** command. These special purpose tools not only execute faster, but they perform their specific task automatically and completely. For example, when **REN** changes a line number, it also changes all references to that line number, wherever they may be throughout the program (search strings do not even access the line number portion of a program line). The **NAME** command can rename a variable I to J without changing all the other I's to J's that are not variables (e.g., in remarks or string constants).

You can append an additional parameter string to a **CHANGE** command, called option switches, to enable or disable various additional search capabilities. This optional

argument consists of a comma followed by one or more single-character option switches. Each option switch character turns on or off a different feature. Once you switch a feature on, it stays on for the **CHANGE**, **EDIT** and **LIST** commands until you explicitly turn it off in a subsequent option string. See Chapter 2, Section 1 for description of all available option switch characters.

Two of the option switches let you define either the ampersand (&) or the asterisk (*) as a special character in a <*search string*> that matches any string of characters. To assist your understanding of this any-string character in search and replacement strings, a number of example **CHANGE** commands follow below. Each example assumes that some previous command has turned on the (, * and # options, and that you have read and understood the option switch material presented back in Chapter 2, Section 1:

Change rem*, rem	Removes the text from all program remarks.
Change "X(*,*)","X(*,S,*)"	Inserts an additional subscript between the 1st and 2nd existing subscripts of all references to array X( ). (Quotes preserve the commas.)
Change "X(*,*,*)","X(*1,*3)"	Deletes the 2nd subscript expression from all references to array X( ).
Change "fn(*,*)","fn(*2,*1)"	Moves the leading parameter of function fn( ) to the end of the list in all references.
Change*;*;*;, *3;*2;*1;	Swap the 1st statement with the 3rd statement on every line.
Change*;*;*,*1;*2;*3;*2	Append a copy of the 2nd statement on the line to the end of the line, on every line.

The important thing to understand from these examples is how to manipulate text by specifying only its surrounding context. You should try out these techniques on some practice source programs (without saving the results) to get a good feel for how they work. When appropriate, any-string substitutions can replace many hours of editing with a few minutes of effort.

NAMES *[#<device>,] [<selector* LIST*>]*

Without any arguments **NAME** invokes an alphabetical listing of all user-assigned names used for variables, functions, procedures and line-labels. This allows a quick review of the names you have assigned to all the objects in your program which you have defined. Typographical errors in such names will generally appear near the correct spelling because of the alphabetical ordering of the display. You can control long or overly rapid displays using the same pause controls supported by the **LIST** command. You can direct the **NAMES** listing to any output device or open file by specifying the *#<device>* argument.

The **NAME** command can also display the names of entities with specific attributes, by listing the desired attributes as arguments to the **NAME** command. For example, **NAME FUNC** will display all the user-defined function names; **NAME $ FUNC** will display only those functions which return a string result. The attribute selectors may be any from the list.

FUNC	User-defined functions of any data type
PROC	User-defined procedures
STRING	String variables, arrays and functions
DIM	Variables currently defined as arrays
REAL	Floating point variable and function names
INTEGER *or* INT	Integer variable and function names
STRUCT	Structure fields of any type
SHARED	Shared entities used in the current program
GOTO *or*:	Line label and line number references
NOT *or*"	All selectors following **NOT** or a minus sign become *de-selectors*, i.e., matching items are *omitted* from the listing.

You can type any combination of selectors in any order after the **NAME** keyword, separated from one another by spaces. MegaBasic displays only those names that satisfy all *the selectors* specified, for example:

NAME SHARED $ FUNC	Displays all string functions in use by the current program which some package has defined as **SHARED**.
NAME DIM $	Displays the names of all string array variables.
NAME:	Displays all the line labels in the program.
NAME $ NOT FUNC DIM	Displays any string name not a function and not dimensioned (i.e., simple string variables).
NAME NOT : STRING	Displays all names except strings and line labels.
NAMEINTEGER FUNC	Displays the names of all integer functions
NAMEINTEGER DIM	Displays the names of all integer arrays.
NAMESHARED REAL	Displays names of all floating point variables and functions which are currently declared **SHARED.**

The **NAME** command depends on the current data type defined by each name. To obtain this information, MegaBasic processes all **DEF** statements in when you type the **NAME** command. If there are syntax errors in any **DEF** statements then MegaBasic aborts the **NAME** command and reports the error found. Also, the names of arrays and the names of **SHARED** objects defined in external packages will not be shown unless you have executed the **ACCESS** statements that bind the **SHARED** names to their references. **NAME** provides a count of the names it displays after listing them.

NAME *<old name>, <new name>*

By following the **NAME** command with two names (of identifiers), separated with a comma, MegaBasic will instantly rename all occurrences of the first name as the second name *throughout* the program. An error message results if the first name does not appear anywhere in the program or if the second name is already in use or if either name is a MegaBasic reserved word. An error also occurs if you quoted either name like a string; just specify the names as if you are typing them into your program.

This command is specifically designed for renaming identifiers and since it *is not* a string match-and-substitute process, it will *not affect* **REMarks** or quoted strings which contain similarly spelled character sequences. You cannot restrict **NAME** to a line range and no verify option is available (unlike the **CHANGE** command). You can spell **NAME** as either **NAME** or **NAMEs**.

DEL *<line range>*

Deletes the specified line range from your program. A dollar sign ($) may used to denote the last line of the program. Use **DEL** for block deletions rather than single line deletions, because you can more easily delete a single line by typing its line number and an immediate carriage **RETURN** (i.e., an empty program line). For example **DEL 30~399** deletes all lines in the 300 range. See the discussion on specifying line ranges back in Chapter 2, Section 1.

REN *[<star ting line>[,<stepsize> [,<line range>]]]*

Provides a general program renumbering facility that renumbers any range or subrange of lines to any other range. MegaBasic does not permit renumbering that would cause line interleaving or duplicate line numbers. However it does support rearrangement of whole groups of lines as well as *simple* renumbering, given the appropriate command. MegaBasic adjusts all references made to lines renumbered by the process, wherever they may be throughout the program. Each of the arguments to **REN** is optional and MegaBasic assumes specific default values for them when you omit them, as described below:

<starting line>	Starting line number where you want the renumbered lines to begin. Line number 10 is used if you omit this argument, which renumbers the entire program by 10 from 10.
<step size>	Increment between the renumbered lines and defaults to 10 if omitted. It must be 1 or greater and it cannot be so large that it forces any line number beyond 65535.
<line range>	Range of lines to renumber in your program as they are before renumbering. If you omit the *<line range>*, the entire program is renumbered.

All arguments are optional, but you have to omit them from right to left. The following examples illustrate how you might apply the **REN** command:

REN	Moves entire program to 10 by 10s.
REN 250	Moves entire program to 250 by 10s.
REN 375,5	Moves entire program to 375 by 5s.
REN500,12,2000-$	Move lines numbered 2000 and up into the range 500 by 12s.
REN200,3,800-899	Move all lines in the 800 range to 200 by 3s.

MegaBasic always validates the implied operation that you request and aborts with an *Out Of Bounds Error* to prevent overlapping line ranges or illegal line numbers. MegaBasic always properly updates references to renumbered lines throughout the program. Line number references to nonexistent lines remain unchanged. The resulting range of lines affected by any **REN** command will become the .. current line range (see Chapter 2, Section 1).

MOVE *[<star ting line>][, <line range>]*

Moves lines from any range of line numbers to a new starting line number, while maintaining the existing increments between the lines. Line number references to the lines moved are automatically updated throughout the program as needed. Both of the arguments to **MOVE** are optional and MegaBasic assumes specific default values for them when you omit them, as described below:

<starting line>	Starting line number where you want the block of lines to be and defaults to line 100 if you omit it (i.e., **MOVE** without arguments moves the entire program to line 100).
<line range>	The block of lines that you wish to move, prior to moving them. When you omit this argument, MegaBasic moves the entire program to the <*starting line*> specified. See the complete discussion on specifying line ranges in Chapter 2, Section 1.

MOVE is like the **REN** command but without any line increment step size. The following examples illustrate the variety of ways to type **MOVE** commands:

MOVE	Move the entire program so that its first line starts at line number 100.
MOVE 4000	Move the entire program so that its first line begins at 4000.
MOVE 335, 450	Move line 450 to line number 335.
MOVE 800, 500-$	Move all program lines numbered 500 and up so that the first of these begins at line 800. This form is particularlyuseful for opening up *holes* in the line number space for a new block of program lines.
MOVE 900, 300-399	Move lines in the 300 range to the 900 range.

MegaBasic validates the implied operation that you request and aborts with an *Out Of Bounds Error* to prevent overlapping line ranges or illegal line numbers. MegaBasic

properly updates references throughout the program to line numbers that have moved. Line number references to non-existent lines remain unchanged. After a **MOVE** command, the .. current line range, discussed in Chapter 2, Section 1, is the set of lines moved.

COPY *[<starting line> [,<step> [,<line range>]]]*

Copies all lines within one line range to a second empty line range, leaving the original lines intact and unchanged. Line number references to both the original and the copy are properly maintained. Each of the arguments to **COPY** is optional and MegaBasic assumes specific default values for them when you omit them, as described below:

<starting line>	This is the starting line number where you want the new block of lines to begin. MegaBasic assumes line number 10 for this argument if you leave it off. Omitting it also implies that you have also omitted the other arguments as well.
<step size>	This specifies the increment or spacing between the copied lines. It defaults to 10 if you omit it. The step size must be 1 or greater, and it cannot be so large that it forces the last line number beyond 65535. MegaBasic traps both of these errors.
<line range>	This specifies the block of lines that you wish to copy, as they are numbered before the copy. When you omit this argument, MegaBasic copies the entire program to the *<starting line>* specified. See the complete discussion on specifying line ranges in Chapter 2, Section 1.

COPY works just like renumber, except that the original lines remains unchanged and a renumbered copy appears elsewhere in the program. The following examples illustrate the various forms of **COPY**:

COPY	Copy the entire program to line 10, stepping by 10.
COPY10000	Copy entire program to line 10000, stepping by 10.
COPY4000,5	Copy entire program to line 4000, stepping by 5.
COPY350.1.475	Copy line 475 to line number 350. The step size of 1 is superfluous because you are copying only one line. Editing line 475 to change its line number to 350 might be easier.
COPY 100,20,400-499	Copy the lines in the 400 range to line 100, incrementing by 20 between lines.

MegaBasic always validates the implied operation that you request and aborts with an *Out Of Bounds Error* to prevent overlapping line ranges or illegal line numbers. MegaBasic properly updates all line number references throughout the program to both the original and the copied lines. Line number references to non-existent lines remain unchanged.

DUPL *[<star ting line>][, <line range>]*

Duplicates all program lines within a line range to a second empty line range, using the same increments between the lines as the original. **DUPL** relocates line number references as needed. Each of the arguments to **DUPL** is optional and MegaBasic assumes specific default values when you omit them, as described below:

<starting line>	Destination line number of the new block. Omitting it (i.e., no arguments) duplicates the entire program at line 10.
<line range>	Specifies the source block range to duplicate. When you omit this argument, MegaBasic duplicates the entire program to the <*starting line*> specified. Line range specification is described on Chapter 2, Section 1.

DUPL is like the **COPY** command without the line increment argument. The examples below illustrate a variety of **DUPL** commands:

DUPL	Copies the entire program into line 100 with the same inter-line step sizes.
DUPL12000	Copies of the entire program at line 12000 with the same line increments.
DUPL1255, 425 425	Copies line 425 on line 1255. You can do the same thing by editing line to change its line number to 1255.
DUPL500, 200-$	Copies all lines 200 and up and put the first line to 500.
DUPL1200, 800-999	Copies lines within the 800 to 999 range to 1200.

MegaBasic always validates the implied operation that you request and aborts with an Out *Of Bounds Error* to prevent overlapping line ranges or illegal line numbers. MegaBasic properly updates all line number references throughout the program to both the original and the duplicate lines. Line number references to non-existent lines will remain unchanged.

MERGE *<pr ogram> [<source/destination specs>]*

The **MERGE** command provides a general facility for adding MegaBasic code lines from other files or workspaces to your current program. It combines the lines of the two programs according to their line numbers. Source lines with the same line numbers as lines in the target program *replace* those target lines; source lines with differing line numbers are *inserted* into the target program. Meaningless code may result from overlapping and interleaving lines indiscriminately.

The <*program*> argument specifies the name of either a file or a workspace. If you specify a file, it is brought into memory for the **MERGE** operation and removed upon completion. If you specify a workspace name (i.e., the name of a program already in memory), the **MERGE** is performed without modifying its contents and the source program remains in memory on completion.

Without any further arguments, the *entire* source program is merged into your target program. However you can follow the <*program*> argument with *one or more* additional arguments that specify *where to put* the merged lines into your target program and *line ranges to merge* from the source program. You have to separate multiple specifications

from one another with a comma and each specification can take any of the following forms:

<start>	Merges *all source lines* into the target program starting at the line number specified.
<from>-<to>	Merges source lines from the *range specified* into the same line numbers of the target program.
<subr name>	Merges the source lines of the *named subroutine* (i.e., a function or procedure name) into the same line numbers of the target.
<start>:<from>-<to>	Merges source lines from the *range specified* into the target program at the *starting line* number specified.
<start>:<from>-	Merges all source lines *at or above* the line specified into the target program at the *starting line* number specified. The dash (-) is optional.
<start>:<subr name>	Merges the source lines of the *named subroutine* into the target program at the *starting line* number specified.

For example, **MERGE PROG 100:500-699, 200:SORT, CALC** merges lines 500 to 699 from program **PROG** into the current program at line 100, all lines of subroutine **SORT** into line 200 and all lines of subroutine **CALC** into the same line numbers they already have.

The <*start*> line number actually renumbers the incoming program lines so that they begin on the line number you specified. MegaBasic accomplishes this by adding the appropriate constant value to each and every line number and line reference so that the *beginning* line number comes out as desired. This renumbering process affects neither the current program nor the contents of the source program file you are merging. MegaBasic does not proceed with any merge that would lead to line numbers greater than 65535 and reports such a case as an *Out Of Bounds Error*.

Perhaps the most useful capability is the <*subr name*> specification, which lets you merge procedures and functions from program files or other workspaces directly into your program *by name*. When you specify this *symbolic line range*, **MERGE** searches the source program for a procedure or function by that name. If found, the effective line range specified consists of all lines of the subroutine starting with the initial **DEF** statement, along with any immediately preceding **REMark** lines, up to and including the line containing its terminating **FUNC END** or **PROC END** statement. Failure to find the named subroutine in this manner terminates **MERGE** with an appropriate error message. **REMarks** that follow subroutines are not included. Incorrectly formed subroutines (e.g., missing **FUNC**/**PROC ENDS**) or errors encountered in *other* **DEF** statements along the way may also terminate **MERGE**.

Since you can specify multiple ranges and errors do terminate **MERGE** operation, MegaBasic *describes* each range as it is being merged so that you can tell how far it progressed if an error does occur. Each source / destination specification is processed and completed from left to right as specified in the command and any error encountered will immediately terminate further **MERGE** processing.

Automatic Target Placement

Normally, the merged lines go into the specified target lines numbers, displacing anything that resides there. Often however, this invites problems and mistakes when

you in the process of building a new program by pulling in blocks of code from other programs .

By preceding any <*start*>:<*source*> specification with a plus sign (+), MegaBasic searches for an *available* target range beginning at or above the <*start*> specified large enough to hold the <*source*> lines. The target destination will *always* be a block of line numbers beginning on a *multiple of 100.* If no <*start*> was specified, the search begins at line 1000. An *Out Of Bounds Error* occurs if no available target region could be found.

Section 4: Execution Control and Debugging Commands

This section describes the MegaBasic commands that you use to run your program, test for bugs, stop and examine some variables, continue where you left off with single-stepping, resume until some condition becomes **TRUE** or **FALSE**, etc. The flexible debugging environment provided for controlling and monitoring program execution reduces the effort needed to fully develop software, so any extra effort you spend to master these commands will quickly pay off with more productive testing and debugging sessions. The list below summarizes the execution and debugging commands:

Direct Statements	MegaBasic always attempts to *execute* any line of MegaBasic statements without a line number.
RUN	Clears memory, evaluates the static definitions (**DEF** statements) and begins program execution.
Ctrl-C	Aborts the execution of any program in progress and puts you into the command level. The program may be re-started later again later.
CONT	Re-starts a program that you previously interrupted with a **CTRL-C**, or one that interrupted itself with a programmed **STOP**. Continuation is possible even if you have modified the program, changed variable values, saved it on a file, or performed virtually any other command operation.
TRACE	Selects various options that show the progress and current state of program execution at the program source level as execution proceeds. **TRACE** provides many different options and controls for selective display and conditional invocation of execution tracing. **TRACE** modes are set or reset on a workspace by workspace basis.
CHECK	Quick check of the program in the current workspace for common syntax errors like wrong line numbers, improperly formed loops, unbalanced parentheses, etc. **CHECK** reports all errors at once.

Executing Direct Statements

Whenever you type a line without a line number into MegaBasic at the command level, MegaBasic will immediately execute it. If it is a command then MegaBasic performs that command. But you can also type a program statement or line of statements and MegaBasic will execute it immediately as if it were a command. This technique is called *direct statement execution.*

For instance you can interrupt a running program and display the contents of an array before continuing. Or you may want to use MegaBasic as an intelligent calculator by displaying complex numerical expression values. Direct execution is an important tool for debugging programs, but you can also enter any statements directly simply to experiment and learn about them. The following example illustrates how you might display an entire text file on the screen with one direct statement:

```
Open #8,"TEXT";
While input(8); Input #8,L$; Print L$; Next
```

Direct **FOR**, **WHILE** and **REPEAT** loops execute properly only when you enter the entire loop as *one line*. Direct expressions may access any built-in or user-defined functions, **GOSUB**s and procedures at any time. However if there are any syntax errors in a **DEF** statement anywhere in the program, MegaBasic reports them for you to correct before you can execute any direct statement. This is because MegaBasic performs a local initialization of your program **DEFinitions** prior to executing direct statements.

GOTOs cause a **CONTinuation** (see above) followed by a branch to the line number specified. A direct **RETURN** also **CONTinues** program execution, followed by a **RETURN** from the current subroutine level (unless the program was not **CONTinuable** within a subroutine). You can alter the contents of program variables, and such alterations carry over to **CONTinued** execution.

Before executing a direct statement, MegaBasic scans your entire program for **DEF** statements, so that it can satisfy any potential references in the direct statement to user-defined function and procedure. MegaBasic reports any errors uncovered during this process, and if there were any, terminates without executing the direct statement. Therefore, don't be surprised if an error message with a line number appears after you enter a perfectly correct direct statement that doesn't even use any functions or procedures. Because of this **DEFinition** scan, you can type new user-defined functions and procedures into your program and then immediately proceed to use them in direct statements without ever running the program. This is especially useful for quick testing of new definitions.

RUN *[<line number or command tail>]*

RUN starts program execution from scratch and can begin at the first program line or from the optional <*line*> specified. **RUN** erases any data left over from prior runs or direct statements before program execution commences. You will not usually specify the optional line number, but it can be useful when the main program has several entry points for testing or debugging purposes.

MegaBasic also lets you execute programs from the operating system command level. You do this by typing the program name on the same line that invokes MegaBasic (e.g., *BASIC PROGRAM* as a command to the operating system). Also, your program can access the portion of the operating system command that follows the name of MegaBasic. You can append additional arguments in this command string, known as the *command tail,* to pass a small amount of data to the program you are running, as in the operating system command:

```
BASIC PROGRAM DATA1 DATA2
```

To make these parameters available to the program, MegaBasic stores all characters following its name (e.g., *PROGRAM ARG1 ARG2)* into the edit buffer, which is accessible using the **EDIT$** function (Chapter 9, Section 4). Your program is responsible for extracting such input parameters from the command line when its execution begins. To

test this extraction process from MegaBasic, you can type the argument sequence in a **RUN** command (e.g., **RUN ARG1 ARG2**). When the program begins, this string will reside in the edit buffer for subsequent access. This technique is useful for passing file names to programs and for using MegaBasic programs in batch files.

Which ever workspace you are in when you give the **RUN** command becomes the main program. Prior to beginning program execution, **RUN** performs the following sequence of operations:

- The program residing in the currently selected workspace becomes the main program. **RUN** erases all data currently defined by the main program all initialized variable storage to free space. *A No Program Error* results when you type the **RUN** and the current workspace is empty.
- **RUN** *marks* all temporary workspaces *as free*, and releases any data they own. This consists of all unaltered packages brought into memory with **INCLUDE** or **ACCESS** statements.
- **RUN** preserves all packages that you **LOADed** into memory along with any local data they currently have defined, and it severs all **ACCESS** relationships between them and the current workspace. **RUN** preserves the data defined by such packages so that special purpose packages can remain available indefinitely (e.g., debugging routines or completely independent programs).
- The main program is set to permanent status (regardless of its prior status). The **DEF** statements throughout the program are all initialized and then the program begins execution.

A thorough understanding of the material presented in Chapter 10 is necessary for effective development, testing and debugging of programs spanning more than one workspace.

Ctrl-C

This is not a command, but a control key used for stopping whatever process is currently underway: a sort of a *panic button*. When **CTRL-C** is struck during execution of a program, it **STOPS** program execution like a **STOP** statement, but can be *trapped* like an error. This is useful during the debugging phase to see where execution is currently happening or to immediately terminate an erroneous program.

When a program stops for any reason (i.e., **END**, **STOP**, errors), MegaBasic selects the workspace of the program containing the line in which the stop took place. This is most convenient for debugging purposes and eliminates the need for explicitly selecting packages (via **USE**) in many instances. MegaBasic displays the current package name at the *Ready* prompt whenever it differs from the one selected at the last *Ready* prompt.

Your program can trap a **CTRL-C** interruption using an **ERRSET** statement (Chapter 6, Section 4) as a type 15 error. This provides a programmed response to a **CTRL-C**, instead of interrupting execution. Also, the **PARAM**(1) statement (Chapter 9, Section 5) can enable / disable the **CTRL-C** apparatus during program execution. Since the **CTRL-C** detection mechanism consumes all keyboard characters typed during execution, disabling **CTRL-C** is useful for both preventing user intervention and permitting *one-at-a-time* console *character* input.

When you type a **CTRL-C** at the MegaBasic command level, it aborts the current entry or command and then gives the Ready message, instead the **STOP** message. MegaBasic generates the **STOP** message to indicate the interruption of a running program. The **CTRL-C** break character provides this terminating effect only when you type it from the console keyboard. It is just another control character when entered from any other device. **PARAM**(1) also lets you use the **CTRL**-Break mechanism of **MS-DOS** to interrupt program execution *without* consuming input typed during execution.

CONT

Resumes program execution after a **CTRL-C** or programmed **STOP**. Error information functions (e.g., **ERRLINE**, **ERRMSGS**, etc.) are *not restored* and subsequently relate to the **CTRL-C** instead of to some prior error. Between the **STOP** and a subsequent **CONTinue**, you can execute direct statements without losing the ability to **CONTinue.** You can access variables and **OPEN** files with direct statements while in the command level. Regardless of what package workspace you are in when you type **CONT**, MegaBasic always switches to the workspace in which the **STOP** took place, prior to resuming execution.

You can also modify the program source to *some degree* without losing the ability to **CONTinue** execution. This is powerful during the test and debugging phase of your program development. You can insert new program lines to temporarily show certain intermediate values and computations; you can locate and correct programming errors then re-test them all during one run of the program.

CONTinuability can be lost if you modify certain key program lines. This includes program lines that called **GOSUBS**, procedures or functions that are still active, the beginning of loops and the line on which execution was interrupted. You can always determine such lines by using the **TRACE RET** command, which displays the entire active **RETURN** path. **CONTinuation** is also lost when a **REN**, **MOVE**, **DUPL** or **COPY** command changes the sequence of any program lines. MegaBasic informs you that **CONTinuation** was lost following any action on your part which blocks **CONTinuation**. However it is always safe to insert additional lines into the program without ever affecting **CONTinuability**.

When your program is in a **CONTinuable** state, you can cause a continuation by typing one of several executable direct statements, instead of the **CONT** command. These statements are given below along with a description of what they do:

GOTO	**CONTinues** execution at the program line specified by the **GOTO** statement (e.g., **GOTO** 150).
RETURN	**CONTinues** at the first statement following the most recent **GOSUB** or **PROCedure** call. If no such calls are currently active, an *Unexpected* **RETURN** *Error* results. If your program is suspended inside a function, instead of a **GOSUB** or procedure, you must also supply a **RETURN** *value* to avoid causing an error.
NEXT	**CONTinues** at the first statement following the current **FOR**, **WHILE** or **REPEAT** loop. If no loops are currently active, an *Unexpected Next Error* results.

Trace [#<device>][<line number list>]

The **TRACE** command provides an excellent environment for debugging MegaBasic programs. You can invoke the **TRACE** mode anytime you can type in the **TRACE** command. **TRACE** options are all set / reset on a workspace by workspace basis and, as with all other commands, **TRACE** commands of all types affect only the program in the current workspace. Hence with a multi-package program, you can selectively **TRACE** one package without tracing everything, a common limitation in many symbolic debugging systems.

Typing **TRACE** by itself will put subsequent program execution into singlestep mode, in which remaining unexecuted statements of the current program line are shown while program execution freezes and waits for you to type a **TRACE** control key. Typing **TRACE** with a line number will assert the single-step mode when program execution reaches the line number specified. **TRACE** control keys are then used to manipulate subsequent execution of your program and restrict what is actually traced on the screen. We will describe these shortly.

You can specify up to eight line numbers to indicate where the single-stepping should begin. With multiple line number breakpoints set, program execution proceeds normally until MegaBasic encounters a statement on any of the specified lines. When this happens, execution enters the single-step mode and MegaBasic clears all the specified breakpoints. Multiple breakpoints are useful when you want execution to break at any one of several places, but you do not know which one will be first.

You can direct the **TRACE** display to a device other than the console by specifying the device number immediately after the **TRACE** keyword. However all **TRACE** control characters are always accepted from the console keyboard (device 0). When you omit the device number from the **TRACE** command, MegaBasic uses the last device number explicitly specified by a **TRACE** command, or device #0 if no device number was ever supplied. Hence, once a device number is set, you do not need to specify it in each subsequent **TRACE** command, except to select a new device number.

MegaBasic *beeps* if you enter an unknown **TRACE** control character in the single-step mode. If you single step through a LINK statement (Chapter 10, Section 1) execution breaks at the first statement of the program (i.e., breaks on completion of segmentation or overlay statements). Once invoked, the **TRACE** mode persists until terminated with the **ESC** control or an untrapped error occurs during program execution.

Since **TRACE** mode is set independently on each package, the display generated by the **SHOW** command indicates which packages are being **TRACEd** by placing an asterisk (*) beside the package type of each package in active **TRACE** mode.

A description of each of the **TRACE** keys now follows. MegaBasic immediately acts upon each key as you type it, rather than waiting for a carriage **RETURN** as the other commands normally do.

Execution Stepping Keys	
Sp	Space-bar steps to the next program *statement*. **REPEAT** to observe *statement-by-statement* execution.
-	Step to the next program statement at the *same or higher level* as the current statement shown. A dash single-steps like the space-bar except that MegaBasic steps through **GOSUBS**, procedures, functions and loops as if they are indivisible statements, i.e., MegaBasic does not **TRACE** their internal statements.
N	Step to the next program line—the one *following* the line currently shown.
R	Step to the next invocation of the *current statement*.
^	Step to the first statement outside the current subroutine or loop (**GOSUB**, function, procedure, **FOR**, **WHILE** or **REPEAT** loops). This lets you ignore the remaining details of any loops or subroutines you happen to fall into while tracing your program.
T	Step to the statement following the next line number transfer, such as after a **GOTO**, **GOSUB**, **ERRSET** trap, etc. This lets you to skip uninteresting in-line sequences.
C	Step to the *breakpoint line*—the program line at which the **TRACE** began after a **TRACE** command, or the **TRACE** line shown when you typed a *B control* (below).
B	Marks the currently shown **TRACE** line as the new *breakpoint line* and scrolls up one line to indicate that you typed this command. After the **TRACE** has continued on to other program lines, you can step to the *breakpoint line* by typing the *C control* (above).
X	Step to the statement where the current **TRACE IF** expression becomes *TRUE*. MegaBasic executes at least one statement before re-asserting the single-step mode. **TRACE IF** is described in Chapter 2, Section 4.
Z	Step to the statement where the current **TRACE IF** expression becomes *FALSE*. MegaBasic executes at least one statement before re-asserting the single-step mode.

Trace Control Keys	
Esc	Permanently releases your program for normal, untraced execution. The only way to reinstate the **TRACE** mode is to interrupt your program with a **CTRL-C**, enter a new **TRACE** command, then **CONTinue** program execution.
Ctl	A **CTRL-C** stops the program and enters the command mode, so that you can enter commands and other direct statements. You can resume the **TRACE** mode using the **CONTinue** or terminate the **TRACE** by entering a **TRACE END** command. **ERRSET** statements (Chapter 6, Section 4) will not trap a **CTRL-C** with **TRACE** in effect.
:	Invokes your own custom debugging command that you previously setup using the **TRACE**: command, as described on Chapter 2, Section 4.

You can use **CTRL-C** to enter the command mode, execute commands or examine program variables using *direct statements*, and then resume using the **CONTinue** command. However, several **TRACE** function keys provide convenient, immediate information about the state of your running application. These are described below:

Trace InformationKeys	
A	Displaysthe active program control structures—same the **TRACE** RET display (Chapter 2, Section 4), except that the current **DATA** read pointer location is also shown.
F	Displaysthe set of currently open files, including their names, open modes, file sizes and current file positions- same as the **SHOW OPEN** command display (Chapter 2, Section 5).
S	Displaysthe names, sizes and other statistics for all the present workspaces—same as the **SHOW** command display described in Chapter 2, Section 5.
V	Displays the contents of each variable that appears in the current **TRACE** line. Both the name of the variable and its contents are shown. It displays strings in quotes with unprint-able characters shown as underscores. It evaluates subscript and indexing expressionsas needed to access array or string elements. Since this invokes user defined functions in such expressions, global function side-effects may affect subsequent program operation.

When you type V in the **TRACE** mode, MegaBasic evaluates each variable shown on the line and displays it. However, you need to be careful about array or indexed string variables that contain extended assignments or user-defined functions. For example, consider the following statement:

```
X = ARRAY(l,let J+=1)
```

Every time that ARRAY( ) is evaluated, its J subscript is incremented. This occurs both during execution and when **TRACE**d with the V option. Since this modifies the program execution state, the V option in such a case can and will interfere with subsequent program execution. Likewise, references to user-defined functions needed to resolve variable accesses can also change the execution state and interfere with later execution (i.e., by modifying *global* variables).

In general, this difficulty cannot be detected and handled by MegaBasic. The only defense against its potential interference with your program execution state is being aware of the pitfalls and avoiding the **TRACE** V option when you know it can lead to trouble.

Trace Breakpoint Pass Counts

Sometimes it is useful to delay the actual break until the breakpoint line has been reached some specific number of times. This is called the breakpoint pass count and you can specify a separate pass count on each line number breakpoint given in the **TRACE** breakpoint **LIST**, for example:

```
TRACE 100:5, 850, 2035:415
```

This **TRACE** command asserts three breakpoint lines: 100, 850 and 2035. Notice that line numbers 100 and 2035 are followed by a colon (:) and a number, i.e., the pass count. When the program is executed, it executes normally until line 100 is reached 5 times, line 850 is reached once or line 2035 is reached 415 times. Pass counts can be any value from 1 to 65535. If the line contains more than one statement, each statement executed in that line counts as one pass. For example if the **TRACE** line contains a long loop, the pass count may actually be consumed during that loop even though the line was entered only once.

TRACE *[#<device>J* IF *<logical expression>*

Defines a numeric expression that evaluates to **TRUE** (any non-zero value) or **FALSE** (zero). After you begin or continue execution, your program will run normally (i.e., not traced) until the expression becomes **TRUE** as a result of the changing program state. At that point, MegaBasic enters the single-step **TRACE** mode, so that you can control subsequent **TRACE** operations from there. You can specify a <*logical expression*> of any complexity and it may employ user-defined functions if needed. For example, to begin tracing when your program state makes the value of X equal to Y+Z, enter the following command:

```
TRACE IF X=Y+Z
```

If, after single-stepping through your program, you wish to resume untraced execution until the same logical expression is again **TRUE**, type the X command in single-step mode. To resume tracing when the condition becomes **FALSE**, type the Z **TRACE** control. If you desire a different **TRACE** condition then you have to type a **CTRL-C** to get back to the command mode, enter a new **TRACE IF** command, then **CONTinue** program execution.

Because MegaBasic evaluates the <*logical expression*> prior to executing each program statement, complex expressions will slow down your program execution by some slight but noticeable amount. MegaBasic reports errors in the <*logical expression*> as errors in the current program statement.

TRACE:<*executable line of statements*>

Stores an arbitrary direct statement for later retrieval and execution during the single-step **TRACE** mode. Once set up, you can execute this direct statement from the single-step mode merely by pressing colon (:), in place of one of the other **TRACE** controls. One common application for this is the display of various program variable values and intermediate results while you are **TRACing** the program. In such a case, you might type a command such as:

```
TRACE:  Print X,Y,Z, I,A$, I,B$, I,C$
```

After this command you could **RUN** or **CONTinue** your program with the **TRACE** mode in effect. From the single-step mode, you can execute the **PRINT** statement shown above by simply typing a colon (:). If one executable line is not sufficient for your purposes, define your own debugging subroutines, place them into your main program or a separate shared workspace, then call these subroutines from the **TRACE**: execution line. This can save you a great deal of debugging time, since it lets you access custom debugging procedures at the touch of a button. Additional keyboard input may be taken to select one debugging action from a set of multiple choices; the possibilities are endless.

TRACERET

This command displays the **RETURN** path active at the time that the program stopped (e.g., **CTRL-C**, **STOP**, program error, etc.). The first line number shown will be the point at which the program stopped. The **RETURN** path goes all the way back to the first subroutine call made from the main program.

MegaBasic describes each **RETURN** location with the type of **RETURN** (**GOSUB**, function, procedure, etc.) and the line number and subroutine name to which it **RETURNs**. Each

description is shown on a separate line and since this can potentially be quite lengthy, you can use any of the display-pause controls available under the **LIST** command (i.e., Space-bar, carriage **RETURN**, **CTRL-C**). **TRACE RET** operates only in the command mode and has no effect on the dynamic **TRACE** mode if set.

In addition to the subroutine **RETURN** information, **TRACE RET** also displays all active **FOR**, **WHILE** and **REPEAT** loops and **CASE** statement blocks, along with the line number range they span, and all the nested **ERRSET** error traps levels that have been set along the way. If you suspect that a loop is not terminated where you think it should be, you can **STOP** the program inside the loop (e.g., by inserting a **STOP** statement inside it) and then give the **TRACE RET** command to show its current active line number range.

You can also get the **TRACE RET** display from the single-step mode (i.e., without entering the command mode) by typing the **TRACE** A key.

TRACEEND

Terminates the **TRACE** mode from the command level. You can also terminate the **TRACE** by typing the **ESC** key when you are *single-stepping*. Program execution proceeds without further interruption after turning off the **TRACE** and **CONTinue** execution.

CHECK [#<output device>]

To permit you to **RUN** and test partially complete programs, MegaBasic does not in any way check the syntax of a program and insist that it be error-free before running it. Instead, MegaBasic provides a **CHECK** command for you to use whenever you wish to check common coding errors. This command provides the following checks:

- Reports syntax errors in **DEF** statements. Since **DEF** statements provide information vital to other **CHECKing** activities, such errors terminate the **CHECK** process. After you correct all **DEF** statement errors, **CHECK** will be able to complete the rest of its analysis.
- Verifies all line number and line label references throughout the program to ensure their target line actually exists. The errors found by this check may also include certain references to procedures defined in other packages, so be aware of this when **CHECK** reports an *Undefined Name Error*.
- Checks for proper nesting and termination of **CASE** blocks and **FOR**, **WHILE** and **REPEAT** loops, makes sure that they do not cross any **FUNC** or **PROC** definition boundaries. **CHECK** reports errors for incorrect loop index variables, for encountering a **NEXT** or **CASE** not part of any preceding structure, or missing a necessary **BEGIN** or **END** on a **CASE** statement.
- Verifies that **THEN** and **ELSE** clauses of single and multi-line **IF** statements are nested properly and have left and right brackets properly balanced and present in the right number. **CHECK** examines all expression parenthesis pairs to ensure that each one is properly balanced.
- Reports an user-defined function, procedure and line label names used in the wrong context. For example, functions cannot appear at the beginning of a statement; conversely procedure names must appear in front of the statement; line labels can only appear in **GOTOS**, **GOSUBS**, **ERRSETS**, etc. However, **CHECK** cannot usually determine the correctness of undefined procedure, function and variable names because their definitions may not be available until you execute the program (because of external packages and unexecuted **DIM** statements).

- **CHECK** reports all errors found throughout the program all at once, regardless of how many that might be (except for **DEFS**). However, it reports only the first error of several on a line; you must correct the first error on the line before **CHECK** will report any other errors it contains. You can control the display of long lists of error messages with the pause-step-start keys as defined for the **LIST** command. You cannot restrict the **CHECK** operation to only a partial set of program lines.

You can redirect the error report to any device or open file by specifying the open file / device number (#1 is the printer). Omitting the device number is the same as specifying #0, which outputs to the console screen. You can type a **CTRL-C** to abort the error report at any time, and you can control the report output using the same pause keys that the **LIST** command supports.

The **CHECK** command does not perform a complete, exhaustive analysis of program syntax, but merely locates some of the more obvious errors in program formation. Because of its simplicity, **CHECK** can report errors that may not actually exist, particularly bracketed [1 constructs that span multiple lines (resulting in a mistaken *missing or unexpected bracket error).* Bear in mind many program constructs cannot be verified without actually executing them within their program context, hence a 100% syntax checker is beyond the scope of the MegaBasic development system. For an exhaustive 100% **CHECK** on your program, compile it under the MegaBasic compiler which also verifies all data types and argument **LISTs** as well. Remember to that errors can easily occur within syntactically perfect programs, a problem that all programmers must contend with when using any language.

CHECK [#<output *device>]* LIST

Just like **CHECK**, except that it displays the program source lines that contain the reported errors.

CHECK [#<output *device>]* EDIT

Just like **CHECK LIST**, except you can edit each erroneous line as **CHECK** finds them. After you edit a line and type a carriage return, MegaBasic rechecks the line and reports any additional errors found before moving on to the next line. To skip a line without correcting it, just type a carriage return in response to the line presented for editing.

Section 5: Information and Control Commands

Described in this section are the informative commands that provide displays of useful information on your programs in memory, statistics about current resource utilization, state of program execution and current environment. It also discusses how to be in several *copies* of MegaBasic simultaneously, and how to get out of MegaBasic when finished with everything you are using it for**.** A quick summary follows below:

BYE	Exits MegaBasic and goes back to the host operating system command level. **BYE** also exits a nested **BASIC** environment (see the **BASIC** command below) and goes back to the prior environment.
STAT	Displays a variety of useful information about your program, its execution state, and the supporting resources maintained for general use.
SHOW	Displays all currently defined workspaces by name along with information about their content. It can also show the shared access relationships between the current work space and all the others, information about all currently open files and sizes of currently defined arrays and strings.
USE	Selects other workspaces by name for subsequent operations and creates new workspaces for program entry. It can also continuously cycle through all workspace names so that you can select one without having to type its name.
XREF	Displaysa cross-reference index report for the program contained in current workspace.
CLEAR	Deletes the current workspace and its contents, or only the variables currently defined. It can optionally delete all workspaces or release all memory in use by program variables.
BASIC	Enters an independent nested environment for developing, testing, debugging or running other programs while temporarily suspending the current work underway.

BYE

Terminates MegaBasic and exits back to the operating system command level. Prior to exiting, MegaBasic will request confirmation for any workspaces containing original work that you have not yet saved on a file. If you previously invoked a **BASIC** command (see below) **BYE** will exit from that *instance* of MegaBasic and **RETURN** to the prior copy of MegaBasic. **BYE** is equivalent to **DOS** without any arguments.

STAT *[#<device>]*

Displays various sizes, states and other statistics about the current program and working environment. The display is divided into two groups. First the overall global resources are shown, which are then followed by statistics about the current program and execution state. Display contents may change from one MegaBasic version to the next, but they will generally cover the following topics:

- Overall number of memory bytes allocated to current processes Total memory remaining, including memory allocated to freed packages Amount of space remaining for evaluating expressions Total number of active named objects (variables, functions, etc.) File buffer counts and space remaining on the default drive
- Current workspace name and workspace count (if more,than one) Size of the current program and size of its data (if any) Various statistics about the current execution state States of various debugging and internal parameter settings.

SHOW *[#<device>]*

The **SHOW** command displays a one line description of the contents in each workspace. Each description includes the program name, the workspace contents type, the package execution usage, package access count, the program size and how much data it currently has initialized. The possible *workspace types* and *execution usages* are described below:

Workspace Contents Types	
Keep	Package kept in memory until **CLEARed** (by virtue of being **LOADed**
Work	Contains unsaved program modifications
List	Listable package temporarily brought in by an **ACCESS** or **INCLUDE**
Binary	Assembler package
Hidden	Unlistable package (i.e., scrambled by the **CRUNCH** utility)
Empty	No program lines (deleted if not the current workspace)

Package Execution Usage	
Main	The *Main* program is the first program loaded, or the most recent program selected when the **RUN** command was typed, or the most recently **LINKed** (i.e., **CHAINed**) program during execution.
Free	Completely unused and releasable as free memory as needed.
Uninit	Uninitialized package (no data, no **ACCESSes**, etc.).
Access	Active, **ACCEssible** package in a running application.
Detach	Active package unreachable through any **ACCESS** path from the main program, e.g., an **INCLUDED** package without being **ACCESSed** (see below).
Trans	Active package in a transient state between being *detached* and *accessed*.

Packages that are initialized with data and active for use, but are not accessible from the main program, directly or indirectly (i.e., not on any **ACCESS** path from the main program), are classified as *Detached*. This helps you see packages that are left *floating* without any apparent use, but that otherwise remain active and initialized, consuming memory until they are **DISMISSed** from all packages. For example, if two *Detached* packages **ACCESS** each other, their *epilogues will* not be executed until some other package **DISMISSes** them from each other. This is because *epilogues* are not executed until all **ACCESSes** to them have been **DISMISSed**. Packages that are merely **INCLUDED**, rather than **ACCESSed**, are another example of a *Detached* package.

Because of the potentially large number of packages that applications may keep in memory, **SHOW** commands display the package names in alphabetically sorted order. The currently selected workspace is marked with an arrow (>) in front of the name.

To assist debugging efforts, the **SHOW** listing places an asterisk (*) beside the program type of programs with **TRACE** mode active.

SHOW *[#<device>,]* ACCESS [*]

Displays the **ACCESS** relationships currently in effect from all prior **ACCESS** statements. Two viewpoints are shown: All workspaces *accessible from* the current workspace, and all those which have *access to* this workspace.

To see this display for *all* packages in the active application, specify the optional asterisk (*) at the end of the **SHOW ACCESS** command. Packages without any **ACCESS**

relationships with other packages are not shown. You can specify an optional output device number to redirect the output. The **ACCESS** statement is described in Chapter 10.

SHOW *[#<device>,]* OPEN

Displays information about each file currently OPEN, including the OPEN file number its read / write attributes set at **OPEN** time, the file name and drive, its shared / private access attributes, its current byte size and the position of the current read / write pointer. This display is useful when testing and debugging programs which **OPEN** and process files. You can specify the optional device number to redirect the output of this command to a device other than the console (device #0) or to an open text file.

SHOW [#<device>,] SIZE [<selector list>]

Displays the memory size allocated to active *arrays and strings* and totals for both all *numeric scalar variables* and all *pseudo variables* (only as totals, not individually by name). Variables that have not yet been defined are not shown. The listing is ordered alphabetically and includes the number of memory bytes allocated, the name of the variable and its type. The sum of the sizes shown is displayed at the end of the listing.

You can restrict the listing to only one variable type by including the desired type in the command (e.g., **SHOW SIZE REAL**, **SHOW SIZE STRING**, etc.) You may specify any of the type names supported by the **NAMES** commands, but only string or array variables will be listed by **SHOW SIZE**. See the **NAMES** command for a description of this optional <*selector* **list**>. As with the other **SHOW** options, you can include a device number to redirect the listing somewhere other than the console if you so desire.

USE *[<workspace name>]*

Selects a workspace for subsequent operations. Omitting the workspace name from the **USE** command enables you to switch from package to package until you reach the one you desire. You can control this with single keystrokes that perform the following actions:

Space, → ***or*** ↓	Sequences forward to the next workspace name in *ascending* alphabetical order. After the last one, it cycles back to the first workspace name again.
Backsp, ← ***or*** ↑	Sequences backward to the next workspace name in *descending* alphabetical order. After the first one, it cycles back to the last workspace name again.
Home	Goes to the lowest workspace name in sequence.
End	Goes to the highest workspace name in sequence.
Tab	Sequences forward to the next workspace name that was explicitly **LOADed** (i.e., skipping packages temporarily loaded by the applications).
Character	Skips to the next workspace whose name begins with the specified character.
Enter	Selects the workspace name currently shown as the current workspace.
Ctrl-C ***or*** **ESC**	Aborts the **USE** command without changing workspaces.

When you specify a workspace name in the **USE** command, MegaBasic looks for it among the current workspaces defined. If it finds it among those present in memory, MegaBasic selects that workspace. If it does not find it, MegaBasic creates a new workspace with the name given, subject to user confirmation, then selects it as the current workspace.

To minimize the number of unnecessary workspaces in memory at any given time, MegaBasic automatically deletes workspaces that contain no program lines. This action is taken only when you leave the empty workspace for another. Hence you cannot create several empty workspaces and then go back to fill them in: you have to use them immediately. Temporary workspaces that have been **DISMISSed** and represent free memory are skipped by **USE**, but you can switch to them by specifying their name.

XREF *[#<device>,][<line range>][,<selectors>][by <mode>]*

Provides you with an instant cross-reference of all user-defined procedures, functions, variables, **GOTO'S**, and other line referencing used in your program. It displays each name, label or line number followed by a list of all program locations that refer to each (by line number or by subroutine name). **XREF** indicates references where the name is **DEFined** or **DIMensioned** with an asterisk. **XREF** commands may include four optional parameters: an <*output device number*>, a <*line range*>, a <*selector list*> and a *by* <*mode*> as summarized below:

<device>	Specifies the device number to which the cross-reference listing is sent. If you omit the <*device*>, then **XREF** displays its report on the console. If the device specified is not the console (device #0), MegaBasic inserts page breaks into the **XREF** report using form-feed characters (an *ASCII 12)* at appropriate places.
<line range>	Specifies an optional line range to restrict the **XREF** report to only names and line numbers referenced at least once within the line range. This tells you where *all references* to anything within this range are found throughout the program. Omitting the <*line range*> implies the entire program.
<selectors>	Specifies a list of attributes that selects the kinds of objects that MegaBasic includes in the cross, reference listing. The selector list is identical to the selector list used in the **NAMES** command described on Chapter 2, Section 3.
by <mode>	Selects the how the references will be shown: by *line* shows line numbers, *by name* shows subroutine names. When omitted, **XREF** defaults to the most recent <*mode*> specified or to *by line* if no previous **XREF** requested.

XREF reports on the program maintained in the current workspace. Hence to generate an **XREF** report, you must first **LOAD** your program into a workspace. Then type **XREF** followed by any desired device number and / or line range and terminate with a carriage **return**. **XREF** immediately begins generating its report as specified. To pause the display (especially on the console screen), you can use any of the stop / start / step controls.

You can use **XREF** from the MegaBasic command level at any time. It does not use any working storage, nor does it affect the contents of variables for a temporarily suspended (**CONTinuable**) program. This makes it suitable for use even during a debugging session.

You can restrict the cross-reference produced by **XREF** to names having certain *specified attributes.* This lets you produce a cross-reference listing of, for example, only the procedures, or only the string functions, or only the line labels and line numbers, to quickly answer questions about your program under development without having to **XREF** the entire program, which could take a while for a large program. You do this by listing the desired attributes as arguments to the **XREF** command in the same manner as in the **NAMES** command explained in Chapter 2, Section 3 . For example, **XREF FUNC** will cross-reference all the user-defined function names; **XREF STRING FUNC** will cross-reference only those functions which return a string result. These attribute selectors control which of the following 12 breakdowns are included in the listing:

Procedures	Linelabels	Line refs
Real Functions	Real Variables	Real Fields
String Functions	String Variables	String Fields
Integer Functions	Integer Variables	Integer Fields

You can type any combination of selectors, separated by spaces, in any order after the **XREF** keyword. **XREF** displays only those names that satisfy all the selectors specified. See the discussion in the manual on the **NAMES** command for complete information about how to use and specify attribute selectors.

XREF assumes the data type *currently defined* for each name, which may not always be accurate if the program has not been **run**. For example, it shows references to **SHARED** functions and procedures defined in other packages as *variables* unless prior program execution has already defined them and bound them to the references in the current program. Although fields appearing in **DEF STRUCT** statements are always shown in **XREF LISTings**, those defined in regular *executable* **STRUCT** statements are shown as ordinary *variables* unless those **STRUCT** statements have been executed (i.e., by running your program before your **XREF** listing).

Finally, you can display the references either *by line number* or by *subroutine name,* by appending either **BY LINE** or **BY NAME** to any **XREF** command. If you omit the **BY-suffix**, **XREF** displays in the same mode it did the last time you entered an **XREF** command, or *by-line* if no previous **XREF** command was typed. For example the command **XREF INTEGER BY NAME** displays the names of subroutines that refer to each integer variable or integer function. Program references that are *not within* procedures or functions (i.e.,they are in the main program, the prologue or epilogue, or in between subroutines) are shown in the **XREF** display as referenced by <main>. Multiple references to a name within the same subroutine or line always show up in the **XREF** listing as a single reference.

CLEAR

Deletes the program within the current workspace and then eliminates the workspace altogether (unless it is the sole workspace). Afterward, MegaBasic switches to the next workspace in the *LOAD* sequence. MegaBasic asks you if you want to clear all workspaces or just the current workspace, to which you can answer yes or no.

CLEARDATA

Deletes all data currently defined within the selected workspace (variables, control structures, etc.) and releases the memory resources allocated for them back for reuse. Program **CONTinuation** is not possible after invoking this command. You can use

CLEAR DATA to prepare for a series of direct statements which are known to require more memory resources than otherwise available. The **RUN** command does an implicit *CLEAR DATA* at program initialization time, as does the *LOAD* command if there is not sufficient memory left to load a program.

CLEARFREE

Deletes every program that is no longer in use. The *SHOW* command displays such programs marked as *FREE*. MegaBasic normally deletes these programs only when it needs the memory they occupy for other operations. Hence the only real reason for using *CLEAR FREE* is to eliminate the extra clutter brought about by unneeded workspaces left by prior program testing. *CLEAR FREE is* extremely conservative about what it deletes, and, in particular, it will never remove anything containing *unsaved revisions* or alter the execution state of a program in progress.

BASIC *[<program command tail>]*

Provides a completely separate *copy* of MegaBasic in which other activities may be independently performed. You can invoke the *BASIC* command at any time in the command level to instantly provide a sub-environment in which to run / develop programs. This environment is completely isolated from the environment set up by your prior invocation of *BASIC*, and hence you may perform any sequence of operations you wish without fear of altering higher level environments. To return to the environment from which you re-entered **BASIC**, simply type **BYE** or **DOS**. Such a return also occurs if a a program running in a sub-environment executes a **DOS** statement. MegaBasic frees all resources held by the sub-environment upon returning. All *parent* environments are totally preserved right down to the current state of execution **CONTinuation**.

You can type an optional *command tail* on a **BASIC** command to run a program just as if you were doing it from the operating system level. MegaBasic loads the program, executes it and provides the remainder of the command tail to the program for subsequent access. If the program terminates via a **DOS** statement, it exits the sub-environment and you will be back in the previous environment. Otherwise you will remain in the new environment until you type a *BYE* command. At least 16K bytes of free space must be available in order to invoke *BASIC*.

MegaBasic will provide all currently unallocated memory to the new sub-environment. To this end, MegaBasic removes all programs not currently active or in use, and releases the memory they occupy. The **SHOW** command shows such programs as **FREE** (before MegaBasic releases them).

Chapter 3

Representing and Manipulating Numbers

MegaBasic supports two fundamentally different data representations: numbers and strings. Chapter 3, Section 4 describes strings and how to represent and manipulate them in your programs. Section 3 of this chapter describes in depth the concepts and use of numeric constants, variables, arrays, expressions, operators, functions, vector processing and floating point systems. It is organized into sections of increasing complexity, and each depends somewhat upon understanding those that precede it:

Representing Numbers	Introduction to representing numbers within a computer and choosing the most appropriate numeric representation for solving problems.
Constants	Representing fixed numeric quantities.
Simple Variables	Representing single-value numeric quantities that can change during program execution.
Array Variables	Ordered sets of variable numeric quantities, organized in one or more dimensions.
Operators & Expressions	Mathematical phrases for combining numeric quantities into computed results.
Numeric Functions	User-defined and built-in symbols for combining and transforming data.
Vector Processing	Processing entire arrays and array cross-sections using vector arithmetic expressions and a variety of vector statements.
IEEE Floating Point	Detailed description of the trade-offs between **IEEE** and **BCD** floating point versions of MegaBasic. Topics include 80x87 math coprocessor support along with speed comparisons.

Section 1: Representing Numbers

Numbers are fundamental to all computer applications. Even applications that appear non-numeric, such as graphics and text processing and language translation, are intensely arithmetic beneath the surface. Computers have evolved beyond their early dedication to engineering and equation solving, into tools of creativity and thought expansion, and yet they still thrive best in the medium of numbers and arithmetic.

When we attempt to classify numerical applications, an important distinction can be made between applications involving whole numbers, i.e., numbers without any decimals, and those applications where fractional quantities arise. For example, counting applications usually involve only the whole numbers, while scientific and financial applications are built upon fractions of time, dollars or other physical units. This distinction is important because microcomputers can deal with whole numbers much more efficiently than with fractional quantities. Hence, MegaBasic supports two different internal representations of numbers, one exclusively for whole numbers, called integer representation, and one for general numeric values (including whole numbers and fractions) called floating point (or real number) representation.

All programs could be written with floating point representation exclusively. However, if a program spends much of its effort performing essentially integer arithmetic, its performance could significantly benefit by utilizing the more efficient integer representation and operations wherever possible. The primary reason for supporting integers in a computer language is that integer arithmetic is much faster than floating point arithmetic and integer values use less memory. In the paragraphs that follow, we will examine the strengths and weaknesses of both number representations *(real* and *integer),* as well as how and why to choose one form over the other.

Floating Point Representation

Ideally, a computer should be able to deal with numbers of any size, no matter how great or how small and possess absolute precision with thousands of decimal places. However, even thousands of decimals cannot represent Sqrt(2) or pi exactly, and if they could, they would devour all your memory resources. How wide a range and how much precision do you really need? Since the answer to this question depends on you and the problems you have to solve, MegaBasic supports a two types of floating point representation with a variety of precisions.

Commercial or **BCD** MegaBasic, represents floating point numbers in **BCD** floating point format. This format can represent numbers 10^{63} or as small as 10^{-63}. Such a range encompasses nearly all quantities ever arising from physical phenomena, sub-atomic to cosmic. Scientific or **IEEE** MegaBasic represents floating point numbers in **IEEE** double precision binary format. This format can represent an even wider numeric range than **BCD** format: as large as 10^{307} power or as small as 10^{-307} power.

This range is called the dynamic range of floating point numbers. If you perform a calculation that exceeds this range, a numeric overflow error will occur, stopping your program (a trappable error, however). This can easily occur from multiplying very large numbers (e.g.,$10^{35} * 10^{40}$), or from dividing a large number by a very small number (e.g., $10^{30} / 10^{-50}$). If your application has the potential for producing such errors, you must provide error checking, error traps or other measures to ensure your program remains in control after such errors occur.

Another property of floating point representation is its precision, or the maximum number of decimals it can hold. **BCD** MegaBasic comes in any one of several precisions, from 8 to 18 decimal digits (14-digit is standard), while **IEEE** MegaBasic provides 16-digit precision only. Any particular copy of MegaBasic supports only one precision of floating point. If a number contains more decimal places than the precision of the floating point representation, MegaBasic keeps only the upper, most significant digits that fit and discards the rest. For example, 8-digit precision would represent the number 534.666666682 as 534.66667 (rounded). You would have to use precisions of 12 or more digits to represent this number exactly in the machine. If precision is important in your applications, be sure to use a version of MegaBasic that supports a precision of sufficient size.

The number of memory bytes required by each floating point value depends on the prevailing precision. **IEEE** floating point numbers always require 8 memory bytes for each value. If P represents the number of **BCD** digits precision, then the number of bytes required is given by the expression: 1+P / 2. You can find out the actual floating point format provided by the running copy of MegaBasic from the **PARAM(4)** function.

BCD representation, which stands for *Binary Coded Decimal*, internally represents numbers in base–10, while **IEEE** binary floating point uses base–2. **BCD** floating point has two important advantages over binary floating point. First, software conversion between **BCD** and ASCII display codes and back again is very efficient. Second, and most importantly, **BCD** represents all decimal numbers within its maximum precision *exactly*, without any round off or truncation errors. For example **BCD** represents the dollar figure $24.95 exactly, while in binary floating point it would look something like $24.95000000001 or $24.94999999999. This makes **BCD** particularly useful in financial applications, where all data is typically in decimal form and round-off errors are unacceptable.

No matter what number base you use to represent numbers, there is always some number that it cannot represent exactly. For example neither **BCD** nor binary floating point can represent the fraction 2 / 3 exactly. But a base–3 numeric representation *could* represent it exactly in only one digit (i.e., as .2). **BCD** floating point requires about 12% more memory than a binary representation with similar precision, but the advantages of decimal arithmetic in many applications sometimes outweighs this disadvantage.

The prime advantage that **IEEE** binary floating point has over **BCD** is raw speed. Even without a math chip, **IEEE** arithmetic is considerably faster than **BCD** arithmetic. For more speed, particularly with transcendental functions, **IEEE** MegaBasic automatically supports an *In tel 80x87* math coprocessor if the host machine contains one. However, it doesn't make much difference whether add and subtract are done in **BCD**, **IEEE** software or in *80x87* hardware: they all take around the same amount of time. See Chapter 4 for further details on the trade-offs between **IEEE** and **BCD** floating point.

The computer language of **BASIC** originally supported numbers exclusively in floating point to simplify its implementation and user interface within an educational environment. To improve data processing efficiency, MegaBasic also includes an integer data type. Of course, this means that you have to choose one representation over the other for each number specified. Fortunately, this is easy because MegaBasic automatically chooses floating point whenever you fail to explicitly choose integer representation. Later on we will cover numeric type selection in detail.

You should select floating point representation over integer for values used in a floating point operation, or for numbers with decimals, or for values outside the range of permissible integer values (barring an integer representation). Avoid using floating point

values in array *subscripts* and string *indexing* (described later), or other places where integer representation would suffice. If you do, there is no harm, but your program will simply run more slowly than it could have with the proper integer declarations and definitions within your program.

Many programs written using a **BASIC** that supports only floating point representation usually contain portions which would run much faster using integers. You can usually improve the performance of such programs by modifying them to take full advantage of integer representation without much work. Appendix D, Section 4 contains a *step-by-step* procedure that you can follow to convert such programs.

Integer Representation

MegaBasic can represent *whole numbers* in 32-bit binary integer representation as well as in floating point representation. Integer representation of numbers is important in three ways. First, integer arithmetic is many times faster than identical arithmetic performed in floating point operations. Second, since the vast majority of numeric applications require binary integers for loop counters, array subscripts and indexed string locations, using numbers already represented in binary form eliminates the time consuming job of converting floating point representation into binary representation (which MegaBasic does automatically).

Third, integer representation is physically more compact than floating point. Integers require four bytes per value while floating point requires 8 bytes per value (although it varies from 5 to 10 bytes depending on the floating point precision). This allows larger integer arrays with the same memory requirements as smaller floating point arrays. Also, since most programs spend a great deal of time just moving numbers from one place to another, a more compact numeric representation can also increase program performance.

MegaBasic integers are more powerful than integers of many other microcomputer languages because of its internal representation. MegaBasic represents integers internally in what's known as a 32-bit twos-complement signed integer, while some systems use only a 16-bit version of the same thing. This will represent exactly all integers in the range from minus 2,147,483,648 up to plus 2,147,483,647, instead of –32768 to +32767 with only 16 bits. With integers of this size, many applications which would normally have to use floating point can easily use MegaBasic integers. For example integers can represent dollar figures up to $21 million exactly with MegaBasic integers (in pennies instead of dollar units).

Virtually all programming systems terminate with a fatal error whenever an integer calculation produces, even temporarily, an integer value beyond the range of values that the prevailing integer format can represent. Although the large range of 32-bit integers diminishes this problem somewhat, it can still arise. MegaBasic solves this problem by automatically detecting integer overflows while performing integer calculations and then converting the integers to floating point to complete the intended operation. The burden of detecting and recovering from this type of error is not a concern of the programmer, since MegaBasic handles it automatically. Integer overflows in MegaBasic can only occur when you try to use a value larger than a valid integer when only an integer will suffice. For example, attempting to store such a value into an integer variable will result in a numeric overflow error simply because it is not possible. This automatic recovery from integer overflow is only supported by the MegaBasic *interpreter;* it is not feasible to support under the MegaBasic *compiler.*

Numeric Type Declarations

To maintain compatibility with programs written under the standard assumption in **BASIC** that *all* variables are real, all variables *are real unless you specify otherwise. You* specify integers either by declaring the leading letters in names as integer (or real), or by declaring specific names as integer (or real). To see at any time what is integer and what is real, the **NAMES INTEGER** and **NAMES REAL** commands will show you (Chapter 2, Section 3). The rules and syntax for type declarations are summarized in order of decreasing precedence as follows:

Data Type Rules for Variables & Functions

- Any variable or function name that ends with a percent sign (%) will always be *integer.* A type error occurs if you declare or **DIMension** any variable or function with such a name as real. Similarly a dollar sign ($) and an exclamation mark (!) can appear only as the last character of *string* and *real* names, respectively. This rule overrides all the other rules that follow.
- You can declare numeric arrays directly in **DIMension** statements, as shown in the following example:

  ```
  Dim integer C(30,40), X(N), real L(N), ARRAY(10,10)
  ```

  which declares C() and X() as integer arrays, L() and ARRAY() as real arrays. The reserved words **INTEGER** and **REAL** cause all **DIMension** specifications that follow in the list to be integer or real variables, or until a following **REAL** or **INTEGER** specifier appears in the list. In the same way, the word **STRING** declares string variables in DIM statements.
- You can declare specific names of variables and functions as **INTEGER** or **REAL** using **DEF** statements such as:

  ```
  DEF INTEGER X,VBL(),P
  DEF INTEGER FUNC TOTAL(V1,V2)
  DEF REAL A, ARRAY(), C1
  DEF REAL FUNC SUM(V3,V4)
  DEF STRING LINE(), MSG
  DEF STRING FUNC UCASE(BUF$)
  ```

  The empty parentheses () in the above **DEF** statements indicate names which you intend to be arrays. These specific declarations override any types specified by letter. A *Double-Definition Error* results from declaring the same name as having two different types. You must declare variable types explicitly to make them different from the type implied by its leading letter.
- You can declare leading letters of identifiers as **INTEGER**, **REAL** or **STRING**. A variable or function name that begins with a declared letter will be become an object of the type declared. Use a **DEF** statement to declare letter types, as illustrated below:

  ```
  DEF INTEGER "a, b, c, i–n"
  ```

  where the string constant *"a,b,c,i–n"* specifies the leading letters of integer variables and functions. The quotes are required, but commas and spaces within the quotes are entirely optional. you can use upper and lower case letters for the same effect. Variable and function names beginning with letters left undeclared will be *real by default. You* can similarly declare letters as **REAL** or as **STRING**. A double definition error will occur if you attempt to explicitly declare the same letter as both **REAL** and **INTEGER** (in two separate **DEF** statements).
- If none of the above rules apply, then by default, MegaBasic creates the numeric variable or function as *real.*

Section 2: Numeric Constants

Numeric constants are the most obvious way to express numbers. Examples are: –1, 5675261, 4.536, 0, –11.111, 00934.2, etc. Constants may be signed or unsigned, but MegaBasic treats constants like 1,435 as two separate numbers. The smallest numeric value permitted in MegaBasic is 10^–63 using **BCD** and 10^–307 using **IEEE** floating point versions. Arithmetic operations producing smaller numbers than this always result in zero (i.e., underflow produces a zero result). Constants must not contain any spaces or commas within them: because such characters are used to separate numbers, they would break a constant into multiple constants.

MegaBasic accepts a broad range of numeric notations which includes integers, fixed-point, floating point and scientific notation for decimal numbers. It also supports signed and unsigned integers in binary (base 2), octal (base 8) and hexadecimal (base 16). These various forms are discussed below:

Numeric Notation

To specify ordinary decimal and integer constants, simply type their values with whatever signs, digits and decimal are appropriate to the number desired. You can include more than on sign in front of a number, but this is always redundant. For example the following constants all have the same value: 99, +99, –99, +–+99. If the number of digits exceeds the floating point precision, then MegaBasic rounds the value to the nearest value fitting that precision. See the discussion about precision earlier in Chapter 3, Section 1 for further details about this.

Numeric constants may have, at most, one decimal point. As stated earlier, no spaces, commas or other non-numeric characters can appear within numbers. However, you can precede or follow any constant with one or more spaces for the purpose of improving readability or for separating the constant from other surrounding typed objects.

Exponential Notation

You can also specify numbers in so-called E-notation. Similar to scientific notation, this format includes a scaling factor to indicate a power of ten multiplier. For example, 23.4104E–2 and .234104 are identical values with the first in E-notation. The E-XX portion of the number specifies how far and what direction to shift the decimal place (+ for right and – for left). This representation becomes important when you specify extremely large or small constants. For example the constants –.20152E+42 and 3.3142E–19 would be too unwieldy and confusing with all the zeros needed to represent them in standard notation.

Whatever the exponent portion of the constant is, the net magnitude of the number must fall within the dynamic range for floating point numbers: lE–63 to lE+63 for **BCD** and lE–307 to lE+307 for **IEEE** floating point. Constants smaller than the lower limit evaluate to zero, while MegaBasic rejects constants beyond the upper limit and reports an *Out Of Bounds Error.* If the exponent is a positive power of ten, the plus sign (+) is optional. For example the constants 25E+17 and 25E17 are identical.

Binary, Octal and Hexadecimal Constants

The decimal number system (i.e., base 10) is certainly the most common notation used for expressing numbers, but other number bases can be more appropriate in certain applications. For example, applications involving bit-strings are greatly simplified when you employ binary notation to express numeric constants (i.e., in base 2). MegaBasic accepts numbers expressed in binary, octal, hexadecimal and decimal, wherever a number is expected. To specify a constant in a non-decimal number bases, you must abide by the following rules:

- You cannot specify non-decimal constants with E-notation or decimal points. They can only be positive and negative integers.
- The last character of the constant must be one of several special letters that identify the intended number base: letters H, *B and O* identify Hexadecimal, Binary and Octal constants, respectively. Upper or lower case can be used but lower case is more readable. Decimal is assumed for numbers that do not end with these letters.
- The constant must contain only those digits that are legal for the number base used. Binary numbers can contain only the digits 0 *and 1.* Octal numbers can contain the digits 0 *to 7.* Hexadecimal constants use digits 0 *to 9 and letters A to F.* Hexadecimal constants must begin with a digit (0-9).
- The range of values for the unsigned portion of integer constants is the same for all number bases: 0 to 2147483647 (decimal), corresponding to: *Oh to 7FFFFFFFh* (hex), *Oo to 177777777770* (octal) or *Ob to 11111111111111111111111111111111b* (binary).
- If the highest bit of its 32-bit representation is set to one, then the integer will be negative. For example OFFFFFFFFh and 37777777777o both represent the value –1 (although using signs is more obvious).

Program Constants

Constants within programs represent fixed quantities for use in computations. MegaBasic stores program constants in an internal table for fast access by your program. If you specify the constant in E-notation, or it contains a decimal point, or it is too large to fit integer representation, MegaBasic considers it a real (floating point) constant and physically stores it in floating point representation exclusively.

MegaBasic stores constants in both floating point and integer representations if you specified them as integers (i.e., no decimal points or E-notation) and they lie in the range of 32-bit signed integers. This lets MegaBasic choose the most appropriate numeric representation for the context of each expression providing the fastest possible access lo the constant. For example MegaBasic accesses the constant 7243.0 exclusively in floating point mode regardless of its surrounding context, but typed as 7243 (without the decimal), MegaBasic accesses it as an integer or a real depending on the numeric context.

Input Constants

You can enter constants from the keyboard in response to requests from the computer as directed by the program. When MegaBasic **`INPUTS`** constants into floating point variables, you can enter any number representable in MegaBasic floating point. However if you specify integer variables in an **`INPUT`** statement, you must enter numbers without any decimals to the right of the decimal point and they must fall within the range of 32-bit signed integer representation. The **`INPUT`** statement (Chapter 7, Section 1) checks for this and rejects constants that are inappropriate for the variable specified to receive the value.

Section 3: Numeric Variables

As in most other programming languages, numeric variables in MegaBasic provide the means for storing numbers for later access. Variables represent numbers just as constants do but with one big difference: they represent quantities that can change during program execution. See Chapter 5, Section 2 for details on storing different values into variables with assignment statements.

You identify numeric variables in your programs by a name spelled with one or more characters. The first character must be a letter (A-Z) and subsequent characters must be letters (A-Z), digits (0-9) or underscores (_). The following examples show how and how not to spell numeric variable names:

***Legal* Numeric VariableNames**	**X, X2%, COUNTER, AMOUNT!, T68, LONG_VBL_NAME**
***Illegal* Numeric VariableNames**	**4X, R$,_NAME, BAD @ NAME, INTVAL#, XYZ#10, %N**

Names may be any length up to 250 characters and all characters are necessary to identify the name, i.e., two names must match exactly in order to refer to the same numeric variable. You cannot use MegaBasic reserved words (e.g., **FOR**, **IF**, **READ**, etc.) as variable names. Upper and lower case letters always mean the same thing, and MegaBasic displays letters in user-assigned names in upper case only. Chapter 1, Section 5 discusses the use and construction of names in detail.

You can use numeric variable names in any context where one would normally specify a number. Access to named objects in MegaBasic is extremely rapid and the length of a name has no effect on execution speed or program size, no matter how many times the name appears in the program source. There are several different kinds of numeric variables and this manual refers to numeric variables as scalar variables, simple variables, or just variables to distinguish them from array variables described later in Chapter 3, Section 4.

If you access a variable before storing any value into it (by a **READ** or assignment statement), it will automatically contain the value of zero (0.00). However, you should always explicitly initialize all variables before using them to promote clear program structure and maintainability over time.

Integer vs. Real Variables

A variable stores a value in only one representation: either floating point or integer representation. Therefore we refer to variables as either integer variables or floating point (real) variables. When MegaBasic first creates a variable (i.e., when your program accesses it the first time), it gives it a real type or an integer type, an attribute it retains for the life of the program. This initial type selection is governed by the following rules:

- If the variable name ends in a percent sign (%) then it will always be an *integer* variable; if the name ends with an exclamation mark (!) it will always be a *real.*
- If the variable name explicitly appears in a **DEF INTEGER** or **DEF REAL** declaration statement, then MegaBasic creates it as an *integer* or *real* variable, respectively.
- If the variable name begins with a letter that has been declared **INTEGER** or **REAL**, then the variable will be a correspondingly *integer* or *real* variable.
- If none of the above rules apply, then MegaBasic creates a *real* variable by default.

In summary, variables are floating point unless you specify otherwise in your program. Chapter 3, Section 1 gives further details about type declarations and **DEF INTEGER** and **DEF REAL** statements are fully described in Chapter 5, Section 1. Chapter 3, Section 1 also describes important differences and properties of both real and integer representations that you should be aware of.

Section 4: Numeric Arrays

Another type of numeric variable is the array, which stores an ordered set of numbers under one name. MegaBasic organizes an array as an ordered set of storage locations, called elements, identified by a position number within the ordering. For example A(0), A(1) and A(2) represent the first three elements of array A(). The parentheses indicate that A() is an array and serve to contain the position of the desired element. Positions range from zero by integers up to the size of the array. Arrays, as with all other MegaBasic named objects, must have unique names. You name arrays and assign them integer or real data types under the same rules as scalar variables.

You could imagine the array described above as a column of numbers with positions numbered from zero down the side. Suppose that you had many such columns side by side and that you numbered them from zero along the top. Such a structure is called a *2*-dimensional array. By identifying the row and the column we can locate any element of the group. For example A(I,J) refers to the element of A() in row (I) of column a), where I,J are simple variables containing the element position. By adding further levels to this idea, 3 or higher dimension arrays can exist. An *N*-dimensional array requires N position numbers, called subscripts, to uniquely specify an element in the array.

You can specify array subscripts as simple constants, variables or with any general numeric expression. If the specified subscript is a non-integer value, MegaBasic will truncate it (not round) to the next *lower* integer value. Real (floating point) subscripts are internally converted into integer representation before they can be used to access the array. There is a significant performance advantage in specifying array subscripts using integer representation whenever possible, because MegaBasic performs no time consuming real-to-integer conversions. This is especially true in arrays of two or more dimensions and in compiled programs.

Dimensioning Numeric Arrays

In order for an array to exist, you have to explicitly create it in your program. This requires that you specify its name, its type and the range of valid positions for each dimension. You do this with the **DIM** statement, for example:

```
DIM VECTOR(50), MATRIX(12,15), CUBE(20,20,20)
```

This statement defines array VECTOR() as a 1-dimensional array with element positions 0–50, array MATRIX() as a 2-dimensional array with row positions 0–12 and column positions 0–15, and array CUBE as a 3-dimensional array where all 3 dimensions have 21 positions numbered 0 to 20.

Dimension positions always begin at zero and continue up to and including the limit specified for that particular dimension. An *Array Subscript Error* occurs if you attempt to access a dimension position outside its range. One **DIM** statement can define one or more arrays by simply listing their definitions one after another separated by commas. MegaBasic initializes all new array elements to zero as part of time **DIM** process. If you refer to an array before **DIMensioning** it, MegaBasic implicitly **DIMensions** it as a small, 1-dimension array called a *default array*, which is described in more detail shortly.

Once you dimension an array, every reference to it must specify a subscript in each dimension defined. MegaBasic report an error on a reference like **MATRIX(3)** to the

2-dimensional array example above. However you can **DIMension** the array again at any time to change ib size or number of dimensions. If you do, MegaBasic erases it prior contents and sets every element to zero. Using this mechanism, arrays can grow or shrink depending on your program requirements. When arrays are made smaller the unused memory space is available to the system for other uses.

Since the dimensions of arrays can vary during the execution of your program, determining the current dimensions of a given array can be useful from time to time. The **DIM()** built-in function (see p. 425) provides such information for any variable. **DIM(x)** gives the *number of dimensions* of the variable X; **DIM(X,I)** gives the highest position defined for *dimension I* of variable X counting the dimensions from left to right.

Default Arrays

For compatibility with other **BASICS**, whenever your program accesses an array element from an array that does not yet exist, MegaBasic automatically creates small 1-dimensional array. MegaBasic normally creates such arrays, known as default arrays, with 11 elements numbered 0 to 10, equivalent to **DIMx910)**. You can change this default upper bound (of 10) by setting **PARAM**(13) to any upper bound value from 0 to 1023 of your choice.

We strongly recommend that you do not write programs that rely upon default arrays because this practice often complicates the test and debugging phase of developing such programs. For example, by merely misspelling the name of an array in some reference to it, MegaBasic will create a default array of that name if one does not already exist by that name. You can turn off default array creation by setting **PARAM(13)** to a value of –1. Any subsequent references to new, **unDIMensioned** arrays will cause an *Undeclared String Or Array Error,* helpful in locating unintentional default array creations.

Maximum Array Size

As in all computer languages, the amount of memory available to MegaBasic limits the maximum size of new arrays. Within this constraint, however, MegaBasic supports much larger arrays than most other microcomputer languages. You can compute the total number of elements in an array by multiplying the position counts of each dimension. For example an array specified by DIM ARRAY(2,10,8), has 297 elements, as computed by (2+1)*(10+1)*(8+1) = 3*11*9 = 297, where the position count of a dimension is one plus ib maximum subscript value. The memory space taken by an array is simply the number of elements times the element size in bytes. The size of an array element varies with the precision: size = precision / 2+1 (**IEEE** reals are 8 bytes), while integer elements are always 4 bytes each. Hence an integer array with 65500 elements requires 262000 bytes of memory.

You can create as many variables and arrays as you like as long as their combined storage requirements do not exceed the installed memory in your machine. **MS-DOS** based systems are limited to 640k bytes of addressable memory (16 megabytes in *Extended MegaBasic)*. The **FREE()** function, described in Chapter 9, Section 5, provides information about the current memory available so that your program can automatically limit the size of new arrays to match available resources.

MegaBasic supports arrays with any number of elements, as long as no one subscript is higher than 65534. For example, DIM ARR(99,99,99) dimensions ARR() to three dimensions of 100 positions each, or 1,000,000 elements total. To dimension ARR(1000000) is not allowed, because the dimension extends higher than 65534. For

performance and other reasons, several additional restrictions apply to arrays larger than 65534 total elements:

- Pointers to array elements can only access the first 65534 elements. Pointers to arrays (rather than to array elements) are unrestricted.
- The value returned by **INDEX** after a vector **MIN/MAX** will *wrap around* through 0 if the result exceeds 65535.
- Vectors longer than 65535 elements cannot be indexed. Arrays larger than this *can* be indexed, but only as array slices whose length does not exceed 65535 elements.
- Array slices whose successive elements are more than 65535 physical elements apart are not permitted. For example, DIM ARR(10,100,1000) may be sliced as ARR(*,*,J) or ARR(I,*,*), but not as ARR(*,I~ because its successive elements are over 100,000 elements apart.

Obviously, you must have enough memory to support whatever arrays you actually dimension, which tops out around 540k in a 640k DOS machine. Protected-mode versions of MegaBasic, such as Extended MegaBasic, have no 640k limitation and support massive arrays of up to 16 megabytes.

Integer vs. Real Arrays

As with simple scalar variables, MegaBasic stores values of array elements in either real or integer format. All elements of an array provide the same representation, and hence arrays are either all integer or all real. MegaBasic gives arrays a real type or an integer type when initially creating them (either by a **DIM** statement or by its first reference). The following rules govern this initial type selection:

- If the array name ends in a percent sign (%) then it will always be an integer array; if it ends with an exclamation mark (!) it will be a real array; if it ends with a dollar sign ($) is will always be a string array. It is an error to dimension an array with conflicting data types.
- If its name follows the word **INTEGER** or **REAL** in **DIM** statement list, then it takes on that type. For example: **DIM INTEGERX(100),Y(8,10),REAL A(5,5,5)**
- If you re-dimension an array that already exists and the **DIM** statement does not declare the array as **INTEGER** or **REAL**, then the new array will assume the same numeric type as the old array it is replacing. If the **DIM** statement does declare the numeric type, then that type (**INTEGER** or **REAL**) will prevail.
- If the array name explicitly appears in a **DEF INTEGER** or **DEF REAL** declaration statement, then it will take that type.
- If the array name begins with a letter declared as **INTEGER** or **REAL**, then the array will assume that type.
- If none of the above rules apply, then MegaBasic creates a floating point array, by default.

In essence then, arrays are floating point unless you specify otherwise in your program. If more than one of the above rules apply, the lower numbered rule always takes precedence. For example, if you **DIMension** X() as a real array after declaring it an integer array in a **DEF INTEGER** statement, the DIM statement takes precedence and X() becomes a real array.

When you re-dimension an array, the array type (integer or real) always changes to the type specified by the **DIM** statement (e.g., **DIM INTEGER** x(); **DIM REAL** x(); etc.). If the **DIM** statement omits the word **REAL** or **INTEGER**, the array assumes the numeric type already in effect by the prior **DIM** or **DEF** statement. Only the MegaBasic interpreter lets you change array types during execution; the compiler insists that array keep the same type throughout program execution.

You will find further details about type declarations in Chapter 3, Section 1 (a few pages back) and Chapter 5, Section 1 describes the **DEF INTEGER** and **DEF REAL** statements. Chapter 3, Section 1 also describes important differences and properties of both real and integer representations that you should be aware of.

Section 5: Operators and Expressions

The fundamental computational structure in MegaBasic is the *expression,* which you construct from data symbols and operation symbols, much like algebraic notation. Expressions permit you to specify a number as a combination of other numbers. For example (2+5)*3 represents 21 by arithmetically combining 2, 3 and 5. In general, you can use numeric expressions wherever numbers are expected.

Data symbols can be constants (representing fixed quantities), variables (storing the data used), functions (returning computation results), or sub-expressions. A sub-expression is just another expression enclosed inside parentheses to group it as a computational unit. The above expression contains the sub-expression (2+5) to represent the value of 7 in the overall expression.

Operation symbols, called operators, are of two types: unary and binary. Unary operators act on a single number to form a single result number. For example the unary minus operator (–) causes negation of a value that follows it (e.g., –X). Binary operators however act on two numbers to form one result. For example the binary plus operator (+) forms the sum of two values (e.g., X+5).

To facilitate the discussion coming up, we will use the following nomenclature. Numbers acted upon by an operator are called *operands*. The leading operand of a binary operator is called the *left operand* and the trailing operand is called the *right operand.*

Operation Precedence

MegaBasic evaluates expressions by proceeding left to right, accumulating the result with each operation as it goes. The various operators are not however applied with equal priority. Take the following expression using addition (+) and multiplication (*) for example:

```
2 * 3 + 7 * 8 evaluates as:
(2*3) + (7*8) = 6 + 56 = 62
```

A *generally accepted* practice of algebraic evaluation is that, in the absence of parentheses, we should perform the multiplications before the additions. Hence we say that multiplication *takes precedence* over addition. Similarly, MegaBasic applies a *priority scale* to all operators to provide a reasonable order of operations that appear without parentheses. However, you can *force* any order of evaluation as needed by surrounding a sub-expression with parentheses (sub-expressions have the highest prior and take precedence over all operators). For example, to evaluate the addition in the example before the multiplications, just write it like this:

```
2 * (3 + 7) * 8 evaluates as:
2 * 10 * 8 = 20 * 8 = 160
```

The list below summarizes all the MegaBasic numeric operators in order of decreasing precedence. When MegaBasic encounters operators of the same precedence level they are evaluated from left to right.

Priority of Numeric Operations	
18	Evaluation of constants, variables, functions, sub-expressions and string comparisons.
17	Negation (–)
16	Exponentiation (^)
15	Multiplication (*), Division (/), Integer Division (**DIV**), modulo (**MOD**) and the multiple reduction operators (**INT CEIL TRUNC ROUND**)
14	Combining value with sign (**SGN**)
13	Addition (+) and Subtraction (–)
12	Bit-wise integer ones-complement (~)
11	Bit-wise integer shift and rotate operators: << >> ><
10	Bit-wise integer **AND** (&)
9	Bit-wise integer **OR** (|)
8	Bit-wise integer **XOR** (^^)
7	**MIN** and **MAX** operators (not the functions)
6	Numeric comparisons (= <><> <= >= **IN**)
5	Logical complement (**NOT**)
4	Intersection (**AND**)
3	Union (**OR**)
2	Exclusive-**OR** (**XOR**) and Equivalence (**EQV**)
1	Implication (**IMP**)

Operators on the same line have equal precedence and MegaBasic evaluates them from *left to right* as encountered. An example of an expression involving only equal precedence operators is:

X + Y – 3456.03 + ARRAY(J).

MegaBasic permits you to use either integer or real values wherever a number is expected. Internally, MegaBasic usually operates on only one type or the other for any particular operation and if you specify values in the *wrong* type, MegaBasic will convert them to the *right* type. Since this conversion operation is somewhat time consuming, your programs will run much faster if numeric values are always supplied in the form (integer or real) most suited to the operation at hand. The descriptions of each operator follow below and include the specific rules MegaBasic uses for integer and real conversion.

Arithmetic Operators

The arithmetic operators are the most familiar and simplest to describe. The left and right operands around an arithmetic operator are simply combined algebraically into a result value using the specified operation. MegaBasic includes the following operators:

A + B	Computes the algebraic sum of A and B.
A - B	Computes the algebraic difference between A and B.
A * B	Compute the product of A multiplied by B.
A / B	Produces the *real quotient* of A divided by B, even if A, B or both are integer.
A ^ B	Raises A to the power of B. A^0 is always 1, even if A is 0. The left argument may be negative for *integer* powers in the range from –32768 to 32767.
A div B	Returns the truncated quotient of A divided by B. An integer result is returned only if A and B are both integers.
A mod B	Returns the smallest non-negative value which added to A produces number divisible by B, sometimes called the *remainder,* and computes the same result as the **MOD( )** function (Chapter 9, Section 1).

These operators process only operands of the same type (i.e., both integer or both real). If they differ in type, MegaBasic automatically converts one of them to the type of the other, and the operation continues. The result of such an operation is always the same as it would have been if performed in floating point only. Integer comparisons are faster than floating point comparisons.

The divide operation (/) first converts any integer operands to real so that a floating point divide can then yield a floating point quotient. To do an integer divide, you must use the **DIV** operator (e.g., **I DIV J**) and supply two integer operands. When either or both operand is real, the **DIV** operator performs a real **DIV** operation and truncate the final quotient to an integer in real representation. **DIV** always performs an integer divide when both operands are integer, resulting in an integer quotient.

Bit-Wise Integer Operators

Integers in MegaBasic consist of a sequence of 32 zeros and ones called *bits* and it can often be useful to be able to manipulate the *bits* instead of the value they represent. The kinds of things you might want to do include forcing a subset of bits to 1's or *O's,.flip* their state between 0 and 1, shift all the bits up or down or rotate them around the integer as if the integer formed a circle of bits.

MegaBasic supports seven operators providing bit-wise logical and shifting operations on 32-bit integers: four logical operators for implementing bit-wise **NOT**, **AND**, **OR** and **XOR**, and three shift / rotation operators. These operators have an *operator precedence* just below that of arithmetic plus and minus, and above the **MIN** and **MAX** operators. *Real operands* are always converted to 32-bit integers before the operation is applied and every result is a 32-bit integer. All seven operators are summarized below in order of decreasing relative precedence:

~ A	Bit-wise *ones-complement* of all bits in A (changes zeros to ones, ones to zeros). This is like the **NOT** operation on strings.
A << B	Performs an arithmetic left-shift on A by the number of bits specified by B. This shifts zero-bits into the low end of the number as it is shifted left. The shift count may range from 0 to 65535, but over 31 will always give a zero result. A single left shift is equivalent to multiplying the number by 2, presuming that the top bit remains unchanged, and is faster than an integer multiply by 2.
A >> B	Performs an arithmetic right-shift on A by the number of bits specified by B. This shifts the sign-bit into the high end of the number as it is shifted right. The shift count may range from 0 to 65535, but over 31 will always give a 0 or –1 result, depending on the sign of the left operand. A single arithmetic rightshift is equivalent to dividing the number by 2 and is faster than an integer divide.
A >< B	Rotates A by the number of bits specified by B. The rotation count may be positive to rotate left or negative to rotate right. Rotations cause bits that fall off the end of the number to be rotated back into the other end. This is similar to the way that **ROTAT$**() operates on strings.
A & B	Combines A and B using a bit-wise **AND**.
A/B	Combines A and B using a bit-wise **OR**.
A ^^ B	Combines A and B using a bit-wise **XOR**.

The magnitude of the shift and rotate counts has no effect on execution time, as these operations are performed in one step rather than a bit at a time. The shift and rotate operators have equal precedence (i.e., below – and above &).

Special Arithmetic Operators

Any comprehensive library of arithmetic operators should include not only the simple and obvious operators, but it should also recognize a few simple combinations of operators that commonly occur in a broad range of applications. MegaBasic provides such combinations as built-in operators that are more compact, easier to program and faster in execution than the original combination of operators.

A good case in point is the sign operator (**SGN**), which combines the sign of the right operand with the value of the left operand. Without any extra facilities to perform this simple computation, you would have to specify the expression: **ABSQ(X)*ABS(Y) / Y**. Instead, using the **SGN** operator, you can compute the same result as: **X SGN Y**. This not only appears cleaner and more obvious, but it executes many times faster, due to its internal implementation that doesn't rely on multiplies and divides to do the work.

Similarly, MegaBasic includes a number of special arithmetic operators that perform certain simple tasks in a faster, more straight forward manner. Each of these is described in the table that follows:

A SGN B	Computes the *value of* A with the *sign of B*. The result value always has the same numeric type as the left operand (integer or real). For example: 38 **SGN** –5 = –38, 38 **SGN** 5 = 38, –38 **SGN** –5 = –38, 38 **SGN** 5 = 38.
A MIN B **A MAX B**	Selects the **MINimum** or **MAXimum** value between the two operands, for example: 2.3 **MIN** –34.7 = –34.7, 23456 **MAX** 45 = 2~456. This is faster than the more general **MIN()** and **MAX()** functions, which also set the **INDEX** function value as a *side-effect*.
A ROUND B	Computes the *closest multiple of* B to A. This is equivalent to the expression: **ROUND(X/Y)*Y**. For example: 135.4592 **ROUND** .1 = 135.5, 53474 **ROUND** 50 = 53450.
A CEIL B	Computes the *lowest multiple* of B *equal to or greater than* A. This is equivalent to the expression: **CEIL(X/Y)*Y**. For example 354 **CEIL** 25 = 375.
A INT B	Computes the *highest multiple* of B *equal to or less than* A. This is equivalent to the expression: **INT(X/Y)*Y**, for example 354 **INT** 25 is 350.
A TRUNC B	Computes the *nearest multiple of B* between A and zero. This is equivalent to the expression **TRUNC(X/Y)*Y**, for example 27 **TRUNC** 5 is 25, –27 **TRUNC** 5 is –25.

The last four operators (i.e., **ROUND**, **CEIL**, **INT** and **TRUNC**) are the so-called *multiple reduction* operators, which reduce a value to a nearby multiple of another number. As in the other arithmetic operators, MegaBasic automatically forces their operands to the same type before the computation begins and produces a result of the same type. Also, faster execution results when you can supply integer operands.

Logical Operators

MegaBasic provides logical operators to manipulate logic and evaluate logical expressions, but they are unusual in that they do not use the full numeric value of their operands. Instead, MegaBasic uses only the zero or non-zero characteristic of their value instead of their *whole* value. Think of this property in terms of **TRUE** and **FALSE**, with **TRUE** being non-zero and **FALSE** being zero.

The result of a logical operation is always an integer zero (0) or one (1) and reflects the combination of two logical values into one logical result. **NOT** *reverses* the logical value that follows it, i.e., **NOT FALSE** is **TRUE** *(1)* and **NOT TRUE** is **FALSE** *(0)*. Notice that **NOT** only has one operand, similar to the negation operator (–). Operands of logical operators may have an integer or real type, but MegaBasic converts logical operands to an integer 0 or 1 before before evaluating the logical operator. Each of the logical operators are described in the table below:

NOT A	Computes the logical reverse of A, i.e., if A is *true* (non-zero), *false* results; if A is *false* (zero), *true* results.
A AND B	Results in *true* only when both A and B are also *true*.
A OR B	Results in true only if A or B or both are *true*.
A XOR B	(Exclusive **OR**) results in *true* only if one operand is **TRUE** and the other is **FALSE**. The expression A **XOR** B is equivalent to the expression (**A AND NOT B**) OR (**NOT A AND B**).
A EQV B	(Equivalence) results in *true* only if both operands are *true* or both *are false*. The expression **A EQV B** is equivalent to the expression (**A AND B**) OR (**NOT A AND NOT B**).
A IMP B	(Implication) always results in *true* unless the left operand is *true* and the right operand *is false*. The expression **A IMP B** is equivalent to the expression **NOT A OR B**.

A useful way to understand logical operations is to list *all possible* logical inputs (i.e., the operands) alongside their corresponding outputs (i.e., the results). This is usually quite easy to do with logical operations because logic only deals with two values: *true and false*, but not *at all practical* with *real or* integer operations because of the enormous number of combinations. Such enumerations with logical values are called *truth tables*, an important tool in applied logic. A truth table providing a complete definition of all MegaBasic logical operators now follows:

Operator	Left Operand	Right Operand	Logical Result
NOT	--	False True	True False
AND	False True False True	False False True True	False False False True
OR	False True False True	False False True True	False True True True
XOR	False True False True	False False True True	False True True False
EQV	False True False True	False False True True	True False False True
IMP	False True False True	False False True True	True False True True

Ordering Terms For Faster Evaluation

Notice that in certain cases, the result of a logical operation is known by simply knowing the logical value of the left operand, i.e., cases where the result is independent of the right operand. These cases can be summarized as follows:

```
FALSE AND (any value) = FALSE

TRUE OR (any value) = TRUE

FALSE IMP (any value) = TRUE
```

During the course of evaluating logical expressions, MegaBasic may ignore the right-operand (skip its evaluation) when any of the above identities holds. This is done in order to evaluate expressions in the least possible amount of time. In some cases, the time saved can actually lead to a program that runs many times faster. Take for instance the expressions used in the following IF statements:

```
(a) If VALUE=1 and FUNCT(X,Y,Z)>Sqrt(RlS) then ...

(b) If VAL1 or VAL2 or VAL3 or VAL4 then ...

(c) If TEST=LIMIT and (VAL1 or VAL2 or VAL3) then ...
```

In case (a), when **VALUE** equals 1 it is necessary to evaluate the rest of the expression, which involves a user-defined function named **FUNCT** and a square-root computation.

However if **VALUE** equals 0 (false), then the final result is known to be false, so MegaBasic skips over the complicated right operand without having to evaluate it. As this example shows, a many-fold speed improvement may result when the right operand requires much more computation than the left, and the left operand is false.

In case (b), MegaBasic evaluates the expression from left to right and as soon it encounters a *true* term, further evaluation is unnecessary. This is because *true* **ORed** with anything is still *true*. When connecting terms with the **OR** operator, you can make it more efficient if you arrange the terms so that the term most likely to be *true is* first, the next most likely *true* term is second, and so on. You can optimize a similar **AND**-sequence by ordering the terms in a similar manner (i.e., most likely false value first).

Case (c) is a combination of cases (a) and (b). The **ORed** sub-expression to the right of the **AND** is only evaluated if **TEST** equals **LIMIT** (on the left). However when evaluating it, MegaBasic proceeds only until encountering a *true* term (among **VAL1** or **VAL2** or **VAL3**). MegaBasic applies these optimizing identities at all levels of expression evaluation, no matter how complex the expression.

Do not assume that this optimization is always performed, because different implementations may or may not do it (e.g., the compiler *does* evaluate things differently). We mention it here so that you can order your operands for the most efficient processing and so that you do not depend on the right operand necessarily being evaluated. Nor should you depend on the lack of evaluation of a right-operand, even if the above conditions are met. Right-hand operands that affect the contents of variables or other program-state conditions must be coded with the knowledge that they *may* or *may not* need to be evaluated.

Logical Expressions In Arithmetic Calculations

Since logical expressions always evaluate to either zero (0) or one (1), you can use them within numeric expressions for computational purposes which might not otherwise appear to be logically oriented. In many instances, combining logical terms with numeric terms can yield a faster computation or a more compact or convenient representation than would otherwise be possible. You can sometimes eliminate IF statements with such techniques, for example:

Example Logical Expression	Equivalent IF Statement
COUNT = COUNT+ (THIS or THAT)	If THIS or THAT then COUNT = COUNT+1
VALUE = LIMIT/(2+ (X=Y AND Z>10))	IF X=Y AND Z>10 then VALUE = LIMIT/3 else VALUE = LIMIT/2

Remember that the result of a logical operation is an integer result, never a real result. You might want to consider the possible performance consequences of this depending on the context. However such consequences only affect execution speed and are logically transparent to the particular application involved.

When you employ logical operators for numerical purposes you must be aware of the operator precedence involved, or you could easily produce meaningless results. Although this is the case with expressions of any type, the range of operators in MegaBasic is greater than is generally supported in most other languages. Therefore you should experiment with unfamiliar operators in simple expressions to understand them before applying them in complex situations.

Comparison Operators

Comparison operators compare two numbers (integer, real or mixed) or two strings and pass back the outcome of the comparison. You can compare an integer value with a real value, but MegaBasic automatically forces them to the same type internally before actually comparing them. A *Type Error* occurs if you attempt to compare a number with a string. When you perform a comparison, you are looking to see if some relationship between the numbers is *true or false*. For example you may want to test whether one number is equal to another number. The equality comparison returns *true* if they are equal and false if not equal.

By convention, MegaBasic (like most other computer languages) represents logical values with numbers: 1 means *true* and 0 *means false,* and represents such values in *integer format* rather than in floating point. Logical values (true and false) are primarily found in **IF** statements and **WHILE** or **REPEAT** loops to decide what the next step of the program should be. Based upon the outcome of a comparison, your program can choose one set of actions over another. However, logical operators can also be used within arithmetic computations for their 0 or 1 value whenever desired. For example the statement: **COUNT=COUNT + (X>Y)** adds 1 to **COUNT** only if X is greater than Y. All the comparison operators are described in the table below:

A = B	Returns a *true* if A and B are exactly equal, and false otherwise.
A < B	Returns *true* if A is less than (below) B, and false otherwise.
A > B	Returns *true* if A is greater than (above) B, *and false* otherwise.
A <= B	Returns *true* if A is less than or equal to (not above) B, and false otherwise.
A >= B	Returns *true* if A is greater than or equal to (not below) B, and false otherwise.
A <> B	Returns *true* if A and B are not exactly equal, and false otherwise.
A IN B	Returns *true* if all 1-bits in A are also set to 1 in B, and *false* otherwise.

You can compare two expressions results just as you compare simple values. The operator precedence scale becomes important in such comparisons to reduce the need to control operation order with parentheses in expressions involving many diverse operators. The following expression illustrates such a calculation:

```
A + B * C > X * Y ^ Z OR Q – R / S = A * B AND F *17 < B + C
```

You can greatly improve the readability of expressions like this by carefully inserting / deleting spaces between operators to make them stand out and by grouping the operations with parentheses, as in:

```
(A+B*C)>(X*Y^Z) OR (Q–R/S=A*B) AND (F*17<B+C)
```

Since string comparisons also return 1 for **TRUE** and 0 for **FALSE**, you can use them within larger numeric expressions as needed. For example the expression I + A$=SB$ computes the value I+1 if A$ and B$ are identical, or the value I+0 if they are not. Refer to Chapter 4, Section 4 for details on how MegaBasic compares strings.

Section 6: Numeric Functions

As we have shown, numbers can be expressed as constants, variables and numeric expressions. However they may also be expressed as results of special procedures called functions. Functions are similar to array variables, in that they are referred to by name and include additional information which affects the value that they represent. The difference is that an array element merely accesses the value it holds, but a reference to a function invokes a computation of the value symbolized by the function name. As with constants and variables, functions may be combined with other values within numeric expressions to calculate further results.

A function is a procedure which computes a result based upon data which you have communicated to it. To identify each procedure, functions are assigned names just like variables. To use a function, you merely type its name and its input data just as if you were typing an array name and its subscript list. For example, consider the following three function references:

Sqrt(17)	**Round(X,3)**	**Min(R+2,189,VALUE)**

First, we have the square-root of 17. Second, we specified a value equal to the contents of X rounded to exactly three significant digits. Third, we access a value equal to the minimum value specified among the expressions: R+2, 189, and **VALUE**. Functions are always of the same form:

<function name> (<argument list>)

Input information to the function is specified after its name, enclosed in parentheses, as a list of numeric or string values called an argument list. Each input value is called an argument and is specified using any general expression. The values computed by these expressions are used by the function in forming its ultimate result. The number of arguments and their type (string or numeric) depends on the particular function being used. When more than one argument is present, they are separated from each other with commas. Some functions have no arguments, and are specified with the function name alone: no parentheses follow it.

MegaBasic possesses a library of over eighty built-in functions and also allows you to create your own functions, written in MegaBasic statements. Chapter 9 provides a complete description of all the built-in functions in MegaBasic and how to use them. Defining your own functions is a somewhat more advanced topic that is thoroughly covered in Chapter 8, Section 3. Refer to those subsections for more complete details. A summary of the numeric functions now follows for quick reference.

Summary of Arithmetic Functions	
Int(X)	highest integer not above X
Ceil(X)	least integer not below X
Trunc(X)	X without its fractional part
Mod(X,Y)	remainder of X divided by Y
Frac(X)	the fractional part of x
Round(X)	X rounded to the nearest integer
Round (X,P)	X rounded to P significant digits
Abs(X)	absolute value of X
Sgn(X)	1 with the sign of X, 0 if X=0
Sgn(X,Y)	Y with the sign of X
Max(X,Y,...)	the maximum value among a list
Min(X,Y,...)	the minimum value among a list
Index	Secondary result of certain functions
Rnd(X)	pseudo random sequence
Integer(R)	integer representation of R
Real(I)	Real representation of I

Summary of Mathematical Functions	
Sqrt(X)	square-rootof X
Log(X)	logarithm base 10 of X
Ln(X)	logarithm base e of X
Exp(X)	*e* to the power of X
Pi	the constant *pi*
Sin(X)	sine of X radians
Asin(X)	arcsine of X
Cos(X)	cosine of X
Acos(X)	arccosine of X
Tan(X)	tangent of X
Atn(X)	arctangent of X
Poly (X,A,D)	generalpolynomialevaluation

Because variables are created by default when encountered for the first time and not **DIMensionded**, misspelled function names will result in variables being created under those names. Such errors can be very difficult to diagnose because there is no way for MegaBasic to detect the error. For example, **SQRT(I)** returns the square-root of I, and **SQR(I)** returns the Ith element of array **SQR()**.

Two facilities exist in MegaBasic to aid the discovery of misspelled names. One is the **NAMES** commands (Chapter 2, Section 3), which displays an alphabetical list of all user-assigned names in the program. Unrecognized names that appear in this display should be investigated. Mistyped variable and function names tend to be displayed in close proximity to the correct spelling of the user-assigned name, due to the alphabetical ordering of this display. The second debugging aid is the **XREF** command (Chapter 2,

Section 5), which displays all references to any name. Names that have only one reference should be scrutinized as possible misspellings.

Integer vs. Real Functions

MegaBasic lets you to use either integer or real values wherever a number is desired. Internally, MegaBasic usually requires one type or the other in order to process the intended operation and if you specify values in the *wrong* type, MegaBasic will convert them to the *right* type. Your programs will run faster if numeric values are always supplied in the form (integer or real) most suited to the operation at hand. Implicit type conversions involving the built-in functions are governed by the following considerations:

- All transcendental functions, such as **SQRT()**, **LOG()**, **SIN()**, **COS() ATN()**, etc., use a real argument and return a real result. Other functions which always return a real result include: **PI**, **POLY()**, **RND()**, **FRAC()** and **VAL()**.
- A number of functions return a result of the same type as their argument(s). These include: **ROUND()**, **TRUNC()**, **CEIL()**, **INT()**, **ABS()**, **MOD ()**, **SGN()** with two arguments, and **MIN/MAX** functions. The **MIN/MAX** functions return a real result if any value in their argument list is real, otherwise they return an integer result.
- All other MegaBasic numeric functions return integer results. Using them in exclusively integer contexts will be faster than in combination with real values (also called *mixed-mode* expressions).

Integer and Real Conversions

Two special functions are provided to force any expression value into real or integer representation, regardless of the current type of the value. The **REAL**() function always returns a real representation of its single numeric argument. The **INTEGER**() function always returns an integer representation of its single numeric argument. A error will result if you attempt to form an integer from a value too large to represent as a 32-bit signed integer (i.e., above 2147483647 or below –2147483648). If a real value with places to the right of the decimal is supplied to the **INTEGER**() function, the number will be truncated to a whole number and then converted to an integer. Such truncation of real values will always occur any time a non-integer real is converted to an integer representation.

An integer value can be converted to a real value without precision loss in all floating point precisions except 8-digit **BCD**, in which integer values beyond 100 million (+–) cannot fit within the floating point representation. Therefore the value is truncated to contain only the leading 8 decimal digits of the integer. Values between 100 million and 1 billion will always be within 9 of the actual value; values over 1 billion will be within 99 of the original integer value after being converted to real. If your program never uses integers of this size then 8-digit MegaBasic can be used without any difficulties.

Section 7: Vector Processing

MegaBasic supports an integrated family of vector processing capabilities. In the same way that a string is a sequence of characters, a vector in MegaBasic is simply a sequence of numbers. Vector operations are provided that allow you to manipulate vectors in expressions (vector arithmetic), to potentially control thousands of operations in one statement. Some **BASICS** provide a few matrix operations, but vector processing techniques can be applied to implement any matrix operation, such as matrix inversion, multiply, add, transpose, linear programming, etc., without restricting the language to only those matrix operations that were included. Procedures for some of these operations are implemented in **LIBRARY.pgm** included in the MegaBasic software set.

An algorithm implemented with vector operations can execute from 3 to 12 times faster than the same algorithm implemented iteratively (i.e., looping through the individual elements). This is because vector operations generally replace the innermost loops of many algorithms with one or two single vector statements, where virtually all the processing is concentrated, and the vector operations themselves are compiled on-the-fly and executed, instead of interpreted.

Vector Variables

Several statements and functions are supported that provide a complete vector processing facility in MegaBasic. To effectively use these constructs, you need to understand how to specify vector variables and vector expressions. A vector variable is defined as:

- Any numeric scalar variable or single array element reference. This is the shortest possible vector: a vector of length one.
- An array name without any subscript expression, representing a vector that consists of all elements contained in the array, even if the array has more than one dimension. In multi-dimensional arrays, the element order is the same as the traversal by the following program:

```
Dim ARRAY(L,M,N)
For I = O to L; For J = O to M; For K– O to N
Print ARRAY(I,J,K); Next K; Next J; Next I
```

In other words, we advance a subscript only after sequencing through all possible combinations of all the subscripts to the right of it. This type of vector variable lets you process any array as if it were one long list of numbers.

- An *array slice*, representing a vector consisting of all elements of the array that intersect with a *slice* through one or more dimensions of the array. Array slices are described below.
- A *concatenated vector variable,* which is a list of vector variables separated by commas and surrounded by brackets. This type of vector expression is discussed later on.

Specifying Array Slices

In a vector context you can access a series of array elements by specifying one of the array subscripts with an asterisk to mean all elements contained in that dimension. For example consider the array **ARRAY**(20,30) consisting of 21 rows (0 to 20) and 31 columns (0 to 30). The vector **ARRAY**(i,*) is the sequence of elements from row I spanning **ARRAY**(i,0), **ARRAY**(i,1),..., **ARRAY**(i,30). Likewise the vector **ARRAY**(*,j) is the sequence of all elements in column J. Think of the asterisk as being a *wild-card* that means all possible locations in that dimension. This notation is called an *array slice*, because it refers to all the array elements intersected by a *slice* through an array.

You can specify more than one asterisk subscript, as long as they are *adjacent to one another* in the subscript list. The following table illustrates various array slices using a three-dimensional array to help you understand the meaning of the asterisk notation. We will conceptualize this array as a stack of levels, each consisting of elements arranged in rows and columns.

ARRAY(i,j,k)	Single element vector, using the value in column K of row J on level I.
ARRAY(i,j,*)	The elements at all columns at the intersection of row plane J and level plane I.
ARRAY(i,*,k)	The elements in all rows at the intersection of column plane K and level plane I.
ARRAY(*,j,k)	The elements in all levels at the intersection of row plane J and column plane K.
ARRAY(i,*,*)	The elements from all rows and columns on level I, i.e., a slice through the *plane* of one level.
ARRAY(*,*,k)	The elements from column K on all rows in all levels, i.e., a slice through the *plane* of one column.
ARRAY(*,j,*)	This is illegal because the asterisks are not adjacent, and reported as an *Array Subscript Error* if you try it.
ARRAY(*,*,*)	All elements from the entire array. This is equivalent specifying the array name without any subscripts at all.

Scalar variables or arrays that have never been **DIMensioned** or created by default cannot be referenced as a vector. Attempts to use such uninitialized variables in a vector context will be reported as *Out Of Context Errors.*

Concatenated Vectors

As mentioned earlier, a vector variable can be specified as the concatenation of two or more vectors, by enclosing the component vectors in brackets []. For example [X,Y,Z] is a vector with three vector variables, forming a sequence of numbers consisting of the three vectors placed end to end. The component vectors within brackets may have any of the following forms:

- **A scalar variable**
- **An array slice or an indexed array slice**
- **An unsubscripted array name**
- **A scalar expression that does not begin with a user-definedidentifier**

All components of a concatenated vector must have identical data types. In other words, all components must be integers or all components must be floating point (real). A *Data Type Error is* reported if this rule is violated.

The last form lets you specify 1-element vector components using ordinary arithmetic expressions. For MegaBasic to discriminate between such an expression and a vector variable, the expression cannot begin with an identifier (e.g., you can surround scalar expressions with parentheses). Hence scalar expressions considered valid include any expression that begins with a numeric constant, a left parenthesis or a built-in MegaBasic function. Components specified in this manner represent read-only values. If you store data into a concatenated vector, any read-only components that it contains will be treated as variables and modified accordingly. No error is reported for this condition; you simply lose whatever value is stored there.

Vector Variable Indexing

In some situations, you may only want a portion or sub-sequence of a vector specified by an array slice expression. Therefore MegaBasic lets you append an indexing expression onto any array slice, much like the indexing expressions supported for string variables. Unlike string indexing, a vector index position is zero-based (rather than one-based) and you can only index vector variables: vector expressions and concatenated vectors cannot be indexed. See Chapter 4, Section 5 for complete information about indexing strings. The examples below show how index expressions are applied to array slices:

Indexing Vector Variables	
ARRAY(*,I)(J,K)	Elements J through K of column I.
ARRAY(I,*)(J)	All elements in row I from position J to the end.
ARRAY(*,I)(K:L)	L elements starting at position K of column I.
ARRAY(I,*)(:L)	The last L elements of row I.

Extended (or compound) index expressions are also fully supported, i.e., indexing an already indexed vector. Note that you cannot index an array without any explicit asterisk subscripts because MegaBasic assumes that the first parenthetical expression that follows an array name must be a subscript expression, not an indexing expression. Unlike indexed characters strings, indexed vectors must select at least one element; a *null vector* is not allowed.

Vector Expressions

Computations involving vectors are expressed in much the same way as *ordinary* scalar calculations. For example if X and Y are vectors, the expression (X+Y) / 2 will produce a result vector whose elements are the average of the corresponding elements in X and Y. There is virtually no limit on expression complexity or parenthesis depth, and the internal memory required during the computation is only slightly greater than that required for a similar scalar computation.

Vector expressions are evaluated completely for the first element of every term of the expression, followed by the second element of every term throughout the expression, and so on through to the last element of each term. If the vectors of an expression differ in length, then the shorter vectors will *run out* before the longer vectors are accessed. When this happens, the shorter vectors simply *wrap-around* back to their first element again, so that the expression computation can continue until the last element of the *longest* vector has been processed.

For example the expression X+3 is the sum of a vector X and a constant. A constant is really a vector of length one, so that when we evaluate this expression the constant becomes, in effect, a constant vector of length equal to the length of X. Some other

important applications of this wrap-around idea will be shown later on, but for now the important thing to understand is that a vector expression always produces a vector result equal in length to its longest vector variable. You can control any wrap-around by controlling the lengths of the vectors involved.

Vector Operators

All the standard MegaBasic *arithmetic* operators are supported in vector expressions. Mixed-mode (i.e., integer and real) arithmetic is supported under the same rules as in scalar arithmetic, including operator precedence relationships. Unlike scalar arithmetic, if the result of an integer calculation exceeds the capacity of a 32-bit integer, a numeric overflow error is reported, instead of converting the integer to floating point and continuing on. If an error occurs during a vector computation, you can determine on which element the error occurred using the **INDEX** function, which always returns the number of correctly computed vector result elements.

All the *logical* operators (i.e., **AND**, **OR**, **XOR**, **EQV**, **IMP** and **NOT**) and all the *comparison* operators are supported. As in the scalar context, they return an integer 0 or 1 result, or rather, a vector of 0's and 1's. Using these, the expression SUM(X>=10 and XC=30), for example, computes the number of elements in vector X that lie in the range from 10 to 30.

Vector Functions

All arithmetic functions (Chapter 9, Section 1) and mathematical functions (Chapter 9, Section 2) are supported in a vector context, with the following exceptions:

ROUND() with two arguments	**SGN() with two arguments**
RND() with one argument	**POLY() function**

Pi is supported as well as **INDEX**, which begins at zero and increments by one as the vector expression sequences from element to element. **INDEX** can be used in a vector expression as a running counter-vector, or after an error (such as divide by zero) to determine the element that caused the error during the vector computation. The dramatic speed improvement of vector operations over iterative implementation may be reduced when transcendental functions are applied to vectors, simply because such operations are dominated by computation.

None of the file and device I/O functions are supported, nor are the utility and system functions, except for **INTEGER**() and **REAL**(). When you apply a function to a vector expression, each element of the expression result vector is transformed by the function to produce a new vector of transformed elements. For example **SQRT**(X+Y) returns a vector consisting of the square root of the sum of the corresponding elements in vectors X and Y. If you attempt to use any MegaBasic function that is not supported in a vector context, an *Out Of Context* error will be reported.

Scalar Functions on Vectors

MegaBasic provides several functions that operate on a vector and return a scalar numeric result (i.e., a single number) including **MIN()**, **MAX()**, **SUM()**, **LEN()** and **FIND()**. These functions are used in normal expressions because they return a simple numeric result. You cannot use them in *vector expressions* (i.e., they do not return vectors of sums, lengths, minimum or maximum values).

- MIN() and MAX() on Vectors

  MIN() and MAX() let you include vector expressions as argument from which the **MIN** or **MAX** value is determined. You must precede vector expressions by the **VEC** reserved word in so that MegaBasic will evaluate it as a vector. The argument list of vector and scalar expressions is scanned from *left-to-right* and the **MIN** or **MAX** value is returned. Afterward, the **INDEX** function returns the sequence position (one-based) of the value found, as if all scalars and vector elements were scanned as one long list.

- SUM(vector exprn)

  SUM() evaluates a vector expression and returns the sum of the resulting elements. For example, the expression SUM(X*X) computes the sum of the squares of each element of vector X. The word **VEC** is not needed in SUM() because it *only* operates on vectors.

- LEN(VEC vector exprn)

  LEN() returns the length of a vector expression, i.e., its element count. The **VEC** word is needed to indicate that a vector expression is coming up, not a string expression. LEN() does not *evaluate* the vector expression; it only computes the length of the longest vector term within the expression.

- FIND(VEC vector exprn)

  FIND() locates the first *non-zero element* in an arbitrary vector expression, returning either the index position found (zero-based), or –1 if *all elements* were zero. For example, if all elements of X() are zero except for X(17) then **FIND(VEC X)** returns 17. To locate a value in a vector satisfying some condition, specify the condition as the vector expression, e.g., **FIND(VEC X=99)** or **FIND(VEC X>20 AND X<30)**. If you use vector indexing to limit **FIND**() for partial searches, the position returned is relative to the region searched, rather than to the beginning of the entire vector.

Vector Statements

The vector processing statements are simply enhancements of selected scalar processing statements that already exist in MegaBasic. These include vector assignments and swapping, printing, and file reads and writes. In each of these statements, you must indicate that a vector operation is coming up, by preceding the operation with the special reserved word **VEC**. We will describe each of the vector statements in the discussion that follows.

The **INDEX** function, referred to elsewhere in the discussion, returns the number of processed elements at any point. It is often useful for setting vectors to an arithmetic sequence.

Some vector operations can take a while to execute, depending primarily on the number of elements to be computed and the complexity of the calculation. Heavy use of transcendental functions on 100,000 elements without a math coprocessor can take quite some time to complete. During this time, Ctrl-C is not recognized, causing a perceptible delay between the time you type a Ctrl-C and the time your program stops.

Vector Assignments

Virtually all vector processing is performed by vector assignment statements, which have the form:

```
VEC <vector variable> = <vector expression>
```

The reserved word **VEC** announces a vector assignment is ahead, the <*vector variable*> defines where to store the resulting vector, and the <*vector expression*> defines the vector calculation to be performed.

The length of the vector variable dictates the extent to which the vector expression is performed. For example in the vector assignment VEC X = 3, the constant 3 is a one-element vector which is extended (or repeated) to match the length of vector variable X. As another example, consider the following program fragment:

```
Dim X(100), Y(10);
Vec Y = index; Vec X = Y
```

First create two vectors (arrays), one with 101 elements and one with 11. Then assign the **INDEX** function value to each element of Y(*). The **INDEX** function always returns the number of successfully computed vector elements from any vector computation. However within a vector expression, **INDEX** creates a vector consisting of an incrementing series of integer values starting with zero.

Finally, we assign vector Y to vector X. Since Y is shorter than X, MegaBasic extends Y to the length of X by repeatedly wrapping around to the beginning of Y each time it *runs out*. This results in X containing 9 concatenate series of integers 0 to 10, finishing with 0 and 1 in elements 100 and 101. This automatic repetition can be useful in matrix manipulation, as demonstrated by the assignment statement:

```
Vec M(*,*) = M(*,*) + R(*)
where M(i, *) and R(*J are the same length
```

If M() and R() have the same number of columns, the statement above adds R(*) to every row in M(). This implicit repetition means that you must be careful when setting up vector operations to specify vectors of the appropriate lengths at all times. Just one element too few or too many can easily produce invalid results that may be difficult to detect, especially when other vector operations follow.

You also need to use care in applying the automatic repetition to avoid excessive computation when doing simple things. If, for example, S is a scalar variable and X is a 10,000 element array, the statement VEC X - SQRT(S) would compute and store the square root of S 10,000 times, a very time consuming and wasteful approach. It is much more efficient to compute and save a complex result once, then store it into a vector as a separate step.

A concatenated vector variable (enclosed in brackets [ ] as described earlier) may be the target variable of a vector assignment, as in:

```
Vec [A,B,C,D] = X(*,J)
```

The vector expression result on the right is distributed among the variables on the left as if they formed one continuous variable, even if any or all of the concatenated variables are also vectors. If A, B, C and D are scalar variables, then they receive the first four elements from column j of array X(). As always, the result vector is extended to match the length of the receiving vector variable, which in this case is the combined length of the concatenated variables.

When the assigned vector variable also appears in the vector expression to the right of the equals sign (=), remember that each element is computed and stored one at a time. In particular, computing one element using the value of another element of the same vector may not work. Consider the following example:

```
Vec X(*)(1) = X(*)
```

This assignment statement appears to assign the values from elements 0 and up to element positions 1 and up, i.e., shift all element values up by one element. In fact, what this really does is to copy the value of X(0) to all elements of the vector. This is because later elements are stored using the results of earlier elements and vector calculations are always done in ascending sequential order, resulting in the sequence: X(1)=X(0), X(2)=X(1), X(3)=X(2), and so on. This computational property may be useful in certain applications but in most cases specifying such assignments causes errors that may be difficult to diagnose.

With careful application, however, you can take advantage of the sequential nature of the vector computational process for special purposes. An important example of this is converting a series of values into a cumulative series. The following program fragment does just that

```
Vec X(*)(1) = X(*)(1) + X(*)
```

This computation first sets X(1) = X(1)+X(0), then sets X(2) = X(2)+X(1), and so on, so that each resulting element is the sum of itself and all elements preceding it. We can also convert this cumulative series back to is original incremental series using a similar technique, as shown by the program fragment:

```
Vec Y = X; Vec X(*)(1) = Y(*)(1) – X(*)
```

In this case, we have to copy vector X to another vector so that the elements needed in the calculation are not modified before they are used. The result is that each new element X(i) = X(i) – X(i–1).

Swapping Vectors

In matrix applications, one frequently needs to exchange of the contents of two vectors, such as in matrix transposition and matrix inversion procedures. This generally time-consuming process can be performed using the **SWAP** statement which is 7 to 12 times faster than a similar implementation using **FOR..NEXT** loops. For example, the following routine transposes an N-by-N matrix using vector swaps:

```
For l = O to N–1
Swap vec MATRIX(*,I)(I), MATRIX(I,*)(I)
Next I
```

This routine swaps the contents of each corresponding row and column. As shown above, the **VEC** reserved word must precede each pair of variables to be swapped, so as to distinguish them from other scalar variables or strings to be swapped in the same statement. As with all vector operations, the **INDEX** function returns the number of elements processed after the operation has completed.

When you swap two vectors of different lengths, the process continues until the last element of the longer vector has been swapped, and the shorter vector is re-started from the beginning whenever it runs out of elements to swap. For example if you swap a vector with a scalar (a vector with one element), the scalar is repeatedly swapped with each element of the vector. The net effect of this is to *insert* the scalar value into the

vector, and move the *extra* value that *falls off* the end of the vector into the scalar variable. A similar insertion occurs when a long vector is swapped with a short vector.

This capability can be useful in vector sorting, vector element rotation, element insertion and deletion, and other manipulations on numerical arrays. It also means that you must be careful when you specify a vector swap operation to ensure that the lengths of both vectors are exactly correct, to avoid an unintended result.

Printing Vectors

You can **PRINT** the resulting elements of a vector expression by merely specifying the vector expression, preceded by the reserved word **VEC**, as any term of a **PRINT** statement. Each value is printed with the appropriate format, just as if each element was specified as a separate (scalar) expression, for example:

```
Print %"12f2,8i", Vec X(*,j)
```

This statement prints all the values from column j of array X() to the console, in a format that alternates between 12F2 and 8I. This capability eliminates the need for **FOR..NEXT** loops for similar applications of **PRINT**. *You* can **PRINT** simple vector variables and vector expressions of any complexity. The **VEC** reserved word must precede each vector to be **PRINTed**; expressions not marked in this way are assumed to be scalar expressions.

Writing Vectors to Files

Like the **PRINT** statement above, you can specify vector expressions in the output list of a **WRITE** statement. You must precede each such expression with the **VEC** reserved word to inform MegaBasic of your intentions. The vector elements are written to the disk file in binary format (integer, **IEEE** real or **BCD** real) and in the order they occur within the result vector. If **PARAM(11)** has been used to change the floating point precision written to files, each element will be converted to that precision as it is written. If a **WRITE REAL** or **WRITE INTEGER** statement is being performed, the vector elements are converted to the representation indicated as needed. You cannot specify the byte override ampersand (&) or the word override at-sign (@) on vector write operations.

Reading Vectors from Files

You can read vectors from a file by specifying a vector variable in the input list of a **READ** statement. You must precede each such variable with the **VEC** reserved word. The vector elements are read from the disk file in binary format (integer, **IEEE** real or **BCD** real) into sequential elements of the receiving vector. The number of values read is determined by the length of the receiving vector variable. If **PARAM(11)** has been used to change the floating point precision read from files, each element will be converted from that precision to the internal precision of MegaBasic as it is read into the vector element. If a **READ REAL** or **READ INTEGER** statement is being performed, the values of the type indicated are read and, if necessary, converted to the numeric type of the vector variable as they are stored. You cannot specify the byte override ampersand (&) or the word override at-sign (@) on vector read operations. The non-file **READ** statement (for **DATA** statements) does not support vectors.

Reading numeric vectors from files is 4 to 10 times faster when the vector elements follow one another in memory (i.e., contiguous elements). A vector is contiguous if it is the entire array or its rightmost subscript is an asterisk (*). Such a vector is still contiguous after any indexing is applied. In this special but very common case, all the

elements are read directly into the vector in one disk operation, instead of a potentially separate disk read for each element. Separate disk transfers are used whenever the read involves precision conversions (i.e., **PARAM 11 <> PARAM 4**), numeric type conversions (i.e., from **READ REAL** or **READ INTEGER**), or non-contiguous elements (e.g., **VEC** X(*,K), Y(*,*J), etc.).

When the high-speed vector read is performed, the individual elements are not validated in any way. Single-element reads employ a very simple validation of each value read, but this is only of use when the file truly contains garbage. If an actual read error occurs, such as reading past the end of the file, the **INDEX** function does **NOT** return the number of correctly read elements because **INDEX** is updated *after* the disk read has successfully completed.

Section 8: IEEE Floating Point and 80x87 Math Support

MegaBasic is available in either of two fundamentally different floating point representations: **BCD** floating point, and **IEEE** double precision binary floating point. Unlike the **BCD** MegaBasic, **IEEE** MegaBasic provides full 80x87 numeric processor support. This version, called **BASIC87** and **RUN87**, automatically detects the presence of the numeric processor so that subsequent arithmetic operations can take full advantage of its capabilities or emulate its functionality in software when not present.

BCD vs. IEEE Representation

Before jumping into what **BASICS87** can do, it is instructive to contrast and compare the two floating point representations supported by MegaBasic, i.e., **BCD** and **IEEE**. **BCD**, which stands for *Binary Coded Decimal,* is a representation format that *packs* two decimal digits (i.e., 0 to 9) into an 8-bit memory value. A **BCD** floating point number consists of a series of these packed bytes followed by the byte containing the *sign* of the number and a power of ten scaling factor that indicates the *magnitude* of the number. Under MegaBasic this power spans –63 to 63, providing a numeric range from lE–63 to lE+63. **BCD** floating point representation has a number of advantages:

- All decimal numbers within the precision provided by the **BCD** format are represented *exactly*. For example, using 14-digit **BCD** format you can represent hundreds of billions of dollars to the penny without any round-off error. This makes **BCD** well suited for financial work or other applications where input values must be represented exactly.
- Numbers must ultimately be represented in display code or ASCII character representation for both input from the keyboard and output to a printer or screen. Converting between ASCII and **BCD** floating point is very quick and requires only a small amount of program code to perform it. On the other hand, converting between ASCII and binary numeric representations is a much more complex and time consuming task.
- **BCD** numbers can be read directly from *hex dumps* of files or memory without any special conversion performed. This is of great assistance in certain types of machine / assembler code debugging.

Two disadvantages of **BCD** floating point should be noted however. **BCD** is slightly less efficient with storage than pure binary representation. This is because when two decimal digits are packed into an 8-bit byte, a small part of each byte goes unused. For example 5 bytes can theoretically contain 12 digits of precision, but with **BCD** coding they can only hold 10 digits, two per byte. The second disadvantage is that hardware-assisted computation for **BCD** format is virtually non-existent and therefore **BCD** MegaBasic will likely be limited to software-only arithmetic.

IEEE double precision format is a purely binary method for representing floating point numbers. It consists of a 52-bit fractional part called the mantissa, an 11 bit power of two scaling factor called the exponent, and one more bit for the sign. This representation has three advantages over **BCD** format:

- **IEEE** arithmetic implemented in software can be more efficient than **BCD** arithmetic implemented in software, especially multiply and divide.

- Hardware computational support for **IEEE** format is available on many fronts. In 8088 / 86 / 286 applications, the Intel 80x87 chips provide this, and have exceptional support for transcendental functions.
- **IEEE** binary representation provides the maximum storage efficiency possible. Its 8 bytes provides enough precision to store 16 digits and an exponent that supports a dynamic range from lE–308 up to lE308. In contrast, an 8-byte **BCD** floating point number can hold 14 digits with a dynamic range from lE–63 to lE63.

An important disadvantage to be remembered about **IEEE** double precision format is that very few numbers with decimals can be represented exactly. For example 0.1 cannot be represented exactly, just as 1 / 3 cannot be represented exactly under **BCD** format. However, this problem is more pervasive with **IEEE** format simply because decimal numbers are the basis for nearly all input and output of numerical information, as well as specification of numerical constants in programs. This does not mean that calculations are any less accurate using **IEEE** format, just that, in many cases, the original decimal data will contain small round-off errors after it is stored internally. Such round-off errors are inherent in the **IEEE** representation, and are not *bugs* in software that supports it, like **BASIC87**.

IEEE/BCD Compatibility

BASIC87 is designed to run programs originally written under **BCD** MegaBasic without any program changes. There are, however, a number of areas that you need to be aware of which can potentially alter the outcome of certain operations. We consider all these differences to be insignificant in the vast majority of applications. The issues are as follows:

- When floating point variables are read from or written to data files directly, the prevailing floating point representation is assumed: **BCD** versions read / write **BCD**, **IEEE** versions read / write **IEEE**. PARAM(11) can be set to modify this behavior, however, as we will discuss later on. **IEEE** values require the same amount of memory / file space as floating point values under the 14-digit **BCD** version: 8 bytes each.
- Operations that use knowledge of the internal **BCD** representation will no longer work correctly. The only way this can occur is by using **EXAM** and **FILL** statements to access memory locations containing floating point numbers, or by passing the memory address of these numbers to machine code subroutines outside of MegaBasic. Very few programs will be affected by this incompatibility.
- Results from complex calculations will have small differences in the least-significant digit or two, so you should not rely on identical full-precision results for your program to be correct. If you can run your application under different precisions of **BCD** MegaBasic, then you should also be able to run it under **IEEE** MegaBasic.
- Decimal numbers stored in **IEEE** floating point variables will not always be represented exactly, the way they are in **BCD** variables. This shows up in **FOR..NEXT** loops with certain non-integral step sizes, sometimes causing the loop to execute one less iteration than with **BCD** floating point numbers.
- The exponents of numbers formatted with E-notation require one more column of width (for 3 digits instead of 2). This may result in an overflow of the field width and no value will be shown. If this rare case is encountered, the field is filled with asterisks (*) and the program continues on.

- The relative time to compute most floating point operations will differ under **IEEE** and programs relying on such timing may require adjustment. A few operations are slower under **IEEE** MegaBasic, such as ROUND(x,n) and converting between ASCII and floating point, but most operations are much faster.
- The RND() function generates a completely different sequence of random numbers under **IEEE** than under **BCD**. Results from programs using RND() run under different versions will not match.
- The **TYP**() function is not able to reliably distinguish binary data from strings and end-of-file marks (and *never* could). Therefore **TYP**() should not be used to determine the next data type on files that contain binary data (such as **IEEE** floating point, binary integers, etc.).

Floating Point Values on Files

By default, MegaBasic always read / writes floating point values using the precision and representation used internally by the running version of MegaBasic. To allow different precisions of MegaBasic to access the same data files, **PARAM**(11) has always been available to control the precision assumption used when performing floating point file transfers.

BASIC87 can read / write floating point values in any **BCD** precision from 6 to 18 digits, just like the **BCD** versions of MegaBasic. It can also transfer both **IEEE** double-precision (standard) and **IEEE** single-precision formats. This can be done by setting **PARAM**(11) to one of the following values:

1	Selects single-precision format for all floating point transfers. This is a 4-byte representation that can store numbers with about 6.5 digits of precision, ranging from 8.34E–37 to 3.37E38. A small conversion penalty is involved for each value transferred.
2	Selects the standard double-precision format used internally to hold and process **IEEE** floating point numbers. All 8 bytes of the **IEEE** floating point number are transferred in this format to maintain full precision. This is the fastest format to transfer **IEEE** floating point numbers between data files and your program.
6-18	Selects **BCD** floating point format with the precision indicated. Since only even numbers of digits are possible, odd values are rounded up. The extra time it takes to convert between **IEEE** and **BCD** format as values are transferred between memory and files should be considered when choosing **BCD** over binary transfers.

Values written in smaller precision or read in higher precision are rounded to the smaller destination precision. Values too small to represent in the target precision are set to zero, while values too large to represent will cause a numeric overflow error. Values written in higher precision or read from lower precision values are padded with extra zeros as needed.

Single-precision **IEEE** format is provided for applications that need to store low-precision numbers as efficiently as possible, and for accessing available data files written in that format. **BCD** format is supported to allow access to existing data files written by **BCD** versions of MegaBasic. A numeric overflow will occur if **BCD** values larger than 10^63 are written. **BCD** MegaBasic does not support the single / double precision **IEEE** format.

PARAM(11) normally defaults to the precision / format that the running copy of MegaBasic uses to represent floating point numbers internally. For example under 14-digit **BCD** and **IEEE** versions, **PARAM**(11) equals 14 and 2, respectively. However its default value can be modified permanently for any particular copy of MegaBasic using the **CONFlG.pgm** utility.

PARAM(4) returns the numeric precision / format used internally by the running copy of MegaBasic. Under **IEEE** versions it returns 2, and for the **BCD** versions it returns the **BCD** precision (as in the above values for **PARAM** 11).

Software/Hardware Performance

Performance is the real reason for using **IEEE** floating point representation. **BCD** add / subtract operations are very efficient and remain competitive even against the 80X87 processors. However, all other areas of floating point processing exhibit obvious gains when a math chip is used (e.g., multiply, divide and *especially* the transcendentals).

The degree of speed improvement you experience will vary with the *system clock* speed and *math chip type* being used (e.g., an 80287 is faster than an 8087, but slower than an 80387). There are several different brands of math chip and their performance varies widely. Because of this variation, certain internal operations may still run faster in software than with math chip assistance.

Math chip presence is automatically detected by MegaBasic at start up. If not present, all operations are performed strictly in software. The use of the math chip can be disabled or enabled under program control so that you can test your software under both environments without having to physically remove the chip or run your tests on a different machine. To enable / disable the math chip, use the following statements:

PARAM(20) = 0	Disables all use of a numeric coprocessor.
PARAM(20) = 1	Enables an 80X87 coprocessor if present.

PARAM(20) can also be tested to determine if a math chip is being used. It will not, however, return 1 if it is present, so you need to test it for a non-zero value instead of one. This is because the value returned by **PARAM**(20) is a composite value that indicates the chip type (i.e., 8087, 80287 or 80387) and a measure of relative performance of the existing chip as compared with the current **CPU** speed. It is not possible to enable the chip unless one is actually present in the machine. **PARAM**(20) under **BCD** MegaBasic always returns zero and cannot be set to anything else.

The accuracy of both the hardware and software transcendental functions is very good: full 16-digit accuracy is maintained for all functions when using the math chip. The software transcendental functions return results within 15-16 decimals for better than 99% of all arguments supplied. **COS**() and **TAN**() return an occasional result good to only 14 digits (for less than 1% of all arguments). This reduction in accuracy occurs only for arguments that are far outside the primary function domain (i.e., 0 to 2 *pi* for trigonometric functions). In such cases, the argument itself is inherently less accurate, so the reduced accuracy from the function is not significant.

In order to *hide* round-off errors in the least-significant digits of displayed floating point values, numbers displayed in free-form format are shown rounded to 14 digits. You can use *E-notation* or other fixed-point formats to see more digits of precision than this if you need to.

Chapter 4

Representing and Manipulating Strings

MegaBasic possesses two fundamentally different data representations: numbers and strings. Numbers and their associated operations are fully described in Chapter 4, Section 3. Strings are series of adjacent characters (8-bit bytes) used to represent anything from text to integers to arbitrary binary information. Their representation and manipulation is fully discussed in this section which is grouped into the following categories:

String Constants	Fixed strings for use in display, input or manipulation. Characters and the ASCII characters set is also covered.
Simple Variables	Character strings that can be altered during program execution.
String Arrays	Ordered sets of variable strings organized by one or more dimensions.
String Expressions	Phrases for computationally transforming and combining string objects into new strings. All string operators are discussed.
String Indexing	Notational conventions for extracting and accessing sub-sections of larger strings.
String Functions	User-defined and built-in symbols for combining and transforming strings.

Most typical business application programs spend much of their time dealing with strings: word processing, mailing lists, report generation, command processing, record processing, and formatting to name a few. Strings can represent binary information, text, packed numbers or virtually any other data representation. MegaBasic has a carefully chosen set of operations which when used in combination can efficiently perform all string operations supported by other high-level computer languages (such as PL / 1) with exceptional string handling facilities. Becoming fluent in MegaBasic string handling concepts can greatly simplify many of your non-numeric data processing applications.

Section 1: Characters and String Constants

The smallest quantity of information that can be represented or processed by a computer is the bit, an abbreviation for *binary digit*. One bit can only represent two values, one and zero, with which we can associate meanings such as: *on/off; true/false, yes/no, in/out, black/white, full/empty,* etc. However if we combine two bits together, a total of four values can be represented using all the possible combinations (i.e., 00, 01,10,11). Each additional bit double the number of possible combinations that can be formed, and hence the number of states that can be represented by the group.

By grouping 8 bits together as a unit, we can express 256 values, one for each of the possible combinations. These 8-bit units, called *bytes,* are perfect for representing characters because their 256 possible values is sufficient for assigning a different value to each letter, each digit, each of the various punctuation marks (e.g., ?!@#$%^&*()<>,.:“’’;[]), and still have many left over for special purposes, such as carriage returns, spaces, form-feeds, etc.

In order for such a character set to really be useful, everyone who uses it must agree on the same characters for the same 8-bit values. After all, when you print the letter Q on one device it should also be the letter Q on some other device. Therefore a standard called the *ASCII character set* has been assigned to the series of 8-bit values so that independently developed computing machinery can communicate characters with one another.

Actually, there are a number of different *standard* character sets that exist, but *ASCII is* the most commonly and widely accepted standard. Appendix D, Section 3 contains a table of all ASCII characters alongside their corresponding 8-bit values (in binary, decimal and hexadecimal). The **TRAN$( )** string function (Chapter 9, Section 3) can convert strings of characters from one character set to another, should the need arise.

Awareness of the ASCII character set is of central importance when you compare strings with one another using MegaBasic statements. In order to sort strings, for example, you need to know if one string is *less-than, greater-than* or *equal-to* another string. The notion of above and below depend of the internal values of characters rather than the characters themselves.

However, individual characters cannot convey very much information. As you read this sentence, notice that you are not reading one character at a time, but reading words or even phrases of words as indivisible units of information. Characters are important but larger *chunks* of information are much easier to handle, move around and manipulate. Therefore in a programming language, character information is processed in multi-character chunks called *strings.*

A string is a sequence of zero or more characters (8-bit bytes) treated as a single data object. As with numerical quantities, strings may be expressed as constants, variables, arrays, functions and string expressions. For example the string constant *“This is a String”* is a string with 16 characters. The quotes are used to clearly separate the string characters from those around it but are not actually part of the string. Without the quotes, it would be difficult (if not impossible) to tell which characters are in the string and which characters are outside the string.

String constants typed into MegaBasic programs are always delineated using two double quotes ("...") or two single quotes ('...'), making it possible to include either quote character (but not both) within a string constant. You must type quotes around string constants within MegaBasic programs, but when a string is typed as input to a program request you never put quotes around it unless the quotes are part of the string itself. It would be very restrictive and cumbersome if you had to surround all your typed input with quotes. The following MegaBasic program statement illustrates how you would print a message on the screen using a string constant:

```
PRINT "This message goes on the screen without quotes."
```

A string can have *zero or more* characters and although it may seem that a string with *zero* characters would have little use, it actually occurs in string applications just like the number zero occurs in numerical applications. Such an *empty* string is called a *null string. A* null string constant in a MegaBasic program is typed simply as two quotes with no characters in between (i.e., "" or ''). Suppose that your program requests string input from the keyboard and the operator types in nothing. Your program can simply compare the input received with a null string and take the appropriate action. Remember that spaces, like those between the words in this paragraph, are not null strings but actual characters in a string. For example the string constant " " is a string consisting of three characters, all spaces.

If you ever forget to include the terminating quote (" or ') at the end of a string constant, MegaBasic will automatically place one at the end of the line. This can be convenient when you are typing a string constant as the last item of a line, since the second quote need not be typed. However, if other terms or statements follow a string constant on the same line, omitting the final quote causes all following characters to be included as part of the string constant. Therefore, MegaBasic informs you that it added a missing quote on programs edited or inserted into the program.

String constants are used in programs to represent fixed character sequences (usually text) which are manipulated with other strings to form string results. This is analogous to the use of numeric constants (Chapter 3, Section 2) in programs as fixed quantities. Since string constants are typed from the keyboard, only the printable subset of ASCII characters can be placed in them. If you type any control codes (values 0 to 31), the MegaBasic line editor picks them up and uses them for various editing functions and they never get into the string constant. However, string constants are not the only way to express strings, as you will see in the sections that follow.

Section 2: String Variables

Character strings from zero to 65502 bytes long may be stored in string variables for later retrieval by name. A variable name may appear anywhere that string data is acceptable. By merely referring to any string variable by name, its entire contents are immediately made available. String variable names must begin with a letter (A-Z), usually end with a dollar sign ($), and contain any number of intervening letters (A-Z), digits (0-9) or underscores (_). Names are discussed in-depth in Chapter 1, Section 5. The following examples illustrate how and how not to spell string variable names:

Legal String Variable Names	**A2$, S$, WORD$, LONG_STR$, LINE2$, HEADING**
Illegal String Variable Names	**5CHARS$, $A, TEXT%, TITLE!, STRING#, TYPE$$**

Using a string variable name wherever string data is expected gives access to the data stored in the variable. Assigning string data to a string variable replaces its previous contents with the new string, a process that can be performed by assignment statements (Chapter 5, Section 2), **EXAM** statements (Chapter 7, Section 3), **INPUT** statements (Chapter 7, Section 1), **SWAP** statements (Chapter 5, Section 2) and **READ** statements (data Chapter 5, Section 1, file Chapter 7, Section 2). For example, the following short program stores a message into a string variable named **LINE$** and then prints the contents of **LINE$** on the screen:

```
10 LINE$ = “This message is stored in LINE$”
20 PRINT LINE$
```

Unlike string *constants,* the characters stored in string *variables* may assume the full 8-bit ASCII character code range from 0 to 255. String variables in many computer languages cannot store the entire range of 8-bit values (0 to 255), but the full range is vital to many non-text applications. Bit-strings are a typical example of such an application, an important tool which is described later in this section.

String variables in MegaBasic may be defined to hold any length string up to 65502 characters, as long as the available memory in your machine is sufficient. However since strings are variable length objects, MegaBasic sets aside a memory area for each string variable large enough to hold any string up to its defined maximum length. Unless you explicitly define the maximum size for a new variable, MegaBasic will automatically assign a maximum size of 80 characters, by default. You may assign your own maximum string size using a **DIMension** statement like this:

```
DIM LINE$(50), BUF$(9999), CHAR$(1)
```

where **LINE$** may store 0 to 50 bytes, **BUF$** may store 0 to 9999 bytes and **CHAR$** can store only 1 or 0 bytes. The same **DIM** statement can define one or more strings by listing their definitions one after another, separated with commas as shown above. Both string and numeric (array) variables may appear in the same **DIM** statement.

Newly **DIMensioned** strings are filled to their maximum length with spaces (ASCII 32). This default may be altered at any time to any ASCII code from 0 to 255 using **PARAM(7)** in Chapter 9, Section 5. The 80 character default length of undimensioned string variables may be set to any value from 1 to 4095 using **PARAM(12)**.

Although default string variables are convenient, we recommend that you do not write large programs that rely on them because they are often too large for small string applications and this practice can complicate the test and debugging phase of developing such programs. For example, by merely misspelling the name of an string in some reference to it, MegaBasic will create a default string of that name if one does not already exist. You can turn off default string creation by setting **PARAM(12)** to a value of –1. Any subsequent references to new, **unDIMensioned** strings will cause an *Undeclared String Or Array Error,* helpful in locating unintentional default string creations.

If you want a new string variable to contain zero characters (a null string) from the start, simply assign a null string to it immediately after you create it, as shown in the example below:

```
DIM STRING$(1000); STRING$ = ""
```

This creates a string variable named **STRING$** which is initially set to contain a null string (), but has the capacity to hold up to 1000 characters.

DIMensioning a string variable already defined re-defines that variable to the new size specified. Such an operation is useful for releasing unneeded memory back to the system for further use, and to permit program control over the size of string and array variables. Since **DIMensioning** always re-initializes strings (with the default ASCII code), all previous contents of the variable are lost, as is also the case with numeric arrays.

String variable and function names do not have to end with a dollar sign ($), although this is a standard practice that makes the data type of string variables, arrays and functions more obvious when reading the program. Strings names can be declared in the same manner as numeric variables (integer and real). The complete set of rules and syntax for declaring names as string entities is presented on the next page. If, however, you are new to MegaBasic, we recommend that you name all string entities with names ending with a dollar sign ($) to avoid any additional complication during your initial efforts in learning MegaBasic. Later on, you can experiment with and take advantage of the other methods in MegaBasic for declaring string variables, arrays and functions.

Rules For Declaring String Names

All variables and user defined functions are, by default, floating point (real) unless you specify otherwise. To declare a specific name to be a string, you can end its name with a dollar sign ($), declare its leading letter as **STRING**, or by declaring it explicitly as **STRING**. Use the **NAMES** command to see what names are string, integer and real. The rules and syntax for type declarations are summarized in order of decreasing precedence as follows:

- Any variable or function name that ends with a dollar sign ($) will always name a string object. A *Type Error* occurs if you attempt to declare or **DIMension** such a name as real or integer.
- String arrays may be declared directly in DIMension statements, as shown in the following example:

  ```
  DIM STRING MSG(30,40), X(1000,50), BUFFER$(512)
  ```

  which declares **MSG()** and **x()** as string arrays and **BUFFER$** as a simple string variable. The words **STRING**, **INTEGER** and **REAL** cause all **DIMension** specifications that follow in the list to be string, integer or real variables, until another specifier is encountered.

- Specific names of variables and functions may be declared as **STRING**, **INTEGER** or **REAL** using **DEF** statements such as:

```
DEF STRING LINE(),MSG
DEF STRING FUNC UCASE(BUF$)
DEF INTEGER X,Y,V(),P
DEF INTEGER FUNC TOTAL(V1,V2)
DEF REAL A,B,ARRAY(),C
DEF REAL FUNC SUM(V3,V4)
```

  The empty parentheses () indicate names intended to be arrays. These declarations override any types specified by letter. A *Double Definition Error* results from declaring the same name with different types. This rule overrides any data type associated with the leading letter of such names (see below).

- You can declare the data types by *leading letter.* A name beginning with a declared letter will be become an object of the type declared. This is accomplished using a **DEF** statement such as:

```
DEF STRING "s-v, z", INTEGER "a, b, c, i-n"
```

  where the string constant s-v, z specifies that names beginning with the letters s,t,u,v and z will be strings and *a,b,c,i-n* specifies the leading letters of integers. The quotes are required, but commas and spaces within the quotes are entirely optional. Upper and lower case letters are treated as indistinguishable. A double definition error will occur if you attempt to explicitly declare the same letter with different data types.

- If none of the above rules apply, then, by default, the name will be assigned a floating point (real) data type and cannot be a string.

Section 3: String Arrays

Another type of string variable is the array, in which a group of string values can be stored under one name. String arrays are organized as an ordered set of storage locations, called array elements, that are identified by a position number within the ordering. For example **LINE$(0)**, **LINE$(1)** and **LINE$(2)** represent the first three string elements of array **LINE$.** Parentheses are used to indicate that **LINE$** is an array and serve to contain the position of the desired array element. The positions are sequentially numbered from zero up to the size of the array.

The 1-dimensional array **LINE$** above could act as storage for a list of lines of text, collectively representing a page of text, giving you direct access to each line on the page by its line (position) number. Suppose that we combine many such pages together into one string array for access by page (position) number. This is called a 2-dimensional array. By identifying the line and the page we can directly access any line in the *volume*. For example **LINE$(PAGE,ROW)** refers to line **ROW** on page **PAGE**, where **PAGE** and **ROW** are simple variables specifying the array element positions. By adding further levels to this idea, you can define and access string arrays with 3 or more dimensions. An **N**-dimensional array requires **N** position numbers, called subscripts, to uniquely identify an element position in the array.

DIMensioning String Arrays

In order for an array to exist it must be defined in your program prior to its use. The definition of a string array must include its name, a maximum position for each dimension subscript, and the string capacity of each of the array elements. Specify string array **DIMensions** just like numeric arrays except that you must include the maximum length of each array element as the last value of the **DIMension** list. Take the following 2-dimensional string array definition example:

```
DIM BUF$(7,20,16)
```

This defines a two-dimensional string with rows numbered 0 to 7, columns numbered 0 to 20 and individual string array elements having a capacity of O to 16 characters each. You must always refer to **BUF$** with a subscript list to indicate a specific array element. For example:

BUF$(i, j)	A correct reference to string element at row I, column J.
BUF$(i)	Too few subscripts is an error that stops the program.
BUF$	Omitting all subscripts is also a fatal error.
BUF$(i, j, k)	Too many subscripts is also an error.

If you specify the wrong number of subscripts in an array reference, as in the last three examples above, MegaBasic will report an *Array Subscript Error*. When accessing string array elements, specify only the array **DIMension** positions and leave off the length parameter, which is given only when **DIMensioning**.

You can **re-DIMension** the array at any time by re-defining it in another dimension statement. All stored strings redefined in this manner are erased after such an operation and re-initialized. Arrays can thus grow or shrink depending on your program

requirements. When arrays are made smaller the unused memory space is available to the system for other purposes. The following list summarizes some important aspects of using string arrays:

- An *Array Subscript Error* occurs if you attempt to access a dimension position outside its defined range or use the wrong number of subscripts when accessing it.
- A single **DIM** statement can define one or more arrays by simply listing their definitions one after another separated by commas.
- All array elements are initialized the same way as simple strings.
- You cannot assign the same string variable name to both a string array and a simple string variable. If you create a string array using the name of a simple string variable that already exists, the simple variable and is contents will be erased and the specified string array created under the same name.
- All string arrays must be defined explicitly, otherwise MegaBasic thinks they are simple string variables instead of arrays.
- Array subscripts which are given as fractional quantities are truncated to the next lower integer value (rather than rounded). For example **BUF$(3.723,0.201)** is treated as **BUF$(3,0)**.
- For the best performance, you should employ integer expressions and variables for array subscripts whenever possible. Floating point variables can be used, but they will be converted internally to integer representation. Such conversions are time-consuming by nature and best avoided if possible.

Since the dimensions of arrays can vary during the execution of your program, determining the current dimensions of a given array can be useful from time to time. The **DIM()** built-in function provides such information for any variable Chapter 9, Section 5). **DIM(S$)** gives the number of dimensions of the variable **S$; DIM(S$,1)** gives the highest position defined for dimension I of variable **S$**, counting the dimensions from left to right.

Maximum String Array Size

As in all computer languages, the amount of memory available to MegaBasic limits the maximum size of new arrays. Within this constraint, however, MegaBasic supports much larger arrays than most other microcomputer languages.

The number of elements in an array is computed by taking the product of the dimensions. For example the **BUF$** array of the previous examples has 168 elements, as computed by (7+1)*(20+1) = 8 * 21 = 168. One is added to each dimension to obtain the true position count of each dimension.

The memory space taken by a string array is simply the number of elements times the element size in bytes. The size of a string array element is its dimensioned length (i.e., the last number in its **DIM** specification) plus 2 (for internal overhead). Hence the total memory required by the **BUF$** array is 168 * (16+2) = 3024 bytes.

You can create as many variables and arrays as you like as long as their combined storage requirements do not exceed the installed memory in your machine. **MS-DOS** based systems are limited to 640k bytes of addressable memory (16 megabytes in *Extended MegaBasic).* The **FREE()** function, described in Chapter 9, Section 5, provides information about the current memory available so that your program can automatically limit the size of new arrays to match available resources.

MegaBasic supports arrays with any number of elements, as long as no one subscript is higher than 65534. For example, DIM BUF$(99,99,99,4) dimensions BUF$() to three dimensions of 100 positions each, or 1,000,000 elements total (6 bytes / element). To dimension BUF$(1000000,4) is not allowed, because the dimension extends higher than 65534. For performance and other reasons, one restriction applies to string arrays larger than 65534 total elements: *pointers to array elements* can only access the first 65534 elements. Pointers to *arrays* (rather than to array elements) are unrestricted.

Obviously, you must have enough memory to support whatever arrays you actually dimension, which tops out around 540k in a 640k DOS machine. Protected-mode versions of MegaBasic, such as Extended MegaBasic, have no 640k limitation and support massive arrays of up to 16 megabytes.

Section 4: String Operators and Expressions

Strings are manipulated and processed by combining them in phrases called string expressions, similar to numeric expressions. String expressions permit you to specify a string as a combination of other strings and are formed from string symbols and string operations. Although the notation of string expressions looks similar to numeric expressions, their operation is totally different. The example below combines two strings together into a string result:

This string expression:	**"ABCDE"+"12345"**
evaluates to this result:	**"ABCDE12345"**

As you can see, the plus sign (+) has a different meaning depending on whether it is being applied to numbers or to strings. A plus operator used with strings is called a concatenation operator, because it is used to connect or concatenate two strings into a longer string.

String symbols used in string expressions include string constants, string variables, string functions (both user-defined and built-in) and string sub-expressions. A sub-expression is actually a portion of a larger expression that has been surrounded by parentheses, grouped as a computational unit.

String operations, called string operators, are of two types: unary and binary. Unary operators act on a single string to form the result string. For example the **NOT** operator preceding a string (e.g., **NOT Z$**) will produce a result string of the same length but with each byte logically complemented. Binary operators however act on two strings situated on either side of the operator to combine them in some fashion producing a result string, as in the concatenation operator demonstrated above.

MegaBasic evaluates string expressions from left to right accumulating the results from each operation as it goes. The various string operators are not however applied with equal priority. Take for example the following string expression involving concatenation (+) and string repetition (*) factors:

```
"ABC"*2+"xyz"*3
```

This expression repeats *ABC* twice and concatenates it to *xyz* repeated 3 times (i.e., *ABCABCxyzxyzxyz).* Since the string factors (*) are evaluated before the concatenation, we say that such factors *take precedence* over concatenation (just like their numeric multiplication takes precedence over addition). Similarly, all string operators have been assigned to a priority scale that controls the order of operations when several precedence levels are present in the same expression, much like the numeric operator precedence ordering.

When required, you can override these default priorities by surrounding any operation by parentheses to force its evaluation in the order of your choice. The example below illustrates a situation where concatenation (+) is performed prior to a string repetition factor:

```
("ABC" + "xyz") * 5
```

The concatenation in parentheses is evaluated first, followed by repeating its result five times. The table below lists the various string operators in order of decreasing precedence followed by a discussion of each.

String Operator Precedence	
12	Evaluatestring constants, string variables, string functions and sub-expressions.
11	String Indexing (Chapter 4, Section 5)
10	String Repetition Factors (*)
9	String Concatenation (+), String Subtraction (–)
8	String Comparisons **(= <> <= >= <> IN)**
7	LogicalComplement (**NOT**)
6	MIN, MAX
5	MATCH
4	Intersection (AND)
3	Union (**OR**)
2	Exclusive-OR (**XOR**), Equivalence (**EQV**)
1	Implication (**IMP**)

This ordering is similar to that of numeric expressions except that strings have some different operators. You can control the ordering of operations using appropriately placed parentheses. Be careful using complex string expressions in string comparison operations. The comparison operators are not really string operators since they produce a numeric result (i.e., integer 0 *for false, 1 for true).* They are included in the table above only to show their precedence within mixed mode expressions. It is the programmer's responsibility to ensure that mixed string and numeric expressions are sufficiently parenthesized to resolve any inherent ambiguities.

String Concatenation

The simplest of the string operations is concatenation (+), which merely appends two string operands together, end to end, in the order given. For example *ABCDE+12345 = ABCDE12345.*

String Subtraction

AB returns A$ with all instances of characters specified by B$ removed, for example:

Subtraction Operands	Result String
" $34,564,194.37- " $,"	**"345641 94.37"**
"string functions" - "aeiou"	**"strng fnctns"**
"this is a test string"-" "	**"this is a test string"**

Removing extraneous characters from strings is a frequently needed operation that is particularly tedious and slow using any other available means. This operator carries the same precedence as the concatenation operator (+). For example A$+B$–C$ is evaluated as (A$+B$)–C$. Note that the expression A$–B$–C$ is equivalent to (A$–B$)–C$ and to A$–(B$+C$).

One application of string subtraction is counting the occurrences of one character in a string. To do this very efficiently, the **LEN()** function is used which computes the length of a string. Using the **LEN()** function and string *subtraction*, the following example computes the number of spaces contained in A$:

```
LEN(A$) – LEN(A$–“ ”)
```

which simply computes the difference in length between A$ with spaces and A$ with spaces removed. Of course, this computation can be generalized to count occurrences of any character or set of characters. Without string subtraction, this computation would require a programmed loop that checks each character one at a time, taking 20 to 100 times longer.

String Repetition

Any term of a string expression may be repeated by following the term by a multiply operator (*) and a numeric expression **[e.g., ABC*(X+Y)]** . First the factor is evaluated **(X+Y)**, then the string is repeated by that many times. The repetition factor expression needs parentheses surrounding it only if it contains more than one numeric term, as in the example above. When only simple factors are used, no parentheses are required, as in the string expression:

```
A$*X+B$*37+C$*23.
```

Any complex string expression may be multiplied by enclosing it in parentheses followed by the desired multiplier **[**e.g., **(A$+STR$(N)+“XYZ”) * (R+2)]**. Compound nesting is permitted to virtually any depth. Typical applications of string multiplication include dynamic formatting of strings in print statements, high-speed graphics, and initialization of large strings.

Many computer languages provide string repetition as a separate function, which is not nearly as convenient or intuitive as the MegaBasic string multiply. Other computer languages include a separate function just to generate some fixed number of spaces (“ ”) in a **PRINT** statement. To do this in MegaBasic you need only to include a space multiplied the appropriate number of times (e.g.,“ ”*N) whenever you need it in any **PRINT** or other statement.

String expressions are always formed in MegaBasic’s control stack, which can rapidly overflow when compound repetition factors build up enormous strings that exceed the available memory space.

String MATCH Operator

A$ **MATCH** B$ compares A$ and B$ and generates a string of characters showing which bytes match (with ASCII 255) and which bytes do not match (with ASCII 0). If the two argument strings differ in length the longer one is truncated to the same length as the shorter one before the operator is applied. A$ **MATCH** B$ carries a precedence just above the Boolean AND operator.

This operator is useful for creating masks which may then be used for selective overlay and wild-card character matching algorithms. **MATCH** performs a process which would otherwise require a complicated loop of statements taking far longer to complete. It is useful to those requiring special assistance in pattern matching applications, and should be considered an advanced topic. No examples of its use will be given.

String MIN/MAX Operators

A$ **MIN** B$ and A$ **MAX** B$ are available as string operators which compare the corresponding characters of A$ and B$ (as in **MATCH** above) and return one string of the same length consisting of the characters selected by the operation (**MIN** or **MAX**). For example:

given:	**A$ = "012345"**	***then:***	**A$ Min B$ = "012210"**
	B$ = "543210"		**A$ Max B$ = "543345"**

Both Min and Max carry a precedence just above the Match operator described earlier. One application for **MIN** and **MAX** is character range restriction. For example: *A$ max " "*len(A$) will* force all *control characters* in A$ (ASCII 0-31) to spaces. This expression uses a repetition factor on " ", a topic discussed earlier.

Logical Operators in String Expressions

Logical string operators (**NOT, AND**, **OR**, **XOR**, **EQV** and **IMP**) perform processes similar to their function in numeric expressions, except that both operands and result are bit strings. A bit string is simply a character string that is being used or interpreted as a sequence of bits rather than as a sequence of characters. There is no physical difference between character and bit strings and MegaBasic considers both as simply strings from different points of view. There are always eight times as many bits as there are characters in a string, because each character within a string consists of 8 bits.

By providing a repertoire of operations specifically designed for bit manipulation, MegaBasic allows you to process character strings as bit strings. The logical string operators act on each of the bits in a string, or the corresponding bits in two strings. For example, **NOT** performs a logical reversal on each bit of the operand string following it (1s become 0s, 0s become 1s). Its result string is the same length as its operand.

The other logical string operators operate on two string operands, producing a string result which is a logical combination of corresponding bits in the operands. If the operand strings differ in length, the longer of the two will be truncated to the same length as the shorter string before actually combining the operands. The same set of logical operators already described for numeric operations (in Chapter 3, Section 5) are also supported for bit string operations.

Each of the logical operators is defined in the table that follows. To illustrate how they work, we will show the effect of each operator on all the possible combinations of two bits (i.e., 00, 01,10,11). It is important to understand that logical operators combine all of the corresponding bit-pairs of two strings (except for **NOT**) which means that one logical operation is performed for every result bit. It is this simultaneous combination of all bits of bit strings that gives these operators their speed and power.

Left Operand	0011
Right Operand	0101
NOT (right operand)	1010
AND	0001
OR	0111
XOR	0110
EQV	1001
IMP	1101

The zeros and ones are used only to illustrate bit values, but in actual practice, the operands and the result are all (bit) strings.

As with numeric expressions, there are many different ways in which to express a given logical combination. A number of equivalent logical expressions are described below to further illustrate the logical operators as they are used in actual practice. Given that A$, B$ and C$ contain bits strings of equal length which will be used as terms in the various examples below.

(a)	A$ XOR B$	(A$ AND NOT B$) OR (NOT A$ AND B$)
(b)	A$ EQV B$	(A$ and B$) OR (NOT A$ AND NOT B$
(c)	A$ IMP B$	NOT A$ OR B$
(d)	NOT (A$ AND B$)	NOT A$ OR NOT B$
(e)	NOT (A$ OR B$)	NOT A$ AND NOT B$
(f)	A$ AND (B$ OR C$)	A$ AND B$ OR A$ AND C$
(g)	A$ OR B$ AND C$	(A$ OR B$) AND (A$ OR C$)

Examples (a) to (c) illustrates how you would compute the same result of the **XOR**, **EOV** and **IMP** operators using only **NOT**, **AND** and **OR**. As you can see, considerable effort is saved by using **XOR**, **EOV** and **IMP** when their particular computation is required. Examples (d) and (e) are instances of DeMorgan's Law, which is a rule for logically converting **ANDS** to **ORS** or **ORS** to **ANDS** using **NOT**. It is useful for reformulating logical expressions into simpler forms. Example (f) shows how the logical expansions of **AND** and **OR** terms is performed.

As we shall see, bit strings and their associated operations are ideally suited to applications involving the processing of sets. A set is a collection of related items, such as the set of all experiments for which we have data, or the set of all employees in a data base. A given set must define which members are present in the set and which members are absent. Suppose that we have a set of employee records that can possess up to 1000 members, numbered 0 to 999. Further, let us suppose that we wish to extract some subsets, like the set of employees which are managers and the set of employees that earn more than $50,000 per year.

Each of these subsets can be efficiently represented in your program as bit strings, in which each potential set member is assigned a bit position within a bit string. If the bit corresponding to a particular member is a one (1), then that member is present in the set; absent members are similarly marked with zero bits. Letting EMPLY$ be our

employee set, MGR$ be the manager set, and RICH$ be the set of employees making more than $50,000, the following logical string expressions may have some use:

Logical Expression	Resulting Set
MGR$ and RICH$	Set of managers who make more than $50,000
MGR$ and not RICH$	Set of managers who make less than $50,000
EMPLY$ and not MGR$	Set of employees who are not managers
RICH$ and EMPLY$	Set of employees who make more than $50,000
RICH$ xor MGR$	Set of employees who are either managers or those who make more than $50,000 but not both

Since strings can be as large as 65502 bytes, you can represent sets with as many as 524016 possible members, one for each bit in the string (given enough memoryand/or operational stack space). Sets are very general data structures which can be applied in countless ways, and because the individual set operations are provided in the basic instruction set of MegaBasic (as the logical string operators), they execute extremely fast.

Many applications in set processing and systems programming work require the ability to turn bits on (set to 1), turn bits off (reset to 0) and flip their state (change 1s to 0s, 0s to 1s), without affecting the other bits in the bit string. This can is done by applying a bit selection string called a *mask*, which controls which bits to change and which bits to leave unaffected. Given an arbitrary bit string named BITSTR$ and a selector bit string named MASK$ containing 1's for selecting bits and 0's for protecting bits, the following expressions may be used to selectively alter bit strings:

Purpose	Expression	Example
Turning bits ON	**BIT$ OR MASK$**	**0101 or 0011 = 0111**
Turning bits OFF	**BIT$ AND NOT MASK$**	**0101 and not 0011 = 0100**
***Switching* bits**	**BIT$ XOR MASK$**	**0101 xor 0011 =0110**

Other useful applications for bit vector operations include the following conversion from lower case to upper case. It turns out that if you set bit5 of a character to the logical combination of (**NOT** bit6 **AND** bit5) then the resulting character will be upper case (see an ASCII code chart to verify this as an exercise). This operation can be performed on an entire string using the following string assignment statement:

```
U$ = NOT ROTAT$(L$ AND CHR$(64)~LEN(L$),1) AND L$
```

where L$ is the original string, U$ is the upper case result string, and LEN(), CHR$() and ROTAT$() are string functions described in Chapter 9, Section 3. A similar statement may be implemented to convert from upper case to lower case. If often required within your program, this is best programmed as a user-defined string function (one-line function).

MegaBasic also includes several string functions designed specifically for bit string processing. See Chapter 9, Section 3 for complete information on **ROTAT$**, **BIT, ORD** and **CARD**, which are briefly summarized below:

ROTAT$()	Rotates a bit string, left or right, by N bit positions.
BIT()	Converts between numbers and bit strings.
ORD()	Locate the first 1-bit within a bit position range.
CARD()	Counts the number of 1-bits within a bit position range.

String Comparison Operators

Comparison operators are different from all the other string operators in that they give a logical result instead of a string result. When you compare two strings, you are looking to see if some relationship between the strings is **TRUE** or **FALSE**. For example you may want to test whether one string is equal to another string. The equality comparison returns **TRUE** if they are equal and **FALSE** if not equal.

By convention, MegaBasic (like many other computer languages) represents logical values with numbers: 1 means **TRUE** and 0 means **FALSE**. These values of 0 and 1 are internally represented in integer format instead of floating point, because integers can be processed significantly faster. Although logical values (**TRUE** and **FALSE**) are primarily used in **IF** statements and **WHILE/REPEAT** loops to decide what the next step of the program should be, you can also specify a string comparison anywhere else that a number is expected.

Strings can be compared using the same set of comparison relations that are provided for comparing numbers. Each comparison operator compares its operands and returns **TRUE** or **FALSE** (represented by an integer 1 or 0) to indicate the outcome of the comparison. Both operands must be of the same data type (attempting to compare a number with a string results in a *Data Type Error).* All the comparison operators are described in the table below:

Equal	**=**	Returns a **TRUE** if the left and right operands are exactly equal, and **FALSE** otherwise.
Less	**<**	Returns **TRUE** if the left operand is less than (below) the right operand, and **FALSE** otherwise.
Greater	**>**	Returns **TRUE** if the left operand is greater than (above) the right operand, and **FALSE** otherwise.
Below or Equal	**<=**	Returns **TRUE** if the left operand is less than or equal to (not above) the right operand, and **FALSE** otherwise.
Greater or Equal	**>=**	Returns **TRUE** if the left operand is greater than or equal to (not below) the right operand, and **FALSE** otherwise.
Not Equal	**<>**	Returns **TRUE** if the left and right operands are not exactly equal, and **FALSE** otherwise.
Subset	**IN**	Returns **TRUE** if there are no 1-bits in the right operand that are not also 1-bits in the left operand. This is really a bit-string operator.

When strings are compared, the ASCII codes of corresponding characters are compared from first to last until a difference is detected or the end of either string is encountered.

Strings are equal only if all characters are identical and both strings are of equal length. If one string *runs out* before a difference is encountered, the longer string is taken as *greater than* the shorter string. The following pairs of string constant comparisons illustrate some of the subtle properties of string comparisons:

"AB" < "ab"	Upper case letters are assigned to a lower set of ASCII codes than the lower case letters. If you want a comparison in which upper and lower case letters are treated the same, you should convert the letters of both strings to one case before comparingthem.
" " > " "	Spaces are *greater than* a null string. In fact, all strings are greater than a null string, except another null string. Null strings do not have an official ASCII code because it is not a character. But MegaBasic internally assigns the value –1 to a null string for convenience and continuity.
"aa" < "aaa"	These strings are not equal because they differ in length. In such a case, the shorter string is less than the longer string. Strings must be identical in all respects to be considered equal.
" 25"<"–25"	This is true because a space character (" ") has a lower ASCII code than a minus character (–), which illustrates how strings of numbers do not necessarily compare the same way as their corresponding numeric comparison (i.e., 25 > –25).

It is important to remember that string comparisons give logical (0 or 1) results which may be used anywhere that numbers are permitted, rather than string results like the other string operators. Comparisons of expressions are supported as well comparisons of simple values. The scale of operator precedence becomes important in such comparisons to permit expressions involving arithmetic, logical and comparison operators with little or no need for parentheses to group the various sub-operations. For example the expression X + A$=B$ computes X+1 if A$=B$, or X+0 if A$<>B$. String comparison operators always take precedence over arithmetic operators.

Exercise care when complex string expressions are supplied as comparison operands. String operators look similar to numeric operators but their actions are totally different. If you are not sure how MegaBasic will evaluate certain combinations of operators, you can always supply extra parentheses to clarify and enforce the exact meaning that you desire.

Changing the Collating Sequence

Although the ASCII character set was originally designed with string sorting and comparisons in mind, you may occasionally encounter applications requiring string sorts and comparisons based upon a different character ordering or collating sequence. MegaBasic accommodates this with the translate function (**TRAN$** in Chapter 9, Section 3), a general purpose character conversion function which can map any character to any other character throughout a string. Strings to be compared using a non-ASCII ordering are first translated to the new character set and then compared normally, as described above.

Bit-String Comparisons

If your application is using bit-strings, one common operation you may need is a test to determine if one set is a subset of another set. The IN operator performs this function which, for the expression A$ IN B$, returns **TRUE** (I) if every bit position in A$ that

contains a 1 is also a one in each corresponding bit position of B$, and returns **FALSE** (0) otherwise. In terms of operations on sets (represented by bit strings), A$ IN B$ tests to see if the set A$ is a subset of set B$. Like all other string comparisons, IN returns an integer result, rather than a string, where 1 means true and 0 means false.

If B$ is longer than A$, only the portion of B$ equal in length to A$ is compared. If A$ is longer than B$, A$ IN B$ can only be true if all the extra characters of A$ have all zero bits [i.e., bytes containing **CHR$(0)**] and all the others are **IN** B$. IN is a bit-string operation generally used in combination with other bit operations, including: **BIT()**, **ORD()**, **CARD()**, **ROTAT$()**, **NOT, AND**, **OR**, **IMP, EQV** and **XOR**.

Section 5: String Indexing and Substrings

It is often desirable to *access* portions of strings, called substrings, rather than the whole string. Most programming languages implement such access through special functions like **LEFT$()**, **MID$()**, **RIGHT$()** and **SUBSTR$()**. MegaBasic uses a different method to access substrings, called indexing, that is easier to learn, executes faster, requires less typing and performs the job in a more general fashion.

By convention, we will refer to the left and right ends of a string (oriented horizontally) as the beginning and end of the string respectively. String indexing is based on the idea that each character in a string has a position relative to the beginning of the string. We will assign the first character to be in position 1, the second character in position 2 and so on to the end of the string. Any portion of a string can therefore be specified by a position range within the defined positions of the string. For example if A$ is our string and we wish to access positions 10 through 27, we would express this as follows:

```
A$(10,27)
```

As long as the length of A$ is 27 or more, this indexing expression accesses the 18 characters in A$ starting at the one in position 10. If A$ contains less than 27 characters, MegaBasic will access all characters from position 10 to whatever the length of the string. A null string results if A$ is less than 10 characters long. Any string constant, string variable, string function or sub-expression may be the subject of an indexing expression.

Variations on this theme provide several other modes to specify substrings in different ways having advantages over one another. Each of the string indexing modes is discussed in the table below. The examples shown in the table use the variable A$ to represent a general string expression to which the indexing expression is applied.

String Indexing Expressions		
Interval	**A$(I,J)**	A$(I,J) refers to the substring starting at position I and ending with the byte at position J.
Open Ended	**A$(I)**	Refers to the substring consisting of all bytes from position I to the last byte of the string.
Position & Length	**A$(I:L)**	Refers to a string of length L starting with the byte at position I. This is equivalent to A$(I, I+L–1) using the interval method. A null string results if L=0.
Right Length	**A$(:L)**	Refers to a string of length L taken from the end of A$. Equivalent to A$(LEN(A$)–L+1:L).
Single Byte	**A$(I:)**	Refers to the single character substring in position I of string A$. Equivalent to A$(1,I) or A$(I:1).
Last Byte	**A$(:)**	Refers to the single character substring at the end of A$. This follows from the preceding two indexing modes as a special case. This is equivalent to A$(LEN(A$)).

Given a string A$ (I,J), MegaBasic returns a null string whenever J is less than I (J=0 is permitted) or I is greater than the length of A$. Also if the substring specified exceeds the length of the stored string, only that portion which actually exists in the string will be accessed. For example if A$ contains the string This is *a String*, then A$(9,1000), A$(9:100) and A$(9) all refer to the same string: *a String.* An *Out Of Bounds Error* occurs if you specify a starting position less than 1.

Any string or string expression can be indexed, not just string variables. Just type your index expression in parentheses immediately after the string expression and upon evaluation, only the indexed substring of the expression result will be returned. Index expressions have a higher precedence than any of the string operators, hence you must surround the string expression to be indexed with parentheses if they contain multiple terms, for example:

```
(A$ + B$ – C$)(1,J)
```

Without the parentheses around the expression A$+B$–C$, only the last term of the expression (C$) would have been indexed. String constants can also be indexed, like any other strings, and doing so has some important applications. Consider the following example:

```
"JanFebMarAprMayJunJulAugSepOctNovDec"(1*3–2:3)
```

This string expression converts integers 1 to 12 into the corresponding names of the month (i.e., their names abbreviated to three characters). This process of decoding a number into some set of keywords or names is frequently required in interactive software and report generators of all kinds. Indexing a string constant lets you do this in one simple expression without any string variables or complicated loops to program.

Unlike the month abbreviations above, your keywords may not all be the same length, a property required by this indexing application. To remedy this apparent deficiency, insert some *padding* characters after each of the shorter keywords to force them all to the same length. Then, index the string constant using that length and remove the padding characters from the result using string subtraction (Chapter 4, Section 4) or the **TRIM$()** function (Chapter 9, Section 3).

This technique depends on all the keywords fitting into a string constant, which must itself fit within one program line (255 characters maximum). Longer lists of keywords must be stored in a string variable of sufficient length and indexed in a similar manner. You can also access longer lists by breaking them into several smaller string constants that reside on different lines. You would then have to **GOTO** the appropriate line before performing the indexed access as described above.

Indexing String Arrays

String arrays may be indexed by following the array subscript expression with a string indexing expression (a second set of parentheses). In such a case, you are gaining access to a substring in an array element of a string array. String array elements are always functionally identical to simple string variables in any context. For example:

```
CUBE$(1,J,K)(FIRST,LAST)
```

which specifies the string from position **FIRST** through **LAST** of the string element in row I and column J on level K. When using string arrays, take care to keep the subscript expressions and the index expressions separate in your mind as well as in your program.

Assigning Strings to Indexed String Variables

An indexed string variable may be the target of an assignment statement or any other operation that moves data into a string variable. However such an assignment normally only affects the indexed character positions within the indexed region specified and cannot alter the overall length of the string (an *exception* to this follows shortly). Strings moved into these positions are truncated (from the end) when too long to fit. Shorter strings

are placed left justified into the indexed area, replacing only those characters in positions required by the incoming string. Consider the following examples:

A$(1,J) = “string”	Input B$(1:N)	Read C$(:N)

In the first example, the string constant overlays the string contained in the string variable starting at position I on up to position J. If that region is too short then only the left-most portion that fits will be stored. If the region is longer than the assigned string then the right-most portion of the region itself will not be modified in any way. In the second example, the **INPUT** statement can only affect the N positions of B$ starting at position I. In the third example, the **READ** statement affects only the last N positions of C$. It is also important to remember that the indexed region of a string variable includes only positions that actually contain characters, and excludes any region beyond the current end of the string (i.e., you cannot alter positions beyond the end of the string or its length with indexed assignments).

Since the indexed assignment statement above does not fill out the entire indexed region when the string assigned is too short, another type of string assignment is provided for this purpose. By using == instead of = any positions to the right of the string that remain unfilled are set to spaces (i.e., the current *string fill character* set by Param 7). This assignment is described in Chapter 5, Section 2.

Another type of string assignment in MegaBasic lets you replace the entire contents of an indexed region with another string exactly. If the string and the region are different lengths, MegaBasic automatically shifts the characters that follow the region up or down to exactly accommodate the string so that it exactly replaces the region indexed. This method uses the := operator for the assignment instead of = or == and it is the *only indexed* string assignment that can affect the overall length of the string variable content. See Chapter 5, Section 3 for further information.

Extended String Indexing

Index expressions may be appended to any string representation, including another indexed string. This flexibility permits several layers of indexing to be applied to the same string, which can facilitate implementation of various hierarchical data structures stored in large string variables. For example:

```
A$(1,J)(R :L)(T)
```

Each indexing expression is evaluated from left to right and is applied as a simple indexing expression to the result substring of the prior indexing expression. Internally, MegaBasic arithmetically evaluates the series of indexing expressions as a unit and only then does it apply it to the string being indexed. This replaces many potentially time-consuming string move operations with a simple binary arithmetic computation that executes many times faster. At the cost of some arithmetic, this same example could have been done with a single indexing expression as follows:

```
A$(1+R+T-2,MIN(J,I+R+L-2))
```

Not only does this approach execute more slowly, but it is not at all obvious what is really going on. Extended string indexing simplifies certain kinds of operations but in the vast majority of applications simple indexing should be all that is necessary.

Be sure to specify each numeric value in an index expression using integer representation (rather than floating point real) wherever possible. Real expressions may of course be used in this context with entirely correct and identical results, but with a slower response. Index expressions are always in integer form internally, and MegaBasic will convert any real expressions encountered to integer representation every time they are evaluated. When a non-integral index value is evaluated (e.g., 3.721 or 9.834), MegaBasic reduces it to the *next lower* integer value (e.g., 3 or 8) rather than rounding it as some other programming languages do.

Section 6: String Functions

As we have shown, strings can be expressed as constants, variables and string expressions. However they may also be expressed as results of special procedures called functions. Functions are similar to array variables, in that they are referred to by name and include additional information which affects the value that they represent. The difference is that an array element merely accesses the value it holds, but a reference to a function invokes a process which computes the string symbolized by the function name. As with constants and variables, functions may be employed within string expressions to represent any of the strings being combined by the expression.

A function is a process that computes a result based upon data which you have communicated to it. To identify each process, functions are assigned names just like variables. To use a function, you merely type its name and its input data just as if you were typing an array name and its subscript list. For example, consider the following three function references:

TRIM$(L$)	**REV$(L$)**	**STR$(V)**

The first function, TRIM$, removes any leading or trailing spaces from string L$ and returns the intervening characters. The second function, REV$, returns the characters in L$ in the opposite (reverse) order. The third function, STR$, converts the numeric value of V into a printable string representation of that value. Functions are always of the same form:

```
<function name> (<argument list>)
```

Input information to the function is specified after its name, enclosed in parentheses, as a list of numeric or string values called an argument list. Each input value passed to the function is called an argument and is specified using any general expression. The values computed by these expressions are used by the function in forming its ultimate result. The number of arguments and their type (string or numeric) depends on the particular function being used. When more than one argument is present, they are separated from each other with commas. A *Data Type Error* will occur if you specify a number (or string) argument to a function that requires a string (or number) argument in that argument position.

MegaBasic possesses a library of over eighty built-in functions and also allows you to create your own functions, written in MegaBasic statements. Chapter 9 provides complete descriptions of all the built-in functions in MegaBasic and how to use them. Defining your own functions is a somewhat more advanced topic that is thoroughly covered in Chapter 8. Refer to these sections for more complete details. The built-in string functions are briefly summarized below:

Built-in String Functions	
***Len(S$)**	string length
Str$(X)	number to string conversion
***Val(S$)**	string to number conversion
Chr$(X)	ASCII code to character conversion
Chr$(S,Y)	ascending character series
Chrseq$(I,J,..)	multiple character ASCII sequences
***ASC(S$)**	character to ASCII code conversion
Trim$(S$)	leading / trailing space removal
Rev$(S$)	string reversal
Tran$(S$,T$,U$)	character translation
***Match(S$,T$,I)**	simple pattern matching
***Find(T$=S$,W)**	general string search
Min$(A$,B$,..)	minimum string among a list of strings
Max$(A$,B$,..)	maximum string among a list of strings
Inchr$(I)	raw character input
Reseq$(T$,,S)	cyclical string resequencing
***Bit(V$,I:Y)**	packing / unpacking value to / from bit strings
Rotat$(S$,I)	bit-wise string rotation
***Card(V$,I:N)**	counts the 1-bits in bit strings
***Ord(V$,I:N)**	searches bit strings for the next 1-bit
Collat$(V)	sortable string of a numeric quantity

The asterisks (*) indicate functions which return *numbers* (related to strings), rather than strings. Although such functions are associated with strings, a function which returns a string will be referred to as a string function in this manual unless otherwise noted. All the functions above are described in Chapter 9, Section 3.

There is no significant difference between *user-defined* string functions and user-defined numeric functions. String function names are formed exactly like string variable names (similarly with numeric functions). A string result is returned from a string function, a number is returned from a numeric function. Both may be defined with string or numeric parameters and formats include both single and multiple line. See Chapter 8, Section 3 for all the details on defining and using user-defined functions

Because **unDIMensioned** string variables are created by default when encountered for the first time, misspelled string function names will result in variables being created under those names. Such errors can be very difficult to diagnose because there is no way for MegaBasic to detect the error. For example, **STR$(I)** returns the string representation of the value of I, and **ST$(I)** returns the open-ended substring of variable **ST$** starting at position I.

MegaBasic provides three facilities to aid the discovery of misspelled names. First, the **NAMES** command (Chapter 2, Section 3) displays an alphabetical list of all user-assigned names in the program. You should check out unrecognized names that appear in this display. Mistyped variable and function names tend to be displayed in close proximity to the correct spelling of the user-assigned name, due to the alphabetical ordering. Second, the **XREF** command (Chapter 2, Section 5) displays all references to any name. Names with *only one* reference may be misspellings. Third, in program listings, MegaBasic shows user-defined names in *upper case* and MegaBasic reserved words in *lower case*. If you see one of your identifiers in *lower case*, then you better change it because it is a reserved word.

Chapter 5

Data Definition and Assignment Statements

This section provides descriptions of all statements involved in defining data structures and moving computational results between variables. See Chapter 2 for the description of the notation used to specify command and statement formats also employed in this section. See Chapter 9 for all information about the built-in MegaBasic functions. This chapter discusses the MegaBasic statements divided into the following categories:

Data Definition	Setting sizes, providing memory space, establishing initial values and data **types for** working variables and defining data constants for program operation.
Data Transformation	Moving data between variables, packing and unpacking bit-strings and performing computations.
Structured variable fields	Defines variable field structures for building complex record structures within string variables.
Pointer Variables	Methods for indirectly accessing variables, functions, procedures and line labels using information stored in other variables.

Section 1: Data Definition Statements

Data definition is the foundation of most computer programs and so we begin this section by describing the MegaBasic statements for specifying data, defining data types and allocating memory to data structures. These statements are summarized as follows:

DEF <vbl type specifiers>	Declares the data types and uses for variables, user-defined procedures, functions and structure fields.
DIM <vbl size definitions>	Creates string variables and arrays or changes their current size or number of dimensions.
RESTORE <vbl list>	Restores variables to their original contents at creation.
DATA <data list>	Defines string and numeric data for fast access during program execution.
DATA END	Terminates a logical group of **DATA** statements.
READ <vbl list>	Loads data from a **DATA** lists into a set of program variables.
RESTORE <pgm location>	Randomly accesses a **DATA** list for subsequent access by a **READ** statement.
ON...RESTORE <line list>	Selects one of many **DATA** lists on the basis of a computed index value.
LEN (<string vbl>)=<len>	Sets the length of a string variable or string array element without moving any data or changing memory allocation.
REM <descriptive text>	Descriptive comments to assist your program development and maintenance activities.

DEF [SHARED] *<name definition list>*

Defines data type conventions and assigns data types and uses to specific names for use in later program execution. **DEF** is special in that it *is not an executable* statement; it is a *declaratory statement* that is processed *before* program execution and ignored if encountered during execution. To define names that are accessible from other MegaBasic packages, you can specify **SHARED** immediately following the **DEF** reserved word of the statement, which affects all names defined by the **DEF** statement. The topic of **SHARED** names is described in detail later in Chapter 10.

There are five different **DEF** statement forms, one to define variables and data type assumptions, one for field structures, one for pointers, one for procedures and one for functions. This discussion is confined to variable and type assumption **DEF** statements. Defining field structures and pointers is described later in this Section. Procedure and function **DEF**initions are discussed in Chapter 8.

The **DEF** statement does not actually create variables, it simply attaches a *string integer or real type* to each variable name. When the variable is subsequently created by your program, its type is used to determine the proper amount of memory, its initial value and the methods to use during computations for internally manipulating the data representation contained by the variable. This greatly simplifies programming, because MegaBasic takes care of most of the details once you have specified the appropriate variable types.

Data types are are assigned to names on a name by name basis, or by **DEF**ining default type assumptions based on the first letter of its name. Names that end in dollar sign ($), percent (%) or exclamation point (!) are permanently assigned a string, integer or real data type, respectively, and an error results if you attempt to declare such names otherwise. The complete syntax for a type declaration **DEF** statement is as follows:

```
DEF [SHARED] <specifier>, <specifier>, <specifier>,...
```

where each <*specifier*> can be one of the following:

```
<type> <list of names>
<type> <leading letter list>
```

The <*type*> must be one of the words: **STRING**, **INTEGER** or **REAL**. The <*list of names*> *is* a list of one or more names, separated with commas, to be assigned the <*type*> specified. Names that will be used as arrays must be followed by empty parentheses ( ) to indicate that intention, for example X( ), Y( ) or Z( ). There is no comma between the <*type*> and its subsequent list. See Chapter 3 for complete information about **INTEGER** and **REAL** data types, and Chapter 4 for the STRlNG data type.

The <*leading letter list*> *is* a quoted string constant that assigns the specified <*type*> to all names that begin with one of the letters listed. It affects only those names that have not been specifically assigned a specific type in other **DEF** statements. Without any <*leading letter list*> specifier, all letters default to **REAL** to conform with *standard BASIC* type conventions and any undeclared leading letters will retain this **REAL** default. The quoted string consists of any combination of individual letters, in any order. A dash (-) between two letters specifies a range of letters (e.g., “a-z” means all letters). Upper and lower case letters are interchangeable; commas and spaces can be inserted anywhere for improving appearance and readability but are otherwise ignored. Letter declarations also affect the types of user defined functions with names beginning with those letters. Consider the following example:

```
DEF integer X, Y( ), Z, ”i-n”,
real INCR, M1( ), LVAL, N,
string BUFFER, ”s-v”
```

This **DEF** statement assigns an integer type to variables X and Z, and to array Y. Then the string constant ”i-n” defines variables with names beginning with any of the letters 1, J, K: L, M or N as integer variables. Then define INCR, LVAL and N as real variables and M1( ) as a real array. BUFFER is then declared as a string, along with any names that begin with the letters 5,l u or v.

You can simplify a complicated **DEF** statement by breaking it into several **DEF** statements, each defining a portion of the whole. The following set of **DEF** statements are equivalent to the prior example:

```
DEF INTEGER X, Y( ), Z
DEF INTEGER ”i-n”
DEF REAL INCR, M1( ), LVAL( ), N
DEF STRING BUFFER, ”s-v”
```

Variables which you do not explicitly declare in a **DEF** will automatically assume a type derived from the leading or trailing character of its name. Our example above explicitly declares several variables as real, because the leading letters in their names were declared as integer by the string constant ”i-n”.

DEF statement must appear as the first statement of the program line on which they reside and begin with the reserved word **DEF**. A program can have any number of **DEF** statements, which may appear anywhere in a program. When encountered during execution, **DEF** statements are skipped as if they were **REM**ark statements.

If you declare the same name or leading letter as having more than one data type, a *Double Definition Error will* occur. If you declare a name as one type and its leading letter as another type, the specific name declaration will win and no error will be reported. Declaring specific names always overrides letter declarations that would have affected those names.

DEF Statement Ordering

Unlike all other MegaBasic statements, **DEF** statements are *not executable,* i.e., they are ignored when encountered during program execution. All **DEF** statements in your program are scanned in the initialization phase of program execution that occurs just before actually beginning execution (you may notice a slight delay when you **RUN** a large program with lots of **DEF** statements on a slow machine). **DEF** statements are scanned, one by one, in the order in which they appear in your program, and the definitions they contain are compiled into the internal data dictionary that defines the meanings of all program symbols (i.e., variable and field names, line labels, function names and procedure names).

Because of the sequential nature of **DEF** statement scanning, the order in which the **DEF** statements appear in your program can affect what types are assigned. For example the statement **DEF FUNC** x defines an **INTEGER** function X if earlier in the program the statement **DEF INTEGER** "a-z" appears, but it defines an **REAL** function X even though it may be followed by **DEF INTEGER** "a-z". To avoid such ordering dependencies, it is best to explicitly declare function types in their **DEF** statements (e.g., **DEF INTEGER** FUNC x, etc.).

DEF statements can be entered as *direct commands* to affect subsequent type assumptions. Directly entered **DEF** statements must be the first statement of a direct statement command to be recognized, otherwise they are ignored (as they are in programs). Its effect lasts only until the next **RUN** command, which always clears existing data and type assumptions before running a program. Don't **DEF**ine procedures or functions in direct commands because their code is overwritten arbitrarily by the next direct command you enter.

DIM *<list of string and array definitions>*

Sets aside memory space for simple strings, string arrays and numeric arrays. **DIM** is an executable statement, so space for its defined variables will not be allocated until the **DIM** statement is actually executed in the running program. You must therefore make sure that your **DIM** statements are executed before the variables they define are used in any way by your program. If a variable already exists, **DIM**ensioning it will alter its size and re-initialize it as if it is being created for the first time. Consider the following example:

```
DIM ARRAY(12,15), STRING$(300), ROW$(100,20)
```

This **DIM** statement creates a two-dimensional variable named ARRAY, a simple string variable named **STRING**$ and a 1-dimensional string array named ROW$. *ARRAY will* have 13 rows numbered 0 to 12 and 16 columns numbered 0 to *15.* **STRING***$* can store any string value from 0 to 300 characters in length. *ROW$* can store 101 strings numbered 0 to 100, each of which may contain a string value of up to 20 characters.

Arrays may be **DIM**ensioned as string, integer or real (floating point). The words **STRING**, **INTEGER** and **REAL** can be placed in the VIM specification list to control the data type of subsequent variable definitions in the list, for example:

```
DIM INTEGER A(100),B(30),REAL C(75),D(150),
STRING LINE(80),Q(20,30)
```

In this example, arrays A( ) and B( ) will be integer arrays, C( ) and D( ) will be floating point arrays, LINE( ) and Q( ) will be strings. Notice that the type specifiers **STRING**, **INTEGER** and **REAL** affect all array definitions that follow them, until a different data type is specified by a subsequent type specifier. Data Type Errors will occur if you attempt to **DIM**ension names that end with $, ! or % after a conflicting type specifier **[e.g., STRING V%(50), INTEGER X! (100) or REAL A $(80)]**.

When arrays are **DIM**ensioned with their data type explicitly specified as described here, any data type assigned by previous **DIM** statements or **DEF** statements is overridden because the latest **DIM** statement always takes precedence. When no type specifier (i.e., **STRING**, **INTEGER** or **REAL**) is explicitly given in the **DIM** statement, the previously assigned type will prevail. The following program illustrates how this works:

```
Def integer X( ); Rem - X is declared an integer array
Dim real X(100); Rem - Create real array X, override DEF decL
Dim X(750); Rem - Change its size, but keep Its REAL type
Dim integer X(5); Rem - Now DlMension X as an integer array
Dim X(8000); Rem - Change its size, but keep its integer type
Dim string X(50); Rem - Now X is a 50 character string vbl
Dim X(256); Rem - and then change X(J to a 256 char string
```

Changing the type of a variable is only supported in the interpreter. The MegaBasic compiler does not support it, preventing such programs from being successfully compiled. Changing the number of dimensions in an array or their size is supported by both systems.

When arrays are created, each of their elements is given an initial value as part of the array creation process. Each of the elements in numeric arrays is set to zero. Each string array element is set to contain a string of spaces of the maximum string element length specified. Simple string variables are also initialized with a maximum length string of spaces. PARAM(7) may be used to change the fill byte of initialized strings to any other ASCII character if spaces are not desired.

See Chapter 3, Section 4 and Chapter 4, Section 2 for additional information about DlMensioning arrays and strings. Section 3 of this chapter contains a great deal of pertinent information regarding numeric types and how you go about choosing and specifying integer and real variables.

RESTORE*<list of variables>*

Restores (or reinitializes) the variables listed to their original contents at creation time (by default or DlMension statement). When you specify an array name, empty parentheses may be appended to it indicating that the name is that of an array. Simple numeric variables may be listed by name. Strings are always filled to their **DIM**ensioned size with blanks unless the default string initialization ASCII code has been modified using PARAM(7) (Chapter 9, Section 5). Numeric variables including arrays are filled with zeros, for example:

```
RESTORE X,Y,A( ),B$,R$( )
```

where the empty parentheses indicate array variables (optional). This example restores scalar variables X and Y to zero, initializes all elements of array A( ) to zero, and fills

scalar string variable B$ and all elements of string array R$( ) with spaces. Such use of *RESTORE* is unrelated to its use with **DATA** statements.

LEN(<string *variable*>) = <*number of bytes*>

Sets the length of the string variable specified to the byte length specified. The length may be any number from zero up to the **DIM**ensioned size of the variable. You may specify any unindexed string variable or string array element as the target string. This statement neither increases nor decreases the amount of memory available for any purpose. It merely revises the internal length counter associated with the string variable to indicate the length of its contents. Such a revision is useful prior to storing data into the string at absolute index locations, regardless of its prior contents (as in indexed assignments or binary &**READ**s). This statement is not commonly needed for most string processing requirements and exists to handle only those rare instances when nothing else will do.

DATA <*data list*>

Specifies a list of numeric and / or string data expressions separated by commas. **DATA** statements do nothing when encountered during program execution, as their purpose is solely to provide programs with built-in data values which may be assigned to variables by the **READ** statement (described next). Unlike most other BASICs, which only permit constants to be specified in **DATA** statements, under MegaBasic you can specify any **DATA** item as an arbitrary string or numeric expression.

DATA statements are unsuitable for large amounts of data or when the data will be revised during program execution. In such cases you should store the data on disk files for the purposes you have in mind. Take, for example, the following program:

```
10 Data –2, “string1”, 45, “string2”, 126, “string3”
20 Data 321, “string4”, 0, “string5” –99, “string6”
30 For l = 1 to 6
40 Read X,A$; Print X,A$
50 Next I
60 End
```

This program **READS** two data values, **PRINTS** them on the console, then repeats the process a total of six times. If it repeats more than six times, a **DATA READ ERROR** would be generated and the program would stop, because the **DATA** statements only specify six pairs of values. This program would run the same way no matter where the **DATA** statements were placed within the program source.

DATA statements in a program are best visualized as a sequence of statements separate from the rest of the program, but in the order they appear in the program. Data supplied in **DATA** statements is accessed sequentially from the beginning **DATA** statement or from a starting line number given by a RESTORE statement described below. **DATA** statements placed within a **THEN** or ELSE clause of an IF statement are inaccessible from **READ** statements and therefore serve no purpose.

DATA END

Specifies a logical end-of-data. Normally when a **READ** statement encounters the end of a **DATA** list, it skips ahead to the next **DATA** statement and continues on. However in many programs, you may have several groups of **DATA** statements that are not related to

one another. In such a *case, falling through* one group into the next is totally meaningless, but an error may not be reported, causing improper program operation that can potentially be difficult to find.

Therefore, to protect yourself from such problems you can place a **DATA** END statement after each *logical group* of **DATA** statements. If encountered while **READ**ing **DATA** statements, MegaBasic reports *a Missing Data Error* instead of **READ**ing ahead to the next **DATA** statement. Once encountered, your program has to reset the **DATA** pointer using a RESTORE statement before any further **DATA** can be **READ**. **DATA** END is not mandatory but MegaBasic provides it to assist the program development process.

READ *<list of data variables>*

Sequentially **READS** string or numeric data from the current **DATA** statement into the list of data variables. Data variables may include simple and array numeric variables and both unindexed and indexed strings. If the current **DATA** statement runs out of data before filling all data variables, then the next **DATA** statement in the program is automatically found and **READ**ing continues to the end of the variable list. Both **DATA** and **READ** lists are scanned in the order given, and all variable types must match the data items encountered. An error results from a type mismatch or from an attempt to **READ** past the last **DATA** statement in the program.

MegaBasic maintains an internal **READ** pointer to keep track of the current **DATA** position. This pointer is set to the first **DATA** statement when program execution starts. When more than one independent program or package is in memory (see Chapter 10), a separate **DATA** pointer is individually maintained for each. This permits each program to process its own **DATA** statements independently of the rest. **READ** statements cannot *access* **DATA** statements outside the package they reside in.

Whenever an error occurs during the processing of **READ** statements, the program location of the error reported by MegaBasic will always be the location of the **READ** statement involved. However, in many instances it is useful to also know the location of the **DATA** list being scanned when the error occurred. For this reason, the **STAT** command (Chapter 2, Section 5) always displays the current **DATA** scan location once a **READ** statement has been executed.

MegaBasic does not advance the **DATA READ** pointer until after each variable has been **READ** successfully so that, if a *Type Error is* encountered while **READ**ing, an error trap can can recover by **READ**ing the value into a variable of a different type.

RESTORE*[<label>]*

Sets the **DATA READ** pointer to the first **DATA** statement after a specified line number or line-label or to the first **DATA** statement in the program by omitting the <*label*>. This statement is used to reset the **READ** pointer or to provide random access to **DATA** statements. After being **RESTOR**ed either to the first or some other **DATA** statement, subsequent **READ** statements will *access* **DATA** statements sequentially from that point. Such use of **RESTORE** is unrelated to restoring variables as described earlier in this section.

ON *<expr n>* RESTORE*<line list>*

Evaluates the numeric expression and truncates the result to an integer that specifies a selection from the <*line list*>. This integer must be from 1 to the length of the <*line*

list>. The <*line list*> consists of a sequence of line numbers and / or line-labels, separated with commas, which specify program locations of **DATA** statements. The **DATA READ** pointer is set to the first **DATA** statement on or after the line selected via the integer. This is useful for selecting data through a multi-way computational decision, an extended form of the **RESTORE** statement above.

REM *<descriptive text>*

Everything from the **REM** keyword to the end of its program line is taken as a non-executing comment, including statement separators (; and \) or text which would ordinarily constitute valid executable statements. MegaBasic preserves the case (upper / lower) of all letters that follow the **REM** keyword.

REM statements provide additional information and guidance to the programmer during program development and later program maintenance. Well commented programs generally take less total time to construct and debug. A good practice to adhere to is to briefly describe each procedure or subroutine in its first line. To improve readability, MegaBasic automatically inserts a blank line between a group of **REM** statements and the programs that precede them when the program is **LIST**ed.

Some **BASIC** programmers have a tendency to be frugal in commenting their programs because of the amount of memory space they consume. This fear may be justified in some BASIC systems but not in MegaBasic. The **CRUNCH** utility supplied with MegaBasic will remove all **REM**arks and extra spaces from the program and create a new program file 20-50% smaller than the original. Using this utility, you can create a working copy for execution work while retaining its expanded counterpart, generously commented, for program development and maintenance. The runtime version of MegaBasic (**RUN**) performs this source reduction process automatically on every program it executes.

Section 2: Data Transformation and Assignment Statements

You can transform numerical and string information by combining one or more data items into a result using arithmetic, mathematical, logical, or other computational means. The result can then be *assigned* to a program variable or used in further calculations. You need to understand how to use numbers, constants and variables (Chapter 3), strings, string variables and string expressions (Chapter 4), and string and numeric functions (Chapter 9), to make full use of the data transformation statements summarized below:

<variable> = <exprn>	Evaluates a numeric expression and assigns the result to a numeric variable.
VEC <vector> = <vector exprn>	Assigns values to the elements of a vector computed from a vector expression.
<string vbl> = <exprn>	Evaluates a string expression and assigns the result to a string variable.
<string vbl> := <exprn>	Evaluates a string expression and then replaces the contents of a sub-region within a string variable with the result. The length of the target string changes to accommodate replacements of differing length.
<string vl> == <exprn>	Evaluates a string expression and assigns the result to a string variable sub-region, left justified and right-filled with spaces as needed.
<variable> += <exprn>	Evaluates a numeric expression and adds the the result to a numeric variable. Many other operators can also be used, such as *=, –=, /=, etc. These are known as *extended* assignment statements.
Assignments with expressions	Any kind *of* assignment statement can be *embedded* within larger expressions to store intermediate results into other variables while evaluating the expression.
SWAP <vbl pairs>	Exchanges values between pairs of variables. It supports integers, reals, strings and indexed strings, and is especially useful in sorting applications.
BIT(<vbl$>,<range>) = <exprn>	Evaluates a numeric expression and assigns the result to a bit subrange within a string variable (1 to 24 bits wide).

[LET] <*numeric vbl*> = <*numeric expression*>

Evaluates the numeric expression on the right of the equals sign (=) and stores the result into the numeric variable on the left. The prior content of the variable is lost. The variable may be a simple variable or a unique array element. This use of the equals sign has nothing whatsoever to do with its use in equality comparison expressions (e.g., If X=Y then...). The reserved word **LET** at the start of this statement is entirely optional, as it is with all assignment statements in MegaBasic.

Numeric variables and expressions are of two types: integer and real. Usually real values are assigned to real variables and integer values are assigned to integer variables. However MegaBasic does permit mixed-mode assignments of either type (i.e., integer=real, real=integer). Such assignments are inherently slower because of the necessary conversion of the assigned value into the numeric type of the receiving, performed automatically by MegaBasic.

When a real value is assigned to an integer variable, it is first truncated to a whole number by throwing away its fractional portion (if any). For example the values 3.76, 0.4 and –2.9 are stored as 3, 0 and –2 in the integer variable. Reals values can span a larger range than the 32-bit integer representation provided in MegaBasic, and a *Numeric Conversion Error will* occur if you attempt to store a real value below –2,147,483,648 or higher than +2,147,483,647 into an integer variable.

An integer value can be converted to a real value without precision loss in all cases except one. Integer values beyond 100 million (+–) cannot fit within 8-digit BCD floating point representation. Therefore the value is truncated to contain only the leading 8 decimal digits of the integer. Values between 100 million and 1 billion will always be within 9 of the actual value; values over 1 billion will be within 99 of the original integer value after being converted to real. We strongly recommend that you use versions of MegaBasic with 10 or more digits of floating point precision to avoid this conversion limitation. If your program never uses integers of this size, 8-digit MegaBasic can be used without any difficulties.

VEC <*vector variable*> = <*vector expression*>

Computes a sequence of values using a vector expression and assigns them to the elements of a vector variable. A vector is a sequence of numbers in an array that is accessed as a sequential list (i.e., without subscripts) and a vector expression is a computation that uses vectors as terms of the expression. Using the vector processing operations of MegaBasic you can perform thousands of arithmetic and mathematical operations without executing more than one or two MegaBasic statements. Vector statements are particularly useful for implementing matrix operations of any variety (Chapter 3, Section 7).

Extended Assignment Statements

When developing software under any language, you will frequently need to add a value to a variable, or multiply by a value, or change it in some manner that uses its current value to compute its next value. For example the statement ARRAY(I,J) 5 ARRAY(I,J)+l, will increment array element (i,j) by one. In order to speed up this kind of operation, an extended assignment statement is supported that is faster because you only have to specify the variable once, instead of twice.

The examples below show how this is done:

Standard Assignment Statement	Equivalent Extended Assignment
ARRAY(I,J) = ARRAY(I,J)+1	ARRAY(I,J) += 1
TOTAL = TOTAL – EXTRA	TOTAL–= EXTRA
PROD = PROD * MULT	PROD *= MULT
QUOT = QUOT / DIVISOR	QUOT /= DIVISOR
INTQ = INTQ DIV DIVISOR	INTQ DIV= DIVISOR
V = V MOD MODULUS	V MOD= MODULUS
BASE = BASE ^ EXPON	BASE ^= EXPON
VALUE = VALUE SGN SVAL	VALUE SGN= SVAL

When you specify an extended assignment operator (i.e., +=, –=,*=, /=, *mod=*, *div=*, ^=, *sgn=),* be sure that there are *no spaces* between the leading operator and the following equals sign (=). For example X + = 1 will be reported as an error, while X += 1 will be accepted.

When MegaBasic evaluates extended assignment statements, the expression to the right of the equals sign (=) is completely evaluated before being combined with the variable to the left of the operator, therefore:

the assignment:	**X *= Y + 1**
is evaluated as:	**X = X* (Y+1)**
instead of:	**X = (X * Y) + 1**

Extended assignments are especially useful when the target variable is an array with complicated subscripting, because of the time saved by omitting the second reference to the variable. You can use extended assignment statements *only with numeric varia~les;* extended string variable assignments are not supported. Furthermore, the numeric operators supported include only those shown above and do not include any logical operators (e.g., *AND, OR, XOR,* etc.) or comparison operators (e.g., = <~ > < etc.).

Assignments within Expressions

Assignment statements can also appear within any numeric or string expression. The value of an assignment expression is the exact value that was assigned to the variable. For example the two assignments:

```
A(i,j,k) = log(Z+5)/pi; Y = Sqrt(A(i,j,k)) * P
```

can be done as the following single statement:

```
Y = Sqrt(let A(i,j,k) = log(Z+5)/pi) * P
```

As you can see, *assignment expressions* let you assign a value to a variable and then go on to use the same value in another independent way, all as one step. Another way to look at this is that intermediate calculations performed within an expression can be stored in variables during the course of evaluating the expression.

Notice the **LET** reserved word in the embedded assignment statement. **LET** is normally an optional word to introduce an assignment statement. However in the context of an expression, **LET** is necessary to indicate an assignment statement is ahead. Without **LET**, MegaBasic would interpret the equals sign (=) as a comparison of two values instead of an assignment.

Proper use of assignment expressions can improve the performance of your software by reducing the number of statements and variable accesses required to process a given set of operations. Any kind of assignment can be used in assignment expressions, including numeric and string variable assignments, extended assignments to numeric variables (e.g., += –= *= etc.), and string replacement (:=) assignments. When string assignment expressions are specified, the string returned by the assignment is the string that was actually stored into the variable (which may be truncated to fit). The following example evaluates a string expression, stores it into A$, compares the value stored into A$ with B$ and increments N if they compared equal:

```
N += (let A$ = STRING_EXPRN$) = B$
```

Avoid assignment expressions involving the same variable more than once within the same expression, because the order in which terms are evaluated within a complicated expression is not necessarily well-defined. For example, the MegaBasic compiler evaluates the individual terms of certain expressions in a different but equivalent order for performance reasons.

You should also be aware that since the right-hand operand of AND, OR and IMP operators is sometimes left unevaluated, **LET** assignments in that term may not be performed. For example in the expression X *OR (LET Y-Z), Y is* set to Z only if X is nonzero. This is because the expression result is known to be 1 if X is nonzero, causing MegaBasic to skip the right-hand term (along with the assignment) to improve performance.

You have to surround an assignment expression with parentheses if it is followed by additional terms of a larger expression, because MegaBasic evaluates the assigned expression as far to the right as it can before actually assigning the result to the variable. Parentheses are not necessary if the assignment expression is the last term of an expression. Because of this, compound assignment statements can be written as follows:

```
Let A = let B = let C = let D = let E = X*Y/Z
```

The statement computes the value of X*Y / Z and then stores it into E, followed by D, then C, B and A. The leading **LET** is actually unnecessary, but was added merely for consistency and style. Be careful when using compound assignments so that the value is preserved as it passes from variable to variable. If integer and real variables are being assigned, there may be some truncation due to the implicit type conversions that occur when real values are stored into integer variables. Furthermore in compound string assignments, the string value will shrink as it passes from right to left if any of the assigned variables are not as large as the string they are receiving.

A useful routine that employs assignment expressions is given below. This routine displays the contents of BUF$ onto the screen and expands any tab characters (ASCII 9 codes) into the appropriate number of spaces.

```
100 Def proc EXPAND_TAB BUF$; Local l,J; J = O
110 While Let J = match(BUF$,chr$(9),1et 1 = J~1)
120 Print BUF$(1,J–1),tab((pos(0)+1) ceil 8),; Next
130 Print BUF$(J+1); Return; Proc end
```

Notice that the **WHILE** condition expression sets I,J to point to the next sequence of characters that does not contain a TAB character. This simplifies the loop down to only one statement (the PRINT) and improves the processing speed accordingly.

[LET] <*string variable*> = <*string expression*>

Evaluates the string expression on the right and assigns the resulting string value to the string variable on the left. The contents of unindexed string variables are totally replaced by the string value. For example, you can clear a variable of all characters by assigning a null string to it: A$–“”. If the string value being assigned is longer than the maximum length of the variable (as specified by the **DIM** statement), only the left-most portion that fits will be assigned.

If the target string variable is indexed, then only the sub-string portion indexed is affected. Strings longer than this region are truncated to fit and shorter strings are placed left justified within the sub-string field, without altering the remaining positions. If the string value is shorter than its indexed destination variable, the extra positions in the variable are left unmodified. Indexed string assignments to not change the overall length of the string contained in the target variable.

Care should be taken to avoid assigning expressions involving very long or many strings in one expression, since working memory proportional to the length assigned is required. The FREE(2) function (Chapter 9, Section 7) can tell you how much memory is available for evaluating an expression at any point. It should be consulted prior to evaluating potentially infeasible string expressions. Such expressions which would otherwise cause a *Scratchpad Full Error* should be broken up into smaller pieces that can be processed independently. Unless total memory is limited, only in extreme situations should such steps be necessary because the scratchpad has an approximate capacity of 56k bytes.

[LET] <*string variable*> = <*string variable*>

String assignments with only a string variable or indexed string variable on the right-side of the equals sign are assigned 2-3 times faster than the same string specified by a string expression. This is because in that case, the data is directly transferred to the receiving string variable without any intervening operations. For example A$=B$ is performed much faster than something like A$=“”+B$, since “”+B$ must be formed as a result in internal workspace prior to being assigned to A$. Furthermore, because this special case does not use any internal workspace, the size of the transfer is not restricted by any memory limitations. Hence variable-to-variable assignments up to 64k bytes in length are always viable and extremely fast.

[LET] *<indexed string vbl> := <string expression>*

This statement (notice the colon (:) before the equal sign) will place the string on the right into the indexed area on the left, just like the usual assignment. The big difference is that when the indexed area and the assigned string differ in length, the destination is expanded or contracted to fit the assigned string exactly (i.e., not just overlaid). For example:

A$(L:0) := "string"	Inserts "string" into A$ at position L
A$(L:5) := ""	Deletes 5 characters from A$ at position L
A$(K,L) := B$	Replaces the contents of A$(K,L) with B$
A$(K) := ""	Deletes all characters from position K to the end of the string. Equivalent to *A$ = A$(1,K–1)* except that no characters are physically moved.
A$(K) := B$	Replaces all the characters from position K to the end of the string with the contents of B$. Equivalent to *A$ = A$(1,K–1)+B$,* except that only the characters of B$ are actually moved.
A$(:0) := B$	Appends B$ to the end of A$ (i.e., replaces the null string at the end of A$ with B$). This is faster than the equivalent operation *A$ – A$+B$* because B$ is moved directly to A$ without any movement of A$.

In these examples, characters above the specified indexed region in A$ are moved appropriately and the length of A$ is adjusted up or down accordingly. If you specify a destination anywhere beyond the end of the destination string, the string on the right is simply appended to the end of the destination string on the left, for example:

A$(9999) := B$ **or** **A$(:0) := B$**	Appends B$ to A$ as long as the length of A$ is shorter than 9999 characters. It is equivalent to *A$ = A$ + B$,* except that B$ is appended directly to A$ without additional characters being moved.

If the operation implies a total result which is longer than the **DIM**ensioned limit to the destination string variable, all **result characters beyond that limit** are lost, as occurs with the standard string assignment statement. Although this assignment is designed to give the exact same result as if it were programmed using the standard concatenated assignment, it does it 2-10 times faster and in a much more obvious way as shown by the equivalent methods given in the examples.

This is a true string replacement operation that has many important applications. The following example illustrates how easy it is to substitute one string for another, of differing lengths, throughout a large body of text:

```
10 Rem—Substitute B$ for C$ everywhere in Text T$
20 I = 1; L=Len(C$); R=Len(B$)
30 I = Find(T$(1) = C$)
40 If I>0 Then [ T$(1:L) := B$; I = I+R; Goto 30]
```

Line 30 sets I to the location of the next occurrence of C$ in T$ using the FIND function (Chapter 9, Section 3). Line 40 performs the substitution and then repeats the process, which ends when no more occurrences are found. This is a near optimum replacement procedure and illustrates just one typical application of the string replacement statement.

[LET] <*string variable*> == <*string expression*>

When you assign (–) a string value to an indexed string variable or string field, it simply overlays the contents of the indexed string. If it is shorter than the indexed region then the right–most portion of the original string remains unchanged. The above string assignment, which uses the == operator, guarantees that the entire content is replaced.

The double equals sign (==) indicates that any unchanged portion of the target string will be filled with spaces (or other ASCII code controlled by PARAM*7)*. The length of the target string variable is never changed by this assignment statement. If an unindexed string variable is specified, it is handled like an all-inclusive indexed string variable. For example, A$==B$ is equivalent to A$(1)==B$. The == operator must be typed without any spaces between the two equals signs.

SWAP <*list of variable pairs*>

Exchanges the contents between each pair of variables listed. Both variables of each pair must be of the same type (string or numeric), but may contain a mix of string and numeric pairs. Each variable pair is exchanged independently of the others, for example:

```
SwapX,Y, A$,B$, R4(J),Z(I,K)
```

A SYNTAX **ERROR** results from an odd length list. A **DATA TYPE ERROR** results from attempting to *SWAP* strings with numbers or to *SWAP* an integer variable with a real variable. *SWAP* is limited to simple string or numeric data or array elements. You are allowed to *SWAP* an array element with a non-array variable. If the two variables are vectors preceded by the word VEC, the vector contents are swapped, for example:

```
SWAP VEC A(*),B(*)
```

See Chapter 3, Section 7 for complete information on vectors. SWAP statements are from 3 to 5 times faster than the usual assignment statement implementation (e.g., swapping X and Y using the sequence: T=X; X=Y; Y=7). SWAP is useful for sorting routines, where total sorting time can be cut substantially.

Any combination of indexed and unindexed string variables can be swapped. The length of indexed variables is unaffected by a SWAP. The length of an unindexed string variable is always set to the length of its new contents, limited of course by its capacity. The effect of swapping two arbitrary string variables is defined in three logical steps:

- Let A$ and B$ represent two arbitrary string variables. A$ and B$ may or may not be indexed.
- Let BUF$ be an unindexed string variable large enough to fully contain either A$ or B$.
- SWAP A$,B$ is equivalent to the following three steps: BUF$=A$; A$=B$; B$=BUF$.

In other words, swapping two string variables is implemented as if being performed with an intermediate string variable of sufficient size to effect an exchange of the two variables using only simple assignment statements. The rules governing string assignments and their effect on the target string are followed by the SWAP statement exactly. Bear in mind that SWAP uses no intermediate storage at all and executes much faster than assignment statement implementations, however its effect is identical.

Attempting to SWAP two overlapping regions of the same string variable will have unpredictable results. This is because MegaBasic SWAPS strings in place rather than using intermediate storage, as described above. The overlapping regions will not be properly exchanged, and your program will most certainly produce erroneous results.

BIT (<string vbl> [<bit range>]) = <expression>

Evaluates the non-negative numeric expression (on the right), converts it to an integer, and assigns the result to any bit subrange (up to 24 bits wide) within a string variable. The string variable reference may be indexed or unindexed. The optional *<bit range>* specifies a starting bit position and either an ending position or a number of bits to be affected. The following examples illustrate the various possibilities for specifying bit ranges:

BIT(TBL$)	Refers to the leading bit of byte 1 in TBL$.
BIT(TBL$,I)	Refers to bit I of TBL$.
BIT(TBL$:N)	Refers to the first N bits of TBL$.
BIT(TBL$,I:N)	Refers to N bits of TBL$ starting with bit I.
BIT(TBL$,I,J)	Refers to bit I through bit J in TBL$.

The **BIT** function is capable of accessing groups of 1 to 24 bits as a numeric unit. This provides very efficient utilization of memory when large tables of small positive integers are required. A string with a byte length of L bytes has bit positions from 0 to L~1, which can be as high as 524015 (for the largest possible string of 65502 bytes).

If you specify a bit range that lies partially beyond the last byte of the string, the bit range is truncated to fit the actual string. A bit range consisting of zero bits is specified if all bits in the range lie beyond the string. **BIT** requires that the actual bit string being accessed consists of 1 to 24 bits in length and lengths outside this range result in an *Out Of Bounds* error.

The first bit of the specified bit subrange always represents the high order bit of the integer bit sequence being accessed. The table below illustrates the relationships between **BIT** addresses, bit numbers within bytes, and character

Character Position	Bit Addresses	Bit Number In Byte
1	**0 to 7**	**7, 6, 5, 4, 3, 2, 1, 0**
2	**8 to 15**	**7, 6, 5, 4, 3, 2, 1, 0**
3	**16 to 23**	**7, 6, 5, 4, 3, 2, 1, 0**

For example bit4 of the 3rd byte of the string is in **BIT** position 19. The **BIT** positions in the table go by fours only to simplify illustrating the idea. **BIT**( ) may appear on either side of the equals sign depending on whether you are storing a value (left) or accessing a value (right). Other string functions that deal with bit strings include ROTAT$, ORD and CARD (Chapter 9, Section 3). Chapter 4, Section 4 describes various boolean (logical) operators which may be used combine and manipulate entire bit strings in one expression.

Values are stored modulo 2^width, causing reduction of values too big for the given length. For example **BIT(A$,95:8)–259** stores a value of 3 into the 8 bits starting at bit 95 of A$. This is because the value 259 is actually 9 bits wide and only the lowest 8 bits were stored.

The expression to the right of the equals sign must return either an integer or a real numeric result; a *Data Type Error* occurs if a string expression is specified. To obtain the best performance, you should specify integer numeric expressions (rather than real) wherever numbers are required on either side of the equals sign. Real values are internally converted to integer representation before they can be used for string indexing, array subscripts, bit positions and bit widths, and this conversion is inherently time-consuming. Integer values are already in the proper internal form for immediate application.

In addition to its ability at packing and unpacking small integer values in and out of string variables, the **BIT** statement has important applications in processing bit strings. Bit strings are ideal for representing sets, where bit(i) of the bit string is set to one if set element (i) is a member of the set and reset to zero if it is not a member. The **BIT** statement can, of course, set or reset any bit in a bit string. Additional information about bit strings can be found in Chapter 4, Section 4 and Chapter 5, Section 2.

BIT has additional applications in sorting. Multi-byte values written into bit fields are ordered high byte to low byte, permitting such fields to string sort correctly. This is the opposite order from the way multi-byte integers are transferred using FILL, EXAM, **READ** and **WRITE** statements. **BIT** is particularly useful for packing values into a string variable used to communicate the **CPU** register contents in **CALL** statements (Chapter 7, Section 3).

Section 3: Structured Variable Fields

Many applications involve complex record structures where a single data item may contain numerous related strings, reals and integers. To be able to move these collections of values *or field structures* around as a unit can greatly simplify programming and make processing more efficient. MegaBasic supports field structures as *data templates* into string variables, enabling you to assign *names* and *data types* substring regions within string variables which can then be accessed by name. The resulting *super-strings* can then be processed using the rich string facilities of MegaBasic just like other strings.

STRUCT *<field definition list>*

Defines a collection of field names, positionally related to one another, that are used to access data fields of any type by name from portions of string variables. Each field name is assigned a *data type* (i.e., **STRING**, **INTEGER** or **REAL**), a string *index position* and a field *length*. Fields are applied to string variables in a manner similar to string variable indexing expressions, for example:

```
VARIABLE.FIELD
```

This field reference extracts the region of VARIABLE that is indexed by the position and length assigned to FIELD by a **STRUCT** statement, as if the region was a variable with the data type of FIELD. Any string variable can be accessed with fields in this manner, including indexed string variables and other string field references. Fields must be defined by a **STRUCT** statement before they can be be used. The **STRUCT** statement is specified as follows:

```
STRUCT <item>, <item>, <item>,...
```

where each <item> can be in any of the following four forms:

<type> <name>: <length>

Defines a field with the specified <*name*> and data <*type*>, and assigns it the current index position and specified field <*length*>. If the <*type*> *is* omitted (i.e., no **STRING**, **INTEGER** or **REAL** specifier), the field assumes the type implied by its name or its prior type if already defined. If the *:<length> is* omitted, the length defaults to its previously defined length or, if not ever defined, the length implied by its data type (i.e., 80 for strings, 4 bytes for integer, 8 bytes for IEEE real, 5 bytes for 8-digit BCD real, etc.). Long lengths are useful for later references *to the* field as (pseudo) arrays (described shortly). Fields defined with *a zero* length are treated as undefined fields and therefore contribute neither their position nor their zero length to subsequent default definition parameters.

Each field <*name*> *is* an ordinary MegaBasic identifier that is not already defined as the name of a line label, procedure or function. Names of existing variables can be used, but their prior definition disappears and memory storage assigned to them is released.

Each field <*name*> *is* assigned its position on the basis of what precedes it in the STRUCT statement. If preceded by a @<*position*> expression, it takes on the position specified. If preceded by another name, it assumes the position of that name plus the length assigned to that name. The position of the leading <*name*> in the **STRUCT** list (or one immediately following a USE <*vbl*> selection) depends on its prior definition. Fields already assigned a non-zero length simply retain their prior position. Names never before defined in a **STRUCT** statement and fields defined with a zero length are given a position of 1 (and a new length).

(2) <type> <name> (<item>, <item>, <item>,...)

Defines <*name*> with a sub-field list such that the <*name*> field accesses the entire region accessed by its collective sub-fields. The <*name*> *is* assigned its type and position the same way as in form (2), but its length is set equal to the sum of the lengths of its sub-field list within the brackets. The leading sub-field is assigned the same position as <*name*>. This form lets you assign sub-structure to *higher level* names already defined, providing a means to create data structure hierarchies. All <*item*> forms except the usE-form described below may appear within the sub-field list.

Although MegaBasic lets you define related structured variable fields in a hierarchical manner, this specification merely provides a convenient way to assign positions to each name with a minimum of effort. No error occurs if you later attempt to use field names in a context that is logically inconsistent with the original hierarchy.

(3) @<position>

Defines the string index to assign to the next structure field name in the list. The at-sign (@) must appear in front of the <*position*> value so that MegaBasic can tell it apart from forms (1) and (2). Since they are equivalent to string index positions, it is an error to specify a negative or zero <*position*>. This form is used when the sequential assignment of field positions is not desired or to override other defaults imposed by the **STRUCT** statement.

The <*position*> value is relative to the subfield list it is specified within, or relative to 1 if not in such a list, for example:

```
STRUCT FIRST$:40, SECOND$[A$:20, @11, B$:30]
```

This defines B$ within a position relative to SECOND$, equivalent to SECOND$(11), or an absolute position of 51. It also implies that SECOND$ has a length of 40, because A$ and B$ overlap by 10 positions.

(4) USE <host string variable>

Defines a default string variable to access for all **STRUCT** fields that are subsequently defined and resets the current running field position back to 1 as if a new **STRUCT** statement had just begun. This default *host variable is* accessed when a field variable reference is not preceded by an explicit string variable. The USE variable remains in effect until later re-defined. The word **CLEAR** can be specified in place of the <*string variable*> to nullify the default variable selection (i.e., so that there is no default). For example:

```
STRUCT USE A$, FIELD$[A,B,C],
       USE CLEAR, LIST$[I,J,K]
```

This defines the FIELD$ set with an A$ default and the **LIST**$ set with no default. Default selections must appear outside of sub-lists. Later field definitions that follow a USE <*vbl$*> specification are treated as if they began a new **STRUCT** statement (i.e., starting at position 1 unless its is already defined you specify a different @position first).

Multi-Line STRUCT Statements

MegaBasic always continues a **STRUCT** list that ends with a comma onto the next physical line. The next line simply continues with the next <*i~em*>, i.e., it does not begin with the word **STRUCT**. In this manner, a **STRUCT** statement may continue on for any number of lines. However, there are some things you need to remember if you do use multi-line **STRUCT** statements. Make sure that you do not have any program references to any of the subsequent lines (e.g., using **GOTOS**, **GOSUBS** and **RESTORES**) because such meaningless references are not caught until they are executed. A multi-line **STRUCT** statement as a one-statement **THEN** or **ELSE** clause requires brackets [ ] around it as if it was a multi-statement clause. The **CHECK** command reports unbalanced brackets and parentheses within any program line, so if a bracketed **STRUCT** list spans beyond one line, **CHECK** incorrectly reports an error.

Since subsequent lines of multi-line **STRUCT** statements may begin with a field name followed by a colon (e.g., A$:20,...), MegaBasic will treat the such names as a line-labels unless the colon is followed by a digit or an opening parenthesis. This problem can also be avoided by defining fields with the **DEF STRUCT** described below.

Redefining Structured Variables

Names defined in previous **STRUCT** statement are redefined by the most recent **STRUCT** statement. Names of variables that already exist can be redefined as structured variables only if those variables are local to the current package (i.e., they are not **SHARED** variables). Names of line labels, procedures or functions can never be redefined for any new purpose. Once a name is defined as a structured variable field, it can only be redefined as another structured variable field.

STRUCT( ) Function

A special function useful in <*position*> expressions is **STRUCT(F)**, which returns the string index assigned to field F. **STRUCT(F,N)** returns other information about field F depending upon the value of N, as follows:

STRUCT(F,0)	Return the string index position of field F (same as STRUCT(F), with 1 argument).
STRUCT(F,1)	Return the number of bytes in the region assigned to field F.
STRUCT(F,2)	Return the data type of field F (0=string, 1=integer or 2=real).

STRUCT(F,N) returns –1 for all other values of N or if F is not defined as a structured variable field.

DEF [SHARED] STRUCT *<item>, <item>, ...*

You can define structure fields *statically* in a **DEF STRUCT** statement. As with all **DEF** statements, they must appear as the first statement on a line. Such statements consist of the word **DEF** followed by an optional **SHARED** modifier to declare all the fields as *sharable to other packages*, followed by a fully specified **STRUCT** statement. **DEF STRUCT** works just like the **STRUCT** statement except for the following limitations:

- Position and length expressions, if they appear, must be integer constants without any arithmetic, parentheses or other processing specified. A field with a zero length is treated as an undefined field: a field name with a data type only. Field lengths default to the following values when the *:length* constant is omitted: *integers=4, reals=8, strings=0.* String and real fields default to different widths under the executable **STRUCT** statement because the statement assigns a width that is determined at run time, unknown if the program is compiled.
- The leading field immediately following the **STRUCT** word always takes on a position of 1, regardless of any earlier definition of that field name. If all fields are assigned a length of zero, then they will also all be assigned a position of 1. USE <*vbl$*> specifications also reset the position counter to 1 for the next field name that follows.
- Fields that appear in more than one **DEF STRUCT** end up defined by the last **DEF STRUCT** statement they appeared in. **DEF** statements in general are processed in the order in which they appear in your program.
- USE <*vbl$*> expressions are permitted, but they only affect the default host variable assignment of fields defined in **DEF STRUCT** statement, i.e., it does not set the global default **STRUCT** USE <*vbl$*> in effect during execution.

DEF STRUCT is especially useful in programs that will be compiled with the MegaBasic compiler, which requires that **SHARED STRUCT** fields be declared in **DEF** *statements* in order to properly assign the correct data types at *compilation time.* However, **DEF STRUCT** is also useful in interpreted programs to define **STRUCT** fields before actual program execution begins, which is especially useful for field definitions that do not change during program execution.

Fields completely defined in **DEF STRUCT** statements do not generate any code in compiled programs, reducing your executable program size accordingly. Fields defined in **DEF STRUCT** statements can always be redefined by subsequent **STRUCT** statements during program execution, but remember **that the** leading field of a **STRUCT** statement takes on its prior position if already defined. **DEF SHARED STRUCT** fields do not require any actions in the *package* prologue to create them.

When developing and debugging your software and you modify a **CONTinuable** program, MegaBasic re-processes all **DEF** statements to reflect any changes you may have made in **DEF** statements, as well as to catch conflicts. However, re-processing a **DEF STRUCT** statement would cause all statically defined fields to revert to their original data types, positions and field widths. Any changes to those fields made by subsequent **STRUCT** statements during prior program execution would be lost by such a move, upsetting subsequent program **CONTinuation**. Therefore, **DEF STRUCT** statements are ignored on **DEF** statement reassertion passes whenever you modify a **CONTinuable** program. Although this prevents corrected **DEF STRUCT** statements from taking effect while debugging a **CONTinuable** program, it does preserve the current program execution state that is vital to **CONTinuing** after making program changes.

Accessing Structure Field Variables

Structured variables in MegaBasic are designed to allow ordinary string variables to represent a collection of individual data items. Although this is already possible using string indexing and conversion techniques, these methods invite programming errors and suffer from poor performance relative to the logically simple tasks that they perform.

With structure fields, you can access integers, floating point numbers and fixed-length strings within some larger string variable by name instead of using indexing expressions. Using the extremely efficient string operations of MegaBasic, collections of such variables can be moved, read, written or compared many, many times faster and with far greater simplicity than processing the same set of variables individually. You refer to structured variables in programs in the following manner:

```
Variable.field
```

where the *field is* a name to which you have assigned a string position, a length (number of bytes) and a type (i.e., integer, real or string). The *Variable* portion must be a reference to a string variable, which is then accessed through the field name specified. The string variable may be indexed, or specified as a structured field of a larger string variable, which allows for an indefinitely long *path* of fields. For example:

```
RECORD$.PERSON$.ADDR$.ZIP
```

This might refer to an integer ZIP code that resides in the ADDR$ field of a PERSON$, which in turn, is a field in a larger RECORD$ that could contain many other fields. The rules for specifying a structured variable reference are as follows:

- The leading term of a pathname must be a string variable sufficiently large to contain all bytes of the field that follows it. This string variable may be indexed or unindexed, a scalar or string array element, or a pathname that evaluates to a string variable.
- The leading term of a field pathname can be a field name only if a default *host* string variable is defined for that field. Such defaults are defined by the *USE* <*vbl*> specification, which is described in detail in the discussion on default referencing.
- The second and subsequent names in a pathname must be names of fields defined by an earlier **STRUCT** *statement*. An **error** will be reported if any of these field names are undefined or refer to a variable, function, procedure or line label.
- Only the *trailing (last) name* of a structured variable pathname may have an integer or real type. In other words, all pathname fields must be strings, except the last one which may be any type.
- Numeric fields must fit *completely* within the length of the variable specified to its left in the pathname. Unlike string indexing or string fields, which truncates a string that does not fit, MegaBasic reports an error if you attempt to access a numeric value requiring one or more bytes outside the target string or if the field itself is too short to support the numeric representation implied by the field type.
- No spaces, linefeeds or tabs may appear within a pathname. Periods are used to separate the individual fields within the pathname from one another.

- Any string pathname field may be *indexed, so* that field names that follow can refer to the indexed region of the string instead of the entire string field.
- Numeric fields (real or integer) may be followed by an optional pseudo array subscript, to refer to the nth number instead of the first number at the specified location. For example, A$.COUNT and A$.COUNT(0) both refer to the same COUNT field, but A$.COUNT(1) refers to the number in the bytes that immediately follow A$.COUNT(0). Only 1-dimensional subscripts are supported; an error is reported if you specify two or more subscripts.
- Numeric field references are supported in any vector context. The effective vector length is determined from the number of elements that fit into the string variable region accessed by the field.

Structured String Assignments

When you refer to a structured variable string, such as A$.B$, you are really referring to an indexed region of A$. For example if B$ was defined so that it accesses 10 characters at position 20, then A$.B$ and A$(20:10) are exactly identical in all respects. Either reference will be treated the same way in any context. A result of this is that string assignments to such variables behave differently than assignments to *ordinary* string variables. For example, when you assign a string to a longer indexed string region, not all the characters within the indexed region are replaced. Also, an assignment to an indexed string cannot change the length of the string variable (only its fixed-length contents).

To ensure that assignments to indexed strings or structured variable strings completely replace their contents, you have to *pad* shorter strings with spaces (or some other fill character) so that they fill out all bytes of the variable region. This is conveniently done with a special string assignment statement, as follows:

```
<string variable> == <string expression>
```

The double equal-sign indicates that the <*string variable*> region specified will be padded with extra spaces as required by shorter <*string expression*> values. This operation also holds when the <*string variable*> *is* not indexed. For example A$==B$ is really evaluated like A$(1)==B$, and hence the length of <*string variable*> *is NEVER* altered by this assignment statement. Spaces (ASCII 32 code) are normally used to pad strings, but this code can be changed by setting PARAM(7) to any ASCII code from 0 to 255. You can use this statement to assign strings to any kind of string variable, not just structured string variables. Its function is identical to the **LET** statement in Microsoft BASIC.

Passing Fields between Subroutines and Packages

You can pass names of structured variable fields to subroutines as variables (i.e., at-sign parameters), but you cannot pass an actual structured variable accessed by *variable*, because it is really an indexed string reference (even though it *appear* to be an integer or real variable). Of course, you can always pass structured variable references *by value* to subroutines, or use them in any other context that variables are allowed (e.g., assignments, expressions, **READ**s /**WRITE**s, etc.). Structured variable names can be **SHARED** between packages as long as the package defines ib own structured variables as **SHARED**.

Examples of Structured Variables

To further develop your understanding of how to define and use structured variables, several examples are described below. First, the RECORD$ variable of the earlier example could be defined by the following statements (and the assumption of a **DEF INTEGER** A-Z context):

```
STRUCT RECORD$ [PTR$:16, PERSON$:180]
STRUCT UP,DOWN,LEFT,RIGHT
STRUCT NAME$:60, ADDRESS$:60,INFO$:60
STRUCT STREET$:15, CITY$:15, STATE$:15, ZIP
STRUCT REAL SALARY, TAXCODE, etc$:40
```

Remembering that each **STRUCT** list begins with a default position of 1, the above sequence defines the following structured variable fields:

Name	Type	Position	Length
RECORD$	String	1	196
PTR$	String	1	16
PERSON$	String	17	180
UP	Integer	1	4
Down	Integer	5	4
LEFT	Integer	9	4
RIGHT	Integer	13	4
NAME$	String	1	60
ADDR$	String	61	60
INFO$	String	121	60
STREET$	String	1	15
CITY	String	16	15
STATE$	String	31	15
ZIP	Integer	46	4
SALARY	Real	1	8
TAXCODE	Integer	9	4
ETC$	String	13	40

Using these definitions, some of the meaningful pathnames that are possible are listed below:

A$.PERSON$	All PERSON$ fields in A$
B$.PERSON$.ADDRESS$	All ADDR$ fields in B$
C$.PTR$	All PTR$ fields in C$
D$.PTR$.RIGHT	The RIGHT pointer in D$
E$.PERSON$.ADDRESS$.ZIP	The ZIP code in E$
F$.PERSON$.INFO$	All INFO$ fields in F$
G$.PERSON$.ADDRESS$.STATE$	The STATE$ field in G$
H$.PERSON$.INFO$.SALARY	TheSALARY field of H$
I$.RECORD$	All RECORD$ fields of I$

Notice that the structured variable references can be applied to ANY string variable (that is long enough to contain the field being accessed). Also notice that, since the sub-field lists are all based at position 1, you have to specify full pathnames to access any field correctly. However, you can also define this structure so that any field can be referred to using only a single field name in the path. This is done as follows:

```
STRUCT RECORD$ [PTR$:16, PERSON$:180]
STRUCT PTR$ [UP,DOWN,LEFT,RIGHT]
STRUCT PERSON$ [NAME$:60, ADDR$:60, INFO$:60]
STRUCT ADDR$[STREET$: 1 5,CITY$: 1 5,STATE$: 1 5,ZIP]
STRUCT INFO$ [REAL SALARY, TAXCODE, ETC$:40]
```

or equivalently:

```
STRUCT RECORD$ [PTR$:16, PERSON$:180]
STRUCT @STRUCT(PTR$),UP,DOWN,LEFT,RIGHT
STRUCT @STRUCT(PERSON$),NAME$:60,ADDR$:60,1NFO$:60
STRUCT @!STRUCT(ADDR$),STREET$:15,CITY$:15,STATE$:15,
   ZIP, STRUCT @STRUCT(INFO$), REAL SALARY,
   TAXCODE, ETC$:40
```

Here, we have defined sub-fields so that their positions correspond to their final net position in the RECORD$. The same field references given above now reduce to:

A$.PERSON$	AllPERSON$ fields in A$
B$.ADDR$	All ADDR$ fields in B$
C$.PTR$	All PTR$ fields in C$
D$.RIGHT	The RIGHT pointer in D$
E$.ZIP	The ZIP code in E$
F$.INFO$	All INFO$ fields in F$
G$.STATE$	The STATE$ field in G$
H$. SALARY	TheSALARY field of H$
I$.RECORD$	All RECORD$ fields of I$

The access to fields using this layout is a little faster than the prior method because fewer names are used to determine the position of each field accessed. The later method is also less wordy and brief as compared with the former. However, the prior method may be more flexible in some applications because the sublevels defined can potentially be used within different data structure hierarchies.

Default Referencing

References to structured variables normally begin with the name of an actual string variable, which is then followed by a path of field names. In real life applications, this name might be the same for a large proportion of all such references. Therefore MegaBasic provides a method for declaring a string variable to be accessed by default whenever you omit the leading variable name from the structured variable reference pathname. This can greatly simplify and shorten certain complicated expressions involving such references and reduce the running time needed to evaluate them. Defining the default *host* variable involves the following statement:

```
STRUCT USE <string variable>
```

where the <*string variable*> may be a scalar string variable name, and indexed string or a string field reference. String array references are not supported. This statement declares the string variable to be used, by default, whenever you omit the variable name from a structured variable reference. You can also specify the word **CLEAR** instead of <*string variable*> to cancel the current default. An error is reported if you specify neither a scalar string variable nor the word **CLEAR**.

The default remains in effect for all subsequent statements within the current package until a different default name is declared. You cannot affect the current default structure of other packages: each package has its own default. Also, you can localize the default within procedures and functions by placing the word **STRUCT** in a **LOCAL** statement (e.g., LOCAL X,Y,STRUCT,Z$) . This lets you change the default name within a procedure or function without upsetting a possible default already declared around the call to that subroutine.

The current default *host* variable has two distinct effects upon structured variable references depending on whether or not a default was in effect when the variables were defined in a **STRUCT** statement, as follows:

- Fields defined while a default variable is in effect are assigned a *permanent* default that persists even after another **STRUCT USE** statement selects a different default.
- Fields defined with no default variable in effect are assigned tem*porary* default status. References to such structured variables always use the currently selected default as a *temporary* default; references made with no default in effect are reported as a *Structured Variable Error*. When a subsequent **STRUCT USE** statement selects another default variable, these same references follow suit and access the new variable instead.

For example, consider the following sequence:

```
10 STRUCT A,B,C;  Rem -- Defined with temporary default status
20 STRUCT USE A$;  Rem -- Select A$ as the current default
30 STRUCT X,Y,Z;  Rem -- Defined with permanent default of A$
40 STRUCT USE B$;  Rem -- Change current default to B$
```

Line 10 declares several structured fields. Line 20 declares the default name to be A$, so that now A, B and C all refer to A$ by default. Line 30 then defines three more structure fields, X, Y, and Z, and assign each the *permanent* default of A$. The in line 40 we change the current default name to B$. At this point, A, B and C now refer to B$ by default, but X, Y and Z still refer to A$ because they were defined with a *permanent* A$ context.

The default variable idea can be used in very powerful ways. For example a subroutine might refer to a set of variables which are, in fact, structured fields using the current default variable. By modifying the default variable then calling this subroutine, you can control the set of variables that it uses. This default only affects variable accesses within the package it was defined in. Each package can independently define its own structured variable default without affecting the others. Remember, however, that default variables only come into play when you omit the leading *host* variable name from the front of a structured variable pathname.

STRUCTCHANGE <*old vbl*> TO <new *vbl*>

Sets all structured variable field names that currently have the <*old variable*> default so that afterward they all use <*new variable*> as their *permanent* default. This statement affects the *permanent* default assigned to every variable in every package throughout the system that matches the old *permanent* default of <*old variable*>.

The word **CLEAR** can be specified in place of either <*old variable*> or <*new variable*> to specify the *null* default. If you specify **CLEAR** as the <*old variable*>, then all structured variable name with *temporary* status are assigned a *permanent* default of <*new variable*>. If you specify **CLEAR** as the <*new variable*>, then all structured variable names having a *permanent* default of <*cur rent variable*> are set to *temporary* default status. **STRUCT** CHANGE has to scan all variables in the system to perform its task, so don't use it unnecessarily or in tight loops where it may burn up a lot of time.

If the default variable becomes undefined at any time, subsequent *temporary* default references will generate an error. This is not detected until a reference is made. Variables become undefined if the package that *owns* them is **DISMISS**ed and no longer active. Therefore, this can only happen if the default variable is a **SHARED** string variable defined in another package.

Section 4: Pointer Variables

Pointers are supported under MegaBasic in a manner very similar to the pointer facilities provided by C, *PASCAL* and other programming languages. A pointer is a mechanism for accessing variables without using or knowing the names of those variables. In place of a name, an identifying number or address is used, called a *pointer*. The real power of this is that unlike names, *pointers* can be stored in other variables, moved around and manipulated arithmetically. As a result, the choice and access of variables is controlled by the executing program instead of being fixed within the program source code.

MegaBasic pointers can refer to user-defined functions, procedures and line labels, as well as to variables. Furthermore, a pointer can refer to an entire array or to one element within an array. To obtain the pointer associated with a named variable or other entity, simply precede its name with a caret (A), as follows:

```
PTR = ^OBJECT
```

After executing this assignment statement, the variable **PTR** contains a pointer to the variable **OBJECT**. If **OBJECT** had never been defined or assigned a previous value, a pointer value of zero would be assigned to **PTR** (an *invalid* pointer that refers to nothing). To access **OBJECT** using this pointer, you have to precede the pointer value with an asterisk (*), as follows:

```
*PTR
```

This symbol can be used to specify **OBJECT** in any context where you could specify the name **OBJECT.** For example, the following representations show how pointer references correspond to name references:

Pointer Type	*Setting a Pointer*	*Reference Meaning*
Scalar variable	**PTR = ^SCALAR**	***PTR == SCALAR**
Arrayvariable	**PTR = ^ARRAY**	***PTR(i,j) == ARRAY(i,j)**
Array element	**PTR = ^ARRAY(i,j)**	***PTR == ARRAY(i,j)**
Function Call	**PTR = ^FN_ADD**	***PTR(x,y) == FN_ADD(x,y)**

The caret (^) function extracts the pointer to a named object and always returns an integer value. Its argument can be any variable, label function, procedure or array name, or it can be an array element reference. Both string and numeric variables are supported. If you specify an array name without subscripts, then the resulting pointer must be followed by subscripts in all asterisk (*) references. A pointer derived from a subscripted array name must be used without subscripts, i.e., the *pointer reference behaves just like a scalar variable. Except for subscripted array references, the caret function (^) accepts no constants or other expressions of any kind. When extracting a pointer to a function or procedure, do not specify any of its arguments in the pointer (^) function: only specify its name by itself.

Accessing Objects Through Pointers

The asterisk (*) function converts a pointer into a reference to whatever object it is pointing to. However its argument must be an integer variable, an integer array element, or numeric expression (integer or real) enclosed in parentheses. Hence if **PTR** is a real variable, then ***PTR** is invalid and an error is reported. The integer argument of * must evaluate to a valid pointer. The pointer integer variable or expression must immediately follow the asterisk (*) without any intervening spaces, linefeeds or tabs. Invalid pointers cannot always be detected (by MegaBasic or any other language) and the unpredictable events that result from using one include wrong answers, corrupted data and crashed machines. In short, all responsibility for error detection and proper use falls on you, the programmer.

If **PTR** is a pointer to an integer variable, then ***PTR** refers to the contents of that variable, as described above. If that integer variable itself contains a pointer to another entity, then the expression ****PTR** refers to that entity. MegaBasic supports multiple levels of pointer indirection to any depth as long as every pointer involved along the way is a valid pointer. Only the last pointer in such a chain can point to a general object, while the other pointers in the chain must be pointers to integer scalar variables or integer array elements. This is because pointers must be stored in integer variables, as MegaBasic does not have a separate *pointer data type.*

Array Pointer Arithmetic

Pointer arithmetic has meaning only in the context of array element access. For example, if **PTR** = ^ARRAY(I), then the pointer reference *(**PTR**+l) refers to the same value as *ARRAY(1+1).* Successively higher array element pointers will access successively higher elements. When one dimension *runs out,* the first element of the next dimension is accessed. The elements are accessed in the order they appear in physical memory, regardless of how many dimensions the array has. Do not attempt to access array elements beyond the end of an array. MegaBasic does not check for this error, and only incorrect data (and / or crashed data structures) can result from this. Do not perform any pointer arithmetic using pointers to non-subscripted arrays, scalar variables, labels, functions or procedures: it always produces wrong and unpredictable results.

If the pointer arithmetic expression evaluates to a real number instead of an integer, MegaBasic converts it automatically to integer. However, there are two reasons why you should always avoid this. First, real arithmetic is much slower than integer arithmetic, even with an 8087 processor. Second, under 8-digit **BCD** versions of MegaBasic, real-to-integer conversion of large numbers, like pointer values, will lose some precision. This is because 8-digit **BCD** reals have less precision than 32-bit integers. Any such precision loss will destroy the pointer value for any correct purpose, and its subsequent use can even crash your computer.

The **DIM**( ) function (Chapter 9, Section 5) can provide information about the dimensions of an array. For example, if **PTR** points to ARRAY (rather than ARRAY(I)), then **DIM**(***PTR**) returns the number of dimensions in ARRAY. If, however, **PTR** points to ARRAY(I) then **DIM**(***PTR**) will return zero, because **PTR** is pointing to a scalar value, i.e., a single element within ARRAY( ).

Pointers are only valid within the scope of an executing program. As such, they can not be written to a file and then read back in a later invocation of the program. This is because pointers are related to the physical memory location of the objects they point to, which can change from one invocation to the next. A pointer to an object remains valid throughout the *life* of the object it points to. For example, a pointer to a function in another package is valid until that package is no longer in memory. You are responsible for ensuring that pointers are valid before you use them. If MegaBasic determines that a pointer is not valid, it reports a *Pointer Variable Error* (type 41).

Although a caret (^) is also used in MegaBasic as a power operator and an asterisk (*) is also used as a multiply operator, there is no program context from which the meaning of either of these symbols cannot be determined. In other words, there is never any confusion or ambiguity. This property is similar to that of the minus sign (–), which is used for both subtraction and negation.

Pointer Arguments in Subroutines

To further facilitate the pointer capabilities of MegaBasic, an additional argument type can be specified in the argument definition of procedures and functions. Consider the following procedure:

```
Def proc SHOW_VBL *PTR
Print PTR,“ points to”,*PTR
Return; Proc end
```

This subroutine displays both the pointer to a variable and its contents. Its argument can be any scalar variable or array element of any data type (i.e., string, integer or real). Notice the asterisk (*) preceding parameter **PTR** This tells MegaBasic to extract the argument pointer and pass it to the procedure, instead of the argument value. This pointer extraction is identical to the operation performed by the caret (A) function. When SHOW_VBL is called, parameter **PTR** receives the pointer, which can then be used in any manner consistent with the rules for using pointers as described earlier. For example, the procedure above displays the pointer contained in **PTR**, as well as the value it points to (i.e., ***PTR**),

Pointer parameters in function or procedure definitions (e.g., **PTR** above) must be integer scalar variables. Specifying a real or string variable in this context is reported as a *Pointer Variable Error*. The actual argument passed through a pointer parameter can be any named entity that would be permitted in caret (^) expressions (as described earlier). Therefore you also can pass function, procedure and label names to subroutines through this mechanism.

A pointer parameter is equivalent to an integer parameter that is always passed a pointer value (i.e., a caret (^) expression). The purpose of pointer parameters is to eliminate the need to specify the caret (A) in front of all references to such arguments and to hide this implementation detail from the caller. Furthermore, such parameters provide a clean and simple way for passing array element variables to subroutines and for implementing type independent parameters.

Pseudo Variables

Using the pointer facility of MegaBasic, you can manipulate variables without knowing of or dealing with their variable names. *Normally*, such variables had to be *named* somewhere, i.e., they had to exist as *ordinary* variables before you could access them through pointers. A special function lets you to create new variables whose *only access is* through pointers, as described below:

```
P = CREATE(Q)
```

where Q is a pointer to a variable *of the type desired* for the new variable and P receives the pointer to the new variable returned by **CREATE**(Q). There is no relationship between variables *P and *Q except that they have the same type and *Q is not affected in any way. The above assignment statement reads: *Create a new variable with the same type as variable *Q and store a pointer to it in* P. Variables created in this manner are called *pseudo variables*, because they do not have the usual name associated with them.

Arrays and strings created by this function have to be **DIM**ensioned before they are used, as no memory is allocated to them at creation time. If you access such variables without **DIMensioning** them, the usual default variable **DIMensions** will be created automatically as part of the first access. You can create scalar or array variables in any of the three types: integer, real or string. An error results from attempting to create structure fields, procedures, labels or functions.

Variables created by **CREATE**( ) are *owned* by the MegaBasic package that *executed* the **CREATE**( ) function. If this package is removed from the system during execution, all pseudo variables it owns will also be removed and their allocated memory released back to the system for subsequent general use. Subsequent attempts to access variables that no longer exist must be avoided, because the results will be extremely unpredictable and can crash the system.

In some applications, you may want to create and then later release pseudo variables at some point in your program without having to **DISMISS** the package that created them. To free any variable from the system, use the following statement:

```
FREE <vbl>, <vbl>, ...
```

where <*vbl*> *is* a reference to any MegaBasic variable, by name or by pointer reference. Names of arrays should be specified by name or pointer reference only (i.e., without subscripts). This statement operates slightly differently on pseudo variables versus regular named variables. Named variables will still exist after the FREE statement finishes, but without any memory allocated to them (and scalar integers and reals are left unaffected). Pseudo variables are freed completely, i.e., their allocated memory is freed and any pointers to them are no longer valid. For example, the variable created by the P = **CREATE**(Q) statement above, is destroyed by the statement **FREE *P**.

MegaBasic has a maximum capacity for 8190 symbols over all packages of a running program (which includes functions, procedures, labels and fields, as well as variables). Therefore, applications that expect to create vast numbers of pseudo variables may not succeed. Small programs can create more pseudo variables than large programs because the application begins with fewer symbols to start with. If you exceed the symbol capacity, your program terminates with a *Too Many Symbols Error*. Typically, however, even large programs with many packages use only several thousand symbols, which still leaves most of the symbol capacity available for pseudo variables. Use the *FREE(3) function* to find out how much symbol space remains.

Pointer DEF Statements

Pointer *Def Statements* let MegaBasic compiler users specify the type of object that a pointer points to in a **DEF** statement so that, in particular, local pointer functions and external pointer variables and pointer functions can be given a pointer type to support proper compilation. This statement only affects the compilation of pointers in MegaBasic programs, and has no effect when interpreted. Its complete syntax is as follows:

DEF *<name list>:<symbol>, <name list>:<symbol>, etc.*

where <*name list*> *is* a sequence of pointer variable or pointer function names separated by commas and preceded by an asterisk (e.g., *P, * Q ...), and <*symbol*> *is* the name of any already-defined symbol of the type that the pointers in the will be pointing to. Pointer variable names may be followed by parentheses ( ) to indicate pointer arrays (i.e., arrays of pointers). A data type error is reported if any of the pointer names refer to non-integers (e.g., reals, procedures, strings, etc.).

An error is reported if the symbol is not defined in any other **DEF** statements; symbols defined in later **DEF** statements can be specified, but they *must* be defined in a **DEF** statement somewhere. No other type declarative specifiers are permitted; things like **SHARED**, **REAL**, **INTEGER**, etc. must be applied to these symbols in other **DEF** statements. Pointer **DEF** statements can be placed anywhere in the program because they are processed *after* all the other **DEF** statements have been processed.

This **DEF** statement makes it unnecessary to assign a dummy pointer value to pointer variables in order for them to compile properly, making this the preferred method. Furthermore, this method is the *only correct* way to declare:

- that an ***integer function returns*** a pointer to some specific type,
- that a **SHARED** *name* of any kind is a pointer to something specific (necessary for proper compilation of programs that ACCESS those packages), or
- that a pointer argument of a subroutine (e.g., *P parameters) in *another* package is a pointer of a specific type.

This statement lets you specify the type of object that a pointer points to in a **DEF** statement so that pointer variables and pointer functions can be given correct pointer types when compiled under the MegaBasic compiler. Version 5.600 and later of the MegaBasic interpreter ignores pointer **DEF** statements, while earlier versions report them as *syntax errors. You* should *always* declare all pointers in this manner if you *ever* intend to compile your application at some later point.

Chapter 6 Program Control Statements

Normally, program execution proceeds sequentially through the statements in order by line number. Program control statements allow you to change the course of execution to suit the processing requirements. Except for the subroutine facilities, which are covered in Chapter 8, all MegaBasic program control capabilities are described in this section, as summarized below:

GOTOs and Program Termination	Statements that immediately and unconditionally change the course of program execution.
Conditional Execution	Statements that allow the results of calculation and comparison to decide subsequent program behavior.
Loops and Iteration Control	Statements to setup and control blocks of statements repetitively.
Error Trapping	Statements to control program behavior in the event of unexpected errors.

Some of the program control facilities involve specifying where to go for the next statement to execute. Line numbers are the easiest means to indicate program locations for such purposes. However MegaBasic lets you to assign names to lines, called *line-labels,* that may be placed at the beginning of a line, separated with a colon, as in the following example program line:

```
10 LABEL: C = C+1; If C<100 then Goto LABEL
```

Line-labels may be any name legal as a variable name and may end with a dollar sign if so desired. MegaBasic names follow certain rules which are laid out on Chapter 1, Section 5. Once a name is used for any purpose in a MegaBasic program you cannot use the same name for any other purpose (i.e., they must be unique). Once a line has been given a line-label it may be referred to either by line number or by line-label. In other words, line-labels and line numbers are interchangeable when referring to a line. If a line-label is not followed by any statements on the same line (i.e., the line consists solely of a label and a colon), references to that label will actually refer to the next line. Line-labels on lines by themselves can be useful for making the label stand out to improve readability.

The pseudo-line-label **NEXT** may also appear anywhere that line numbers and line-labels are expected. This special reserved word, when used like a line number, refers to the location in the program of the nearest closing **NEXT** statement. This feature is discussed more fully in the context *of* the **NEXT** statement, described later in this section.

It is of course an error to refer to a line number or a line-label which is not present in the program as specified. MegaBasic does not find references to things that do not exist in your program until they are encountered during actual program execution. However,

the **CHECK** command (Chapter 2, Section 4), will find many of these *dangling* references along with others kinds of errors that may also be present.

Section 1: GOTOs and Program Termination

The simplest and most direct of all program control statements are described here, which merely cause execution to either begin somewhere else in the program or terminate execution in a variety of ways as summarized below:

GOTO	Branches to a fixed location in your program.
ON..GOTO	Branches to one out of a list of program locations, based upon an index value.
STOP	Pauses program execution for testing purposes, allowing for later CONTinuation.
END	Terminates the program and passes an exit code back to the process that invoked the program.
DOS	Terminates the program and exits back to the operating system. Also lets you execute *shell commands* from your program (without stopping your program).

GOTO <label>

Causes program execution to continue at the line number or line-label specified. This is sometimes referred to as an unconditional branch. A *Line Number Error* results if the line does not exist as specified. The line label referred to must be in the same program as the **GOTO** statement. When multiple programs or packages are in memory, the only way to transfer control between them is by **PROCedure** or **FUNCtion** calls: **GOTOS** and other line references are not allowed.

The **GOTO** keyword is optional when it is the object of a **THEN** or **ELSE** clause in an **IF** statement (discussed later in this section). Do not use a **GOTO** to permanently exit a subroutine of any kind (function, **GOSUB** or procedure). This is because MegaBasic supports recursive programming and therefore assumes a subroutine is active until a **RETURN** statement (Chapter 8, Section 1) is executed. However, you can jump out of a **GOSUB** with a **GOTO** as long as a **RETURN** statement is eventually encountered, such as jumping to another **GOSUB**.

If a **GOTO** is used to branch out of a **FOR**, **WHILE** or **REPEAT** loop, the loop will be terminated and execution will continue normally. Any number of nested loops can be terminated by one such **GOTO** and MegaBasic always continues properly at the nesting level in effect in the line specified. **GOTOS** of any type all operate in this manner. See the **FOR**, **WHILE**, **REPEAT** and **NEXT** statements for more details. If the line label specified is the keyword **NEXT**, MegaBasic branches to the beginning of the next *current-level* **FOR**, **WHILE** or **REPEAT** loop iteration.

A **GOTO** inside a multi-statement **THEN** or **ELSE** clause terminates the **IF** statement operation, i.e., it is assumed to *always* jump out of the clause. If its target is inside the clause, an error will occur as soon as the closing bracket a) is encountered.

ON <*expr n*> GOTO<*line list*>

Evaluates the numeric expression and converts the result to an integer which selects one position in the <*line list*>. This integer must be from 1 to the length of the <*line list*>. Program control is then transferred to the line selected. The <*line list*> consists of a sequence of line numbers and / or line-labels, separated from one another by commas, which must refer to actual lines that already exist in the program. This is commonly referred to as a computed **GOTO** or a multi-way branch.

A typical **ON**..**GOTO** application is selection of some routine based on a user-entered selection code. The following example gets a character from the user and jumps to the desired routine only if it is a valid selection:

```
10 C$ = Inchr$(0); Print C$,
20 On Match("ABCXYZ",C$)+1 Goto
BADSEL,40,50,60,70,80,90
30 BADSEL: Print " Bad selection, re-enter-- ",; Goto 10
40 ....
50 ...
```

Line 10 gets and displays the user-typed selection code. Line 20 uses the MATCH function to obtain a code from 0 to 7 corresponding to one of the selections or none (0 returned if C$ not in *ABCXYZ)*. The **ON**..**GOTO** uses this code and jumps into the selected routine. Notice that the 0 case has been programmed to reject the users' selection and repeat the processes until a valid response is typed.

END *[<exit code>]*

Immediately terminates execution of the program. Finishing the last physical statement in the program or encountering an untrapped program error has the same effect as an **END** statement. When an **END** statement is encountered, the following steps are performed:

- The optional EPILOGUE routines (Chapter 10, Section 2) of each active MegaBasic package are executed in the same order the packages were loaded from the disk. This mechanism gives each package an opportunity to terminate *gracefully.* However, you can better control the order of EPILOGUE execution using explicit DISMISS statements before you terminate.
- All unwritten file buffers are flushed and all open files are closed.
- The program terminates and returns the optional <*exit code*> back to the *context level* where the program began execution (see below).

After the program terminates, MegaBasic returns back to the command level that originally invoked the program. For example, if you run the program from the operating system command (or *shell) level* that is where you end up after an **END** statement. If you run the program by typing **RUN** in the MegaBasic command level, after an **END** statement you end up at the MegaBasic command level *Ready* prompt. If a process, such as an **MS-DOS** batch file, invoked the program, the process resumes after the **END** statement is executed.

Upon returning to the operating system, MegaBasic provides an exit code that a batch or other process can access through the D0S exit code service. This code is set by MegaBasic depending on what caused the exit back to the operating system:

- Zero is returned when exiting MegaBasic from the command level i.e., using the **BYE** or **DOS** commands, or after a MegaBasic program terminates normally and exits to the system level.

- The MegaBasic error code is returned when a program terminates abnormally, i.e., after an untrapped error (including an untrapped Ctrl-C abort).
- The exit code specified by the **END** <*exit code*> statement is returned the the operating system level if the program was run from there. This code is an optional argument on the **END** statement that can be set to any positive integer from 0 to 255.

This code is useful to the process that invoked the program so that it can base its next action upon the success or failure of the MegaBasic program. Exit codes are only supported by the more recent operating systems: **MS-DOS**, Concurrent CP / M and Xenix.

STOP *[<data output list>]*

STOP suspends program execution and puts you back into the command level of MegaBasic (or all the way back to the operating system from the **RUN** version) and displays a message like:

```
Stop in Line 315
```

which indicates where the **STOP** took place. **STOP** is usually employed for debugging purposes since the program may be continued later on with the **CONT** command (Chapter 2, Section 4). **STOP** statements are therefore quite useful as breakpoints, especially during a debugging session since you can insert and remove program lines of a continuable program and resume execution. **STOP** does not close any files, but it does flush any data which has been written by your program but not yet posted to the file.

A **STOP** statement may optionally contain a data list which is displayed instead of the **STOP** message shown above. This form of **STOP** is exactly like a **PRINT** statement in all respects except that after its data is displayed, program execution terminates in the manner described above.

DOS *[<command line>]*

DOS by itself (i.e., without arguments) is just like **END** except that upon completion, MegaBasic itself is exited and control passes back to the operating system level. Like **END**, **DOS** updates any unwritten file buffers to their respective files prior to actual termination and then all open files are closed.

Under some operating systems (e.g., **MS-DOS** and Xenix) you can specify an operating system *shell* command as a string expression argument to the **DOS** (Chapter 7, Section 3) statement (e.g., **DOS "DIR A:"** or **DOS "TYPE FILE"**). Instead of exiting MegaBasic, the specified shell command is passed to the system and executed, after which your program resumes execution. If the shell command returns an *exit code, you* can pick it up from PARAM(19) right after the **DOS** statement returns.

Section 2: Condition Execution

All decision-making during program execution is made using either **IF** *statements* or a more general form called **CASE** *statements.* These facilities let your program control which portions of a program are executed or ignored, based upon an arbitrary condition or set of conditions.

IF..THEN..ELSE	Executes one of two sets of statements, based on some *true or false* criteria.
CASE BEGIN..END	Executes one statement sequence, selected from any number of cases based on an arbitrary criteria.

Together with the looping mechanisms of MegaBasic (Chapter 6, Section 3), these statements provide everything you need to control program execution without resorting to **GOTO** statements (which tend to obscure program structure and should be avoided whenever possible).

IF *<logical exprn>* THEN *<statement1>* ELSE *<statement2>*

Evaluates the <*logical exprn*> and, on the basis of its outcome, selects and executes one of two different statements. If the <*logical exprn*> *is* **TRUE** (i.e., non-zero), MegaBasic executes <*statement1*> (known as the **THEN**-*clause).* If the <*logical exprn*> *is* **FALSE** (i.e., equal to zero), MegaBasic executes <*statement2*> (known as the **ELSE**-*clause).* After executing either <*statement1*> or <*statement2*>, the **IF** statement is finished and MegaBasic goes on to execute whatever follows it.

You can *omit* the **ELSE** clause from any **IF** statement (i.e., the word **ELSE** along with <*statement2*>*).* If you do, then the **IF** statement simply controls whether or not <*statement1*> *is* executed. The examples below show some simple **IF** statements and how they work:

```
If X<100 Then X = X*C; Rem -- no ELSE clause here
If X=0 Then Print "X is Zero"
   Else Print "X is non-Zero"
```

Although the <*logical exprn*> *is* usually a simple comparison of some type (e.g., IF *X=Y* **THEN**...*)*, an expression of any complexity is permitted as long as it produces a numeric result and the entire <*logical exprn*> fits on the same line. Any combination of string comparisons, numeric comparisons and general numeric expressions are permitted, for example:

```
If A$>B$ or Not X+Y and Z<50 Then Link P$
```

Any single statement may be used as <*statement1*> or <*statement2*>. As a special case, when a **GOTO** follows a **THEN** or **ELSE,** the **GOTO** reserved word may be omitted leaving only the line number or line-label specified, as in:

```
If A$>B$ Then 2000 Else LINLBL
instead of:
If A$>B$ Then Goto 2000 Else Goto LINLBL
```

Since **IF**..**THEN**..**ELSE** is also a single statement, using it in another **IF** statement creates a compound or nested **IF** statement. The optional **ELSE** clause in such a statement is always associated with the nearest previous **IF**. For example:

```
If X Then If Y Then Z=0 Else Y=0
Else If Z Then Y=0 Else X=0
```

The various kinds of things your can specify in **THEN** and **ELSE** clauses are summarized below:

Any single statement	Any single statement that can be executed by itself can be placed after a **THEN** or **ELSE**, including another **IF** statement
Compound statement	Two or more statements separated by semicolons and surrounded by squarebrackets [...]. These are discussed in detail on the next page.
Line number or label	The **GOTO** reserved word is optional on **GOTO** statements that immediately follow **THEN** or **ELSE**. This is called an *implied* **GOTO** statement.
NEXT	Same as **GOTO NEXT** (see the **NEXT** statement in Chapter 6, Section 3).
FOR, WHILE or REPEAT	Any well-defined *multi-statement loop* can form a complete **THEN** or **ELSE** clause without requiring brackets [ ] around the loop, i.e., the loop is treated as a single statement.

An important distinction between **IF** statements in MegaBasic and other **BASICS** is that additional statements that follow an **IF** statement on the same line are in no way connected with that **IF** statement. Some other BASICS Wi11 skip all such statements in the event of a false **IF** condition. Under MegaBasic, you can place following statements on the same line or on successive lines with the same effect.

Compound Statements

For greater expressive power, either <*statement1*> or <*statement2*> or both may be a compound statement, which is several ordinary statements grouped together and executed as a unit. To form a compound statement from several individual statements, surround them with brackets [ ]. Compound statements only appear within **IF** statements (after a **THEN** or **ELSE**) and may extend indefinitely over one or more program lines. The following example should clarify their use:

```
If X=Y Then [R=Z; Swap S,T]
            Else [For l=1 to 10; R=R+X(I); Next]
```

Notice that **FOR**..**NEXT** (and **WHILE**..**NEXT**) loops may be included within compound statements. When an **IF** statement is employed within a compound statement, it too can include compound statements for its **THEN** or **ELSE** clauses. You can use the bracketing mechanism to override the normal precedence of **ELSE** clause processing whenever required, for example:

```
If X=Y Then If A$=B$ Then S=T Else T=S
```

In this example, the **ELSE** refers to (by default) the second **IF** which is only executed if X-Y. Suppose that the desired action is to execute the **ELSE** clause upon failure of the first **IF** test (X=Y). This can clearly be done as follows:

```
If X=Y Then [If A$=B$ Then S=T] Else T=S
```

Null ELSE-clauses in IF Statements

In statements such as: *IF cond1 THEN IF cond2 THEN S1 ELSE S2,* an **ELSE** clause refers to the most recent (or second) **IF** statement. Occasionally this may not be appropriate, when you wish the **ELSE** to refer to an earlier instead. Therefore the **IF** statement supports a *null* **ELSE** clause to permit this kind of specification, for example:

```
IF Cond1 THEN IF Cond2 THEN Stat1 ELSE ELSE Stat2
```

Notice the *double* **ELSES**. The first **ELSE** refers to the second **IF**, just as before, but it has no statement associated with it. Such an **ELSE** is called a *null*-**ELSE** because it acts as a *do-nothing*. The second **ELSE** refers to the first **IF** statement, which is what was desired. Any number of *null* **ELSES** can be strung together to *pad* a multi-level **IF** statement so that the last will **ELSE** match the desired **IF** statement level.

Multi-line IF Statements

Although, for the most part, **IF** statements tend to be short enough to fit completely within the same program line, **IF** statements may span any number of program lines. This lets you construct multiple level **IF** statements of any complexity the way you can in PASCAL or C. A multi-line **IF** statement in MegaBasic works just like the simpler single-line **IF** statement, except that the **THEN** and **ELSE** clauses can extend beyond the one-line limit and physical line breaks can appear almost anywhere within the ~ statement. Specifically, the following rules and limitations must be observed:

- The **IF** and <*logical exprn*> components must fit in and appear on the same program line (which can be up to 255 characters long). Condition expressions longer than this are unusual, but they can often be broken up into smaller pieces that fit the form:

  ```
  If <cond1> then If <cond2> then If <cond3>...
  ```

- Physical line breaks may occur anywhere except within a simple statement or between the **IF** and the <*condition*> expression. For example, a line break may occur on either side of the **THEN** or **ELSE** keywords or on either side of a compound statement bracket.

- Any branch to an explicit line number or label from within an statement will logically exit the **IF** statement. Hence you cannot use **GOTOS** within **IF** statements except to completely jump out of them. However, loop-relative branches can be used without exiting the **IF** (i.e., **EXIT**, **GOTO NEXT**, etc.) when the loop resides entirely within a **THEN** or **ELSE** compound statement.

- **GOTOS** that branch into the middle of an **IF** statement will generally lead to an error condition. This is because an **IF** must be entered from the *top* in order to properly process **THEN** and **ELSE** clauses and deal with compound statement brackets properly. MegaBasic does not detect such errors until it encounters an unexpected **THEN**, **ELSE**, or bracket [ ].

- Only use a **GOTO** within a clause to jump out, never to jump within the clause. MegaBasic treats all such jumps as **IF** clause exits and an error will occur if they don't actually exit.

- Functions, procedures and **GOSUBS** can be invoked from within **IF** statements without any effect upon the state of the **IF**, even if **GOTOS** or other (unrelated) **IF** statements are executed within them. Recursive re-entry into an active **IF** statement is also supported.

- There is no practical limitation on the length of an **IF** statement or on how many **IF**-levels or nested compound statement levels you can have. An **IF** statement can go on for pages as needed. There is no speed penalty associated with using multi-line **IF** statements as compared with using single-line **IF** statements.

CASE BEGIN ... CASE END

A **CASE** statement is a program control structure that lets you perform complex multi-way decisions without using **GOTOS** or **IF** statement. Like a **FOR**, **WHILE** or **REPEAT** statement, a **CASE** statement initiates a process that controls how a subsequent block of statements is executed. A complete **CASE** block begins with a **CASE BEGIN** statement, in which you can specify a **CASE** argument value (string or numeric), and ends with a **CASE END** statement. Between the **CASE BEGIN** and **CASE END** statements, you specify a series of **CASE** branches, each consisting of a **CASE** test statement followed by any number of statements to be executed in the event that the **CASE** test is successful.

When a **CASE** block is encountered during program execution, the following things happen:

- MegaBasic verifies that the **CASE BEGIN** statement is matched by a later **CASE END** statement (an error results if none is found).
- Second, the **CASE** argument is evaluated and saved for later use in the **CASE** tests. For certain reasons described below, this argument is optional.
- MegaBasic scans each of the **CASES** within the **CASE** block to compare the **CASE** argument with the **CASE** test expressions. If the **CASE** argument matches one of the **CASE** tests, the sequence of statements associated with that **CASE** is then executed, after which execution continues on the first statement after the closing **CASE END** statement.

If none of the test values match the **CASE** argument, none of the **CASES** is executed, the **CASE** block is exited and control passed to the first statement after the closing **CASE END** statement. To help understand how all this works, consider the following program fragment:

```
200 Rem -- Branch to the Case that matches X
210 Case begin on X  ;Rem define beginning of CASE block
220 Case 1  ;Print "X equals 1
230 Case 2,5   ;Print "X equals 2 or 5
240 Case Z+3   ;Print "X equals Z+3
250 Case 8 to 25  ;Print "X lies in the range from 8 to 25
260 Case > 5, < 0 ;Print X is below 0 or above 5
270 Case ;Print X is none of the above
280 Case end   ;Rem define end of the BASE block
```

Most elements of the **CASE** statement are represented in this example. Line 210 defines the beginning of the **CASE** block and evaluates the **CASE** argument. If X=1 then the **CASE** test on line 220 is satisfied, and MegaBasic proceeds to execute all of the statements following the **CASE** test up to the next **CASE** statement. This sequence of statements is not limited to one line, and may potentially span many lines or even pages before being terminated by a subsequent **CASE** test or the final **CASE END** statement. A **CASE** branch may contain no statements at all. Such a *null* branch has the effect of exiting the **CASE** block when its corresponding test succeeds.

Notice that the second **CASE** test (on line 230) contains two values. **CASE** tests consist of one or more values, separated from one another with commas. Each test value is applied in turn until one of them succeeds or all of them fail. Furthermore, the **CASE** test values may be specified as expressions, instead of being limited to simply constant values (as in **CASE** statements of other languages). This is illustrated on line 240 where the **CASE** argument (X) is compared with the value Z+3.

Any **CASE** test expression can be preceded by an optional comparison operator, as shown on line 260, that specifies what type of comparison to use. By default, MegaBasic applies an equality comparison when you do not specify the comparison operator (i.e., specifying an equals sign (=) means the same thing as omitting the operator). In the example above, the **CASE** branch is taken if X is greater than 5 or less than 0 (notice that there are two test expressions). Any of the MegaBasic comparison operators are permitted (i.e., = <><=>=<>). A **CASE** test consisting of two numbers separated by the word **TO** provides the means to specify a numeric range, as illustrated on line 250 of the prior example.

You can also specify a **CASE** test without any values, which creates a test that always succeeds, as shown on line 260. Such a **CASE** test is used to execute a group of statements in the event that none of the other tests have been satisfied. Naturally, if you use this capability, this *empty* test must be the last one in the block (before the **CASE END**) because any others that follow could never be reached.

The *on value* of the **CASE BEGIN** statement is optional. By omitting it, all **CASE** test values are treated as *conditional expressions*, like the one used in **IF** statements where, instead of comparing the **CASE** test values with a **CASE** argument, MegaBasic evaluates each test expression as a *true or false* (i.e., non-zero or zero) and the first one that evaluates to *true is* the **CASE** branch taken. The example below shows how this is done:

```
200 Rem -- Branch to the first Case that is true
210 Case begin;   Rem - this is a logical CASE
220   Case Not EOF(D);  Print “First case executed.”
230   Case UP or DOWN;  Print “Second case executed.”
240   Case A+B=2;    Print “Third case executed.”
250   Case X>Y or Z=2;  Print “Fourth case executed.”
260   Case;    Print “Catch-all case executed.”
270 Case end;  Rem - define of the CASE block
```

This feature lets you create your own test conditions when the simple equality method isn’t adequate for your application. For example, one **CASE** test might test X<Y, and the next **CASE** might test V>LOW AND V<HIGH, and so on.

When omitting the **CASE** argument, specify **CASE BEGIN** by itself without any **ON** <*expr n*> clause. An *Out Of Context Error* results if you specify a comparison operator in front of any conditional **CASE** test expressions, as this does not have any meaning. A complete summary of **CASE** statement usage and operation follows.

CASE Block Definition

A **CASE** block is defined as a sequence of statements that begin with a **CASE BEGIN** statement, end with a **CASE END** statement, and in between contain one or more **CASE** branches. All program statements between the **CASE BEGIN** and its first **CASE** branch are skipped and can never be reached by normal program execution. Avoid placing any statements in this region except for **REMarks.**

The **CHECK** command will report any **CASE** blocks that do not contain matching **CASE BEGIN** and **CASE END** statements. The display produced by the **TRACE RET** command will show an entry for each active **CASE** block. Both of these features are useful during the development, testing and debugging of applications that use **CASE** statements, especially nested **CASE** blocks.

CASE Selection Criterion

The **CASE BEGIN** statement may include an optional **ON** expression argument, that defines the target value to locate among the **CASE** branches. *The expres*sion may evaluate to a string or a number. If the argument is omitted, the **CASE** test expressions are all treated as conditional expressions. For performance reasons, a string **CASE** argument is limited to a maximum length of 255 characters, and a Length *Error* results if this limit is ever exceeded.

CASE Branches and Nested CASEs

CASE branches each consist of a **CASE** test statement followed by zero or more program statements that are executed if the **CASE** test succeeds. There is no limit on the length of any one **CASE** branch, as it includes all statements from the **CASE** test statement up to the next **CASE** test or **CASE END** statement. If no statements follow a **CASE** test (before the next **CASE** test of **CASE END**), then it specifies a null **CASE** branch that does nothing when the test succeeds and the **CASE** block simply exits without doing anything. Other **CASE** blocks may be nested within any **CASE** branch to any desired depth.

CASE Test Statements

CASE test statements consist of the reserved word **CASE** followed by zero or more test case expressions separated by commas. A test case expression can be one of the following:

- A string or numeric expression, providing a single value to match exactly.
- A string or numeric expression preceded by a comparison operator (i.e., one of: = <> < <= > >=). String expressions can also be preceded by the **IN** operator, for set membership tests.
- Two string or numeric expressions separated with the reserved word TO. This specifies a range of values that the case test will accept. The first expression should be less than or equal to the second expression to provide a range of values to check against (i.e., the test fails if you specify a higher first value). The range specified includes both end points of the range.
- No expressions of any kind. This specifies a *null test* that always succeeds (i.e., this branch is always taken if encountered).

Your case test may include one or more of the above methods in any combination after the **CASE** reserved word. See the examples provided earlier. An empty or *null* test is useful to implement an *if-all-else-fails* branch in the last **CASE** branch (sometimes known as **CASE ELSE** in other languages). Subsequent **CASES** following such a branch are unreachable by normal execution.

The data type of the **CASE** test expressions must match the data type of the **CASE BEGIN** argument expression. For example, if the **CASE BEGIN** argument is a string, then all of the **CASE** test expressions must also evaluate to strings (otherwise a data type error is reported). If the **CASE BEGIN** argument is numeric, the **CASE** tests must also be

numeric and, if necessary, will be converted to the same numeric type (integer or real) as the **CASE BEGIN** argument before each comparison is made. To avoid *unnecessary* type conversions, arrange for numeric arguments and test values to have the same numeric type. Logical test expressions may evaluate either to integer or real without incurring any conversion penalty.

Logical *CASE* Test Statements

If the **CASE BEGIN** contained no argument value (i.e., no **ON** *expression)*, the test expressions are evaluated as logical expressions. Test success is indicated by the first expression that evaluates to *true*, i.e., a non-zero value. In this context, it is meaningless and an error to specify a comparison operator in front of any logical **CASE** test expression.

If the test succeeds, the statements in the **CASE** branch are executed until the next **CASE** test or **CASE END** statement is encountered. At that point, execution continues with the first statement after the closing **CASE END** statement. If the test fails, the branch is skipped and the next **CASE** test is examined. If no tests succeed, execution drops out of the **CASE** block without executing any of the branches and continues with the first statement after the closing **CASE END** statement.

Exiting a CASE Block

Occasionally you may need to exit the **CASE** block from within the **CASE** branch before the branch has finished executing. There are several ways to do this depending on where you want to go after exiting:

- **CASE EXIT** will exit the current **CASE** branch and resume execution on the statement following the closing **CASE END. CASE EXIT EXIT** will exit two **CASE** block levels and, likewise, you can exit any number of **CASE** blocks by specifying **EXIT** that number of times. **CASE EXIT** operates correctly even if it is buried within one or more levels of loops (i.e., **FOR**, **WHILE** or **REPEAT**).
- **RETURN** statements exit all **CASES** and loops in the current subroutine call and returns to the caller.
- Loop **EXIT** statements executed from within a **CASE** branch will exit the nearest **FOR**, **WHILE** or **REPEAT** loop, even if that loop surrounds the current **CASE** block or several nested **CASE** block levels.
- Use a **GOTO** to branch directly to the closing **CASE END** statement. If you wish to branch out of several nested levels of **CASE** blocks, branch to the **CASE END** statement that closes the highest level you wish to exit from.
- A **GOTO NEXT** branch will start the next iteration of the nearest outer loop, even if one or more nested **CASE** block levels have to be exited.
- A **GOTO** to a **CASE BEGIN** statement or to one at an even higher level will re-start that **CASE** block. If you do this, be sure that the conditions affecting **CASE** selection are somehow modified by the **CASE** branch to avoid an infinite loop.

You should avoid using **GOTOS** in **CASE** blocks because they are not necessary and can lead to confusing code and get you in trouble. Branching (i.e., **GOTO**) to any statement within an *inactive* **CASE** *block will* produce unpredictable results, and eventually leads to an *Unexpected Case Error* when the very next **CASE** statement is encountered. Branching to any line *after* the **CASE END** other than those described above is not defined and may also produce unpredictable results.

CASE Performance Hints

The switching speed of a **CASE** block depends solely on the number and complexity of **CASE** tests that need to be evaluated before a successful match is found. The length of the **CASE** branches has no effect whatsoever on how fast MegaBasic can sequence from **CASE** test to **CASE** test (even if some of the **CASE** branches span many lines). Also, the inclusion of comparison operators in front of any or all test expressions has no effect on how fast the tests are evaluated.

Since **CASE** tests are evaluated sequentially until a match is found, you can significantly speed up some **CASE** statements by carefully ordering the various **CASES** so that the most likely **CASE** appears first, followed by the next most likely, and so on with the least likely test appearing last.

When specifying a series of logical test expressions in one **CASE** test statement, instead of separating them with commas, separate them with *OR* operators. This will combine the separate expressions into a single long expression that evaluates somewhat faster than the series of smaller ones. Note that this is only a performance recommendation, not a requirement.

Section 3: Program Loops and Iteration Control

Loops are sequences of statements that are executed over and over again and usually represent the areas in your program where most of the time during execution is spent. The following *loop constructs* are provided in MegaBasic to support different ways of sequencing and terminating loops:

FOR..NEXT	Loops while automatically sequencing through *one or more arithmetic series.*
WHILE..NEXT	Loops as long a conditional expression *tested at the top* of the loop remains true.
REPEAT..NEXT IF	Loops as long a conditional expression *tested at the bottom* of the loop remains true.
EXIT	Terminates any loop type and branches to either the first statement after the loop or to some other specific location in the program (similar to a GOTO).

FOR *<index vbl> = <range1>, ..., <rangeN>*

A **FOR** statement defines the beginning of a repetitive statement block and specifies an *index variable* and a series of values that it takes on, one at a time for each iteration through the block of statements. For example, the following **FOR** *loops* prints the integers from 1 to 100:

```
For I = 1 to 100; Print l; Next I
```

Notice the last statement, **NEXT** as it defines the end of the **FOR** loop. YOU can place any number of program statements, including other **FOR** loops, between the **FOR** statement and its *closing* **NEXT**. The idea is to execute a group of statements (located between a **FOR** and a **NEXT** statement) repeatedly while setting the index variable to successive values of the numerical series specified. A **FOR** loop will terminate when the index variable *exceeds* the series limit. When this happens, program execution resumes at the statement immediately following the **NEXT** statement (on the same line or later).

You can also specify *more than one* series of values in a **FOR** statement, processed from left to right. When the first series is exhausted, the next one to the right of it is used until the index variable has been set to each value specified by every series. For example, the **FOR** loop below prints all integers from 1 to 10, followed by the integers from 20 to 100 by tens:

```
For I = 1 to 10, 20 to 100 by 10; Print l; Next I
```

This **FOR** statement specifies two series, separated by a comma. The second series additionally includes a different increment to use between values, called the *step size.* Actually, you can specify each series independently in any one of three ways:

- *<first value>* **TO** *<last value>* ***BY*** *<step size>*

  Specifies that the index variable will start at the *<first value>* and is incremented by the *<step size>* after each iteration until the *<last value>* has been exceeded (terminating the loop). To specify a descending series, the *<first value>* must be higher than the *<last value>* and the *<step size>* must be a negative value.

- <*first value*> TO <*last value*>

  Same as form (a) except that the omitted <*step size*> defaults to a <*step size*> of 1. The is the most commonly used **FOR** series format.

- <*single value*>

  Specifies a series consisting of one value, resulting in exactly one iteration. This *series* terminates without incrementing the index variable after executing exactly one iteration.

All series parameters are specified with general numeric expressions and may evaluate to non-integer values. These three forms of series may be mixed in a single **FOR** statement as needed. Most of the time you will be using a single *series* (the second form) but look at the following examples for a feeling of the other possibilities:

```
For X=1 to 100
For X=175 to 38 BY –1
For X=SQRT(Y) to Z–10 by S
For X=–12, Y7, 31, A(F,G)
For X=1 to 10, 20 to 100 by 10, 200 to 1000 by 100
For X=1, 2, 4, 8,10 to 20 by 2, –58 to –1000 by –7.53
For X=FUNCTION1 to FUNCTION2 by R+FUNCTION3
```

Each iteration begins by comparing the current index variable value with the <*last value*>. Execution proceeds through the loop body only while the index variable value remains within its defined series. As each series is completed, the index variable is set to the first value of the next series and the loop continues. The loop terminates at the end of the last series listed in the **FOR** statement. Zero iterations are performed for any series whose first value is already beyond the terminating value specified for that series. Since the series are evaluated only once and maintained internally, none of the parameters for the current series can be modified while the loop is progressing through it.

The index variable must be either an integer variable or a real variable. A *Data Type Error occurs if* a string variable or a numeric array element is specified as an index variable. You must use a real variable for the index *if* the values it will take on during the loop contain decimals (i.e., they are non-integer values), or *if* the index variable is used within the loop in predominantly real computations. Such computations will force MegaBasic to convert the integer index variable to real each time it is used in a real context, which can unnecessarily slow down execution.

In most applications, the **FOR** loop index variable will take on only integer values and will be used for things like array subscripts and string index calculations. You should use an integer index variable instead of a real variable when such is the case, because the loop itself will execute more than 3 times faster. Integer loops in MegaBasic have been specially optimized, and computations involving integers run much faster than the same computations done using real representation. Array subscripts and string index expressions are always processed internally with integers, so when you provide integers from the start, MegaBasic has less work to do than if you specified such values using floating point numbers (which have to be converted to integer representation anyway before they can be used).

Loops must terminate eventually, or your program would never come to an end. **FOR** loops will terminate when all the values specified for the index variable have been exhausted. You can however terminate **FOR** loops in other ways, all of which are summarized as follows:

- Falling out the bottom of the loop after the index variable has taken on the last value of the last series. In other words, letting the loop run its normal course.
- Branching out with a **GOTO** or implied **GOTO** (a **THEN** or **ELSE** *<label>)*. You can jump from an inner loop to any outer loop with any number of levels in between, as long as you do not jump beyond the current subroutine level.
- Branching out using the **EXIT** statement (Chapter 6, Section 3).
- Executing a **RETURN** statement (Chapter 8, Section 1), which exits not only the loop but its matching **GOSUB**, user-defined function or procedure as well.
- Causing an error which is trapped by an **ERRSET** statement (Chapter 6, Section 4) at some higher level.

For efficiency (as well as for recursive reentrance), MegaBasic maintains a **FOR**..**NEXT** internal structure in the scratchpad area for the lifetime of the loop. If you terminate a loop using any of the above methods, MegaBasic automatically recovers the loop structure appropriate to the context into which you jumped. Two types of jumps are not meaningful however

- Branching into the middle of a lower, inner loop is illegal because the inner loop has not been initialized.
- Branching into a higher, outer loop active in a subroutine at a higher level is illegal because MegaBasic cannot leave a subroutine without executing a **RETURN** statement.

At times, you may wish to skip the remainder of the current loop iteration and begin again with the next iteration. For example you might be looping through all the elements of an array and performing some computation only if certain criteria are met. To do this, you need only branch to the **NEXT** statement that terminates the loop. A **GOTO** can do this if the **NEXT** is the first statement on its line. However, MegaBasic will automatically begin the next iteration of any loop if you say **GOTO NEXT** (also ...**THEN NEXT** ...**ELSE NEXT** or any other form of **GOTO**). This is further explained by the discussion on the **NEXT** statement in Chapter 6, Section 3.

WHILE *<logical expression>*

Similar to a **FOR** statement, **WHILE** statements provide a looping structure that repeatedly executes a group of statements (terminated by a **NEXT** statement) until some condition is no longer true. The condition in this case is a logical expression that is evaluated at the start of each iteration of the loop. In order for a **WHILE** loop to terminate, the body of the loop must at some point cause the logical expression to evaluate to zero *(false)*. No iterations through the loop will be made if the logical expression evaluates to zero at the top of the first iteration. For example:

While X<100: X=X+1; NEXT	Increments X until it is greater or equal to 100
While Z=Y; X=X+1; NEXT	Increments X forever if Z was equal to Y.

The first example does nothing if X is already 100 or greater. The second example illustrates what happens when the logical expression is not altered in the body of the loop. It may be that you desire such an infinite loop in your application because you employ other means to terminate the loop, for example:

```
While Y=Y; X=X+1; IF X>=100THEN EXIT; NEXT
```

This example illustrates the use of the EXIT statement (described below) in terminating a **WHILE** loop, bypassing normal termination. The methods for legally terminating **WHILE** loops are summarized below:

- Causing the condition to evaluate to zero (false).
- Branching out with a **GOTO** or implied **GOTO** (a **THEN** or **ELSE** <*label*>). You can jump from an inner loop to any outer loop with any number of levels in between, as long as you do not jump beyond the current subroutine level.
- Branching out using the **EXIT** statement (Chapter 6, Section 3).
- Executing a **RETURN** statement (Chapter 8, Section 1), which exits not only the loop but its matching **GOSUB**, user-defined function or procedure as well.
- Causing an error which is trapped by an **ERRSET** statement (Chapter 6, Section 4) at some higher level.

REPEAT... NEXT *[* IF *<condition>]*

REPEAT has no arguments; its presence denotes the beginning of a **REPEAT** loop. This type of loop always executes at least one iteration because it tests its loop control condition at the end of the loop. In other words, a **REPEAT** loop is just like a **WHILE** loop with its optional test at the end instead of at the beginning, for example:

```
REPEAT; X = X+1; NEXT IF X<100
```

This simple **REPEAT** loop increments X until X<100 becomes *false*. If X is already 100 or greater, it is still incremented once (because the test is made after the body of the loop is executed) and the loop terminates.

The **IF** <*condition*> *is* not an **IF** statement, but a loop termination test that is performed at the end of each iteration. The **IF** <*condition*> *is* not an **IF** statement, but a loop termination test that is performed at the end of each iteration. The **IF** <*condition*> *is* optional, and if omitted, creates an *infinite* loop that must be terminated by some operation within the loop itself (e.g., a **RETURN**, **GOTO** or **EXIT** statement).

Like **FOR** and **WHILE** loops **REPEAT** loops:

- Can be nested to any depth, can appear in compound **IF** statements,
- Can be nested in combination with other loop constructs,
- Support loop-relative branching (e.g., **GOTO NEXT**, EXIT or **THEN NEXT**),
- Can be exited with a **RETURN** statement or by error traps.

NEXT *[<index variable>]*

Defines the end of a **FOR**, **WHILE** or **REPEAT** loop and when executed, **NEXT** restarts the loop from the top if the controlling condition is true. In **FOR** loops, a **NEXT** increments the index variable; on **WHILE** and **REPEAT** loops it re-evaluates the condition expression. The optional <*index variable*> can be supplied if the **NEXT** is terminating a **FOR** loop, and

it must be exactly the same index variable defined in the corresponding **FOR** statement. This is a formality however and is useful only for programming clarity and style; faster loop execution actually results without it. An <*index variable*> *is* illegal in **NEXT** statements that terminate **WHILE** and **REPEAT** loops.

There must be one and only one **NEXT** statement associated with each **FOR**, **WHILE** and **REPEAT** statement throughout your program. Each loop (**FOR**, **WHILE** or **REPEAT**) many contain any number of inner loops, which may themselves contain lower level inner loops. These are called *nested* loops because each inner loop is completely enclosed (or *nested)* within the loop outside it. An error will result if you overlap loops without one enclosing the other. **FOR**, **WHILE** and **REPEAT** loops may be nested in any combination and to any nested depth.

NEXT may also be used in an entirely different context: as a line-label. **NEXT** as a label always refers to the actual closing **NEXT** statement of the current **FOR**, **WHILE** or **REPEAT** loop underway. This can be quite useful in situations when the **NEXT** statement is not the first one on the line, for example:

```
10 For l=1 to N; IF ARRAY(I)>LIMIT then NEXT; Print l; Next
```

Any control statement that can specify a line number or label may also specify **NEXT** as a (pseudo) label, although it is unlikely that it will be used in statements other than some form of **GOTO**. An *Unexpected Next Error* results if the **NEXT** pseudo label is encountered when no loop is currently active.

EXIT *[<label>]*

EXIT statements are used for terminating any **FOR**, **WHILE** or **REPEAT** loop currently in progress, without waiting for *normal* loop termination. An optional line number or label can be specified to tell MegaBasic where to resume program execution after the loop is terminated. Omitting the line reference causes program execution to resume at the first statement that follows the *closing* **NEXT** statement of the loop terminated. For example:

```
For l=1 to 100; If ARRAY(I)=X then exit; Next
```

This loop terminates if the index value increments past 100, or if ARRAY(I)=X. An error results if no **FOR, WHILE** or **REPEAT** loop is currently active. **EXIT** without the <*label*> *is* a clean, easy method to immediately terminate a loop without using **GOTOS**.

An **EXIT** to the *pseudo label* **NEXT** (i.e., **EXIT NEXT**) Will terminate the currently active loop and begin the next iteration of the loop outside of that. An error results if an **EXIT NEXT** is encountered without being immediately surrounded by two or more active loops.

Multiple loop levels can be exited by repeating the **EXIT** keyword by the number of loop levels desired, for example:

```
Exit exit exit 150
```

This exits three loop levels, then branches to line 150. As you might expect, leaving off the line reference causes program execution to resume at the statement which follows the **NEXT** of the highest level loop exited. Multi-level **EXITS** are especially useful in complex nested looping applications where using **GOTOS** to exit loops is either undesirable or impossible.

Section 4: Error Trapping and Control

With the possible exception of extremely simple *throw away* programs, real life programming applications need to be able to tolerate and handle unexpected situations. Files that were supposed to be there, but weren't; user's requesting things that are not within the realm of possibility; calculations exceeding the bounds of numerical or mathematical representation. These and many other unforeseen possibilities can and do arise, and your program must have contingency plans to handle them.

MegaBasic provides several powerful mechanisms to control errors, as summarized in the table below. These mechanisms have been designed to take advantage of the structure of your program in a way that lets you handle errors at the appropriate level of execution in which they arise. This section cover these methods and how best to apply them.

ERRSET	Error trapping and trap generation
RETRY	Automatic retry control on system errors
WAIT	Timed delay generation

Most other programming languages handle errors by explicitly generating and passing error codes around the program. When an error occurs, an error code that identifies the kind of error is generated and passed to the operation that caused it. If the operation can deal with this error, it branches off to do so, but otherwise passes another error code back up to the operation that called it, and so on... The difficulty with this approach is that the details of passing error codes back from level to level can be very complex and error prone in itself and requires a great deal of programmer attention to do correctly if it's to be done at all.

MegaBasic greatly simplifies this process by *automatically* passing error codes and descriptions up to context that actually uses the error description to take concrete recovery actions. It restores the execution context at the error trap level no matter how many levels of loops and subroutine calls were piled on top of the error situation. None of the intervening program code needs to be concerned with error processing at all.

Error Trapping and Program Structure

It is useful to view program execution as a layered structure, much like an onion skin, to understand the error recovery techniques of MegaBasic. Invoking a subroutine descends one level; returning climbs back up a level. An important aspect to these levels is that a higher level should not require *knowledge* of the internal workings of lower levels in order to use them. For example, you don't need to understand how square-roots are computed in order to use the SQRT( ) function. Furthermore, since subroutines may be called from numerous places throughout the program, they certainly have no *knowledge* about the immediate context from which they were called.

When you implement a complex program with many levels, you have to isolate each level from the others so that you can develop, test and debug it independently. MegaBasic **ERRSET** operation supports this goal by letting you deal with program errors on each level independently. Errors in a multi-level program structure can be detected and handled differently at each level of the program.

For example the following list of error responses might be assigned to the same error depending upon which level of the program *sees*

Subroutine Level	Possible Response
Query Processing	Insert Disk into Drive B:
Build report	Unsupportedselection
Find the employee record	No field of that name
Access the index	Index field unavailable
OPEN file statement	File not found Error

The MegaBasic **ERRSET** statement provides independent error control at every level of program execution. Each **GOSUB**, user-defined function or procedure, loop or **CASE**-block may independently control its own **ERRSET** traps without affecting traps set by higher levels of your program. If a lower level subroutine does not set any traps of its own, traps defined at higher levels will control the lower level errors.

Such a transfer out of a lower level to a higher one is fully supported and constitutes the only legal and viable way to bypass the normal **RETURN** mechanism. For example if **GOSUB** 100 sets a trap then calls **GOSUB** 200 which in turn generates an error, the original trap is used. However if **GOSUB** 200 sets its own trap before the error was encountered then that trap is used. Upon **RETURN**, **GOSUB** 100 is still protected by its original trap, unaffected by any **ERRSETS** within **GOSUB** 200.

What this means for applying **ERRSETS** is that the error trap line must be a line at the same program level as the **ERRSET** statement that assigns it. Specifically, avoid assigning error traps that can jump out of the current **GOBUS**, user-defined function or procedure that contains the **ERRSET** itself. Instead, assign a trap to a line within the same structure. A useful way to understand and remember this concept is to think of an **ERRSET** as a special form of **GOTO**. If an **ERRSET** were replaced with a **GOTO** to the same line and that **GOTO** is legal (i.e., it doesn't jump out of the subroutine) then the **ERRSET** is also legal and proper.

For example, to transfer control out of a **GOSUB** when an error occurs within the **GOSUB**, execute the **ERRSET** statement prior to entering the **GOSUB**, with an error trap referring to a line also outside the **GOSUB** (at the same level as the **ERRSET** statement). This is actually the most straight forward and easily debugged method for constructing error traps within a procedure-oriented language of any type. MegaBasic cannot enforce these rules when the **ERRSETS** are made, so you must be careful to apply them properly.

When an error trap occurs, MegaBasic restores the entire state of program execution that existed when the trap was set. This means that the current context of local variables at whatever depth in subroutines is lost and the original state, as of setting the trap, is restored. This provides an air-tight mechanism for error recovery that is repeatable in all situations.

Furthermore, MegaBasic restores the error trap in effect at the time the current subroutine was entered, rather than disabling the entire error recovery system after an error occurs. Errors which occur before another trap is set are trapped at the higher level. In other words, if you set an error trap before entering a subroutine, your program will remain in control even if the subroutine fails to trap its own errors. An **ERRSET** statement without arguments will also restore this higher-level trap instead of nullifying the recovery system.

If PARAM(21) is set to non-zero, high-level error reporting mode is enabled for the MegaBasic package that PARAM(21) was set in (Chapter 9, Section 5). In this mode, untrapped errors are not reported as occurring within that package. Instead, the errors are reported to have occurred in the reference that called the routine in an outside package.

ERRSET *[<trap label> [, <diagnostic vbls>]]*

When an error occurs during program execution, MegaBasic normally prints an error message and the line number in which it occurred, then terminates your program. With **ERRSET**, you can set up an *error trap* that takes control when an error occurs, allowing you to control what happens after an error occurs so that your program can continue without interruption. For example, the function below uses an **ERRSET** to determine if a string represents a valid number:

```
10 Def integer func VALID(V$)
20 Errset TRAP; V = val(V$); Return 1
30 TRAP: Return 0; Func end
```

An error trap is simply a line at which the flow of control begins in the event of an error. Both the **ERRSET** statement and the <*trap label*> it defines must reside within the same subroutine level (i.e., same function, same procedure, same main program level etc.). Error traps are locally defined and neither affect nor prohibit error traps potentially set at higher levels of the program. When properly used, **ERRSETS** support the development of fault-tolerant programs.

Each of the **ERRSET** arguments is described below. All arguments are optional and when omitted, they must be omitted from right to left. For example, there must be a <*trap label*> in order to include <*diagnostic vbls*>.

No arguments	Omitting all **ERRSET** parameters causes the **ERRSET** in effect when the current subroutine or loop level was entered to be restored. If no **ERRSETS** were in effect a that time, then traps are turned off
<trap label>	Line number or line-label specifying the program line to go to in the event of an error. Within loops, pseudo-label **NEXT** may be specified. The label or line number must specify a line within the same subroutine (or main program) as the **ERRSET** statement itself.
<diagnostic variables>	A set of numeric variables can be specified after the <*trap label*>. When the **ERRSET** trap is taken, MegaBasic stores information about the error into these variables, which your program can then read in order to make informed error recovery decisions.

Up to three diagnostic variables can be specified, separated by commas, and you can omit them from right to left. Their meaning is positionally defined as:

- line number where the error occurred,
- error type code of the error, and
- the error message string.

These variables are supported for compatibility with earlier versions of MegaBasic. They are not necessary in new programs because several system functions are supported that return error information in even more detail, as described below:

ERRLINE	Line number in which the error occurred. **ERRLINE**(l) returns the relative statement number on which that error occurred in that line and **ERRLINE** or **ERRLINE**(0) returns the line number itself.
ERRPKG$	Name of the package or work space where the error occurred.
ERRTYP	Error type code of the error. Appendix A contains a complete listing of error messages and codes.
ERRMSG$	Error message string that would have been displayed, had no **ERRSET** trap been in effect. Only the descriptive part of the message is returned.
ERRDEV	Device or file number selected at the time the error occurred. The error may or may not be related to I/O, but when it is, knowing the open device channel can be useful.

You need to use these functions very soon after an error occurs because their values can change if another error or a Ctrl-C occurs before you access them. In addition to its line number, the relative *statement numbers* on the line is useful after an error on a line with many statements and **ERRLINE**(l) returns this number. Statement numbers are not available from *compiled* programs, where **ERRLINE**( ) with any argument returns the same value as **ERRLINE** with no argument. In compiled programs, you can determine the relative statement number by reporting error *addresses* instead of *line numbers* and looking in the *.map* file to figure which line and statement the reported *address* corresponds to.

ERRSET	Restores the error trap in effect when the current program level was entered. That higher level will then be responsible for errors in the current level. This is commonly done to turn off error traps no longer needed. Returning from a function or subroutine will automatically disable its error traps in the same manner.
ERRSET 125	Sets an error trap so that execution will branch to line 125 if any trappable error should occur. This trap remains in effect until the current subroutine **RETURNS** or until redefined by another **ERRSET**. A line-label or line number can be
ERRSET 125,L,T,M$	Same as the previous form except that variable L is set to the line number in which the error occurred, T is set to the error type code of error that occurred and M$ set to the error message. These variables can be omitted from right to left.

When an error occurs and an error trap is defined, the following sequence of actions takes place:

- Restores the program execution context of the most recent **ERRSET**, exiting all levels of subroutines and loops active since that **ERRSET**, and restoring parameters and local variables as if the intervening subroutine levels returned normally.
- All information about the error is made available to the system error functions and stored in any <*diagnostic vbls*> specified in the **ERRSET** statement.

- Restores the prior **ERRSET** in effect before the current subroutine or loop level was entered. If no prior high-level error trap was in effect, no trap is defined and subsequent errors abort the program.
- Program execution restarts at the first statement on the line specified for error recovery (i.e., the <*trap label*> given in the **ERRSET** statement). Your program has now regained control.

Specifically, an error trap set within a subroutine, loop or **CASE** block remains active only as long that program structure is active. After that, the previous error trap, if any, regains control over errors. The important points about **ERRSET** error trapping are summarized below:

- **ERRSET** traps within loops (**FOR**, **WHILE** and **REPEAT**), **CASE** statements, **GOSUBS**, functions and procedures are active only as long as that program structure remains active.
- When an error occurs, the trap taken is the **ERRSET** trap defined at the lowest program level that is at or above the level on which the error occurred. All active program structures between the **ERRSET** level and the level of the error are terminated (no matter how deep), and execution then resumes at the line defined by the **ERRSET** statement. At this point, any **ERRSET** traps defined at higher levels will still be active.
- An **ERRSET** trap can be changed or disabled only by executing another **ERRSET** statement at the same level as the earlier **ERRSET**. For example, from inside a **FOR**..**NEXT** loop, you cannot change or disable an **ERRSET** trap set before the loop was entered. An **ERRSET** statement with no arguments will disable the **ERRSET** trap at the current program level.
- **ERRSET** traps defined within loops last only as long as the iteration in which they were defined. At the end of each iteration, the error trap active before the loop began is always restored. If an error trap must be active within a loop throughout all iterations, its **ERRSET** statement must be invoked at the start of every loop iteration.
- All active **ERRSET** trap levels are displayed by the TRACE RET command, along with loops, **CASE** statements and subroutine invocations.

ERRSET traps are very fast, due to the fact that the process of resolving an error location into a line and statement number is deferred to the point where this information is actually needed (i.e., in **ERRLINE** references, setting the **ERRSET** recovery line number variable and displaying built-in error messages). Often this location is not needed (e.g., as when basing a decision on merely the presence or absence of an error), so such **ERRSET** traps proceed much faster. A typical example of this is using the VAL( ) function to determine if a string represents a valid number.

To assist the program development and debugging process, MegaBasic *does not trap type 10 errors* when programs are **RUN** from the MegaBasic *command level*. Type 10 errors are those involving errors in program formation, e.g., syntax errors, loop construction, etc. Such errors need to be exposed during program testing and not hidden by the error processing mechanisms, as they would be if they were *trappable* errors. Such errors are always trapped when the program is run from the operating system command level.

ERRSET #<er ror type> [,<error message exprn>]

Generates an error of the given type for purposes of debugging and special program control applications. You must have an active **ERRSET** in effect at the time this statement is executed, otherwise a *User Trap Error is* issued. For example: **ERRSET** #9 generates a *Divide By Zero Error* just as if your program had divided a number by zero. The *<er ror type>* can be a numeric expression that evaluates to a value between 1 and 255. **ERRSET** #0 is reserved for the special purpose of clearing the information returned by the **ERRTYP**, **ERRMSGS**, **ERRLINE** and **ERRPKG$** functions (no error is generated in this case).

One reason for *causing* errors with this statement is to *break out* of the current subroutine level and return to a specific program location at a higher level in your program without having to return normally. Such an action is not possible using **GOTO** statements because functions and subroutines are considered to be active until a **RETURN** statement is executed. If a program trap has been set by an **ERRSET** statement at some higher level, you can always break out of any pile of active subroutines (and loops) and resume at the trap location by causing any trappable error. **ERRSET**# is an effective way to generate pseudo errors at any time.

The *User Trap Error* message, generated by **ERRSET** #errcode, may be customized by augmenting the **ERRSET**# statement with an error message string expression, not to exceed 30 characters. For example, **ERRSET #99, INVALID MATRIX** generates a user-defined error which if trapped, returns error code 99 and if untrapped generates the message:

```
INVALID MATRIX Error in Line xxx
```

This feature is intended to further support the implementation of self sufficient procedures and functions (user-defined) which behave and appear as though they are part of the MegaBasic built-in set. By generating errors which, with or without **ERRSET** traps set, appear and behave in an identical manner to those generated by the built-in features, program interfacing is standardized and simplified. As with built-in errors, the system variable **ERRTYP** is set to the error code specified, and the system string variable **ERRMSG$** set to the optional (or default) error message given.

Custom error messages defined within a subroutine should be documented along with the other interface rules, such as the argument list definition and the global data structures it uses. You should word your message carefully so that it *reads well* in the context of a MegaBasic error message.

When defining your own error types, it would be wise to assign error codes well above the range already defined for built-in MegaBasic errors, so that your program can discriminate between them. Additional error types will be added to MegaBasic from time to time and therefore a good range for your own custom error codes might be 100 to 255. However, this is a rule that is not enforced that you can use to avoid any possible conflict with the pre-defined MegaBasic error codes of the present and the future.

RETRY *[<pr ocedure* name> *]*

Defines a procedure which is called when certain errors occur which can potentially be recovered from by retrying the operation that lead to the error. For example, a *Not Ready Error* (type 25) occurs if the printer happens to run out of paper or it is not powered-up when your program attempts to use it. In such a case, a **RETRY** procedure is called (if defined) to provide a programmed response to the situation, giving the user a message and allowing the process to be retried.

RETRY procedures only work for some errors and they require support by the host operating system to inform MegaBasic of the various error conditions from which recovery is possible. Multi-tasking operating systems such as MP / M-86, TurboDos-86 and Concurrent CP / M are among the operating systems supported. **MS-DOS** partially supports **RETRY** and CP / M 86 version 1.1 does not provide any support. Multi-tasking operating systems provide temporary *locks* on system resources (e.g., disks, files, printers, records within files, etc.), which are supported by MegaBasic. If your program attempts to use any locked resources, it must wait until those resources become available (unlocked). The **RETRY** facility in MegaBasic is designed to provide a simple and effective means for synchronizing with such events. The following errors may be retried (see Appendix A for further details):

Type	Error Message
25	Not Ready Error
26	File in use Error Error
27	Non-recoverable Disk
28	Read-Only Error
29	Operating System Error
32	Suspended file access
33	Error Disk unavailable Error

RETRY defines the retry procedure as the name specified, or disables the retry procedure currently defined if no name is specified. **RETRY** may be used at any time to redefine the current retry procedure in effect. Like **ERRSET**, **RETRY** *is local* to the current subroutine level and therefore when a **RETURN** statement *is* executed, the retry procedure defined (or not) at the calling level is restored and back into effect. Hence subroutines have the freedom of defining their own retry procedures as required without upsetting those defined at higher program levels.

The procedure name specified must be the name of a known procedure which does not have any argument list in its **DEF** statement. When the procedure gains control after a retriable error, the **ERRTYP** system variable will contain the error type being trapped, the system string variable **ERRMSG$** will contain the error message phrase, and the **RETRY** function (i.e., not the statement) will return the number of retries that have taken place on this particular error. To retry the operation that caused the error, just **RETURN** normally from the procedure with a **RETURN** statement.

The retry procedure *must not* perform any file operations of any kind when a file operation is the cause of the retry procedure call. To do so will very likely lead to a corruption of the subsequent retry and cause potential damage to any file currently OPEN. It should also avoid doing anything which could lead to the same error that invoked it because **RETRY** procedures are disabled while one is currently active. Your retry process may find it useful to use the **WAIT** statement (Chapter 6, Section 4) to generate timed delays before resuming with retries. This statement depends on the wait function only provided by multi-tasking operating systems and not supported in all single user systems. Retry procedures should restrict themselves to using the **RETRY** count and the **ERRTYP** code to decide on when and how to inform the user, and when to retry and when to abort.

If after some number of retries you wish to give up, you must execute an **ERRSET** # *<error type>* statement to generate your own custom error (described earlier). This is necessary because you can exit a MegaBasic subroutine only by **RETURNing** or

generating an error. In the event that no retry procedure was in effect, an error would have been generated anyway, so this response is consistent and justified. If the program has an **ERRSET** trap in effect, the generated error will be trapped and execution will continue. If no trap is active, then the program will terminate with the error.

The following example illustrates these concepts using the *Not Ready Error* generated when a report printout is attempted:

```
100 Retry INFORM; Rem -- define retry procedure
120 Gosub PRINT_REPORT; Rem -- call the report generator
800 Rem -- Retry procedure for not ready errors
810 Def proc INFORM
820 If errtyp<>25 then Errset #errtyp,"Improper Retry
830 If retry>20 then Errset #25,"Not Readyn; Rem up tO 20 retries
840 Print "Printer not available, type any key when fixed:n,
850 V$ inchr$(0); Return; Rem -- Retry operation
860 Proc end
```

A retry procedure should handle all possible errors that invoke it by using the **ERRTYP** code to branch to one of various routines for each type. When the retry procedure **RETURNS**, it resumes the internal process that was in progress, right where it left off, rather that restarting the MegaBasic statement that lead to the error. Because of this, the recovery process is invisible to the statement that lead to the error, eliminating the possibility of errors introduced by restarting actions already partially complete.

In all situations where retry procedures are useful, **ERRSET** traps could also be employed. However to use **ERRSETS** for such purposes, you would have to speically program an **ERRRSET** trap for each statement that contains potential for retries. A **RETRY** procedure is normally defined once in the initialization of the program and generally requires no further attention. Furthermore, retries controlled by **ERRSETs** necessarily involve restarting the offending statement from the beginning, even though it might have been partially completed. This requires that such recovery methods be carefully programmed to ensure that such statements will yield correct results every time.

Caution Using RETRY

You have to be very carteful within MegaBasic **RETRY** procedures that trap and process errors for operations you intend to retry. First, do not do anything that performs any **DOS** *operating system calls* (e.g., file **OPEN**, **CLOSE**, **READ**, **WRITE**, **TIME$**, **DATE$**, **WAIT**, **PRINT**, **INPUT**, etc.), except for keyboard input and screen output and direct **ROM BIOS** calls. If you violate this rule, **DOS** is left in an unstable state by the system call and upon invoking the subsequent retry, the system will probably crash (a fault with **DOS** that cannot be avoided).

Second, avoid any operations that could cause a reorganization of MegaBasic memory structures because the operation being retried may be relying on an absolute address to a memory structure which, if moved during the **RETRY** procedure, can *invalidate* the address being used. Operations that can cause a memory reorganizaiton include **DIMensioning** variables, accessing uninitialized variables and using a *lot* of scratchpad space (e.g., processing big string expressions, doing subroutine calls to great depths, etc.)

WAIT<number of seconds>

Generates a time-out delay specified by the number of seconds given. During the period specified, MegaBasic is not executing and Ctrll-C will not be detected until the **WAIT** is finished. The number of seconds may include fractions of a second down to 1 millisecond. Its actual resolution is system dependent but generally the time will always be within 60 milliseconds of the time specified.

The **MS-DOS** versions implement the **WAIT** statement with additional timer accuracy that correctly resolves to within 1 millisecond. However, do not rely on the accuracy of any such timinings when running MegaBasic under Microsoft WINDOWS because the timing base is unpredictable.

WAIT statements are especially useful within **RETRY** procedures to slow down the rate at which retries are performed. For example, one retry every two seconds would be sufficient to wait for a locked file to become unlocked.

WAIT is supported under multi-tasking operating systems (e.g., MP / M-86 , TurboDos-86 and Concurrent CP / M) and under **MS-DOS**. Under single-user operating systems it is supported if the system maintains a running clock with at least one-second interval updates. If your system does not support the **WAIT** statement and you need such a capability, use a **FOR..NEXT** loop with nothing inside it and an appropriate inside it and an appropriate iteration count to implement delays.

Chapter 7 I/O and System Interaction

This section discusses the MegaBasic statements available for accessing data files, for character device input and output and for interacting with external system processes and services, as summarized in the table below. See Chapter 2 for the description of the notation used to specify command and statement formats employed in this section. See Chapter 9 for all information about additional I/O and system functions.

Input and Output	Console interaction, formatting strings and numbers, text file processing and serial device control.
File Processing	Create and destroy files, serial and random access to data and to file attributes.
System Interface	Altering MegaBasic system parameters, direct access to memory, I/O ports, to machine-level system calls and other system resources.
Logical Interrupts	Support for asynchronous *event-driven* processes, useful for multi-tasking, background processing and real-time process control

In all situations where retry procedures are useful **ERRSET** traps could also be employed. However to use **ERRSETS** for such purposes, you would have to specially program an **ERRSET** trap for each statement that contains potential for retries. A **RETRY** procedure is normally defined once in the initialization of the program and generally requires no further attention. Furthermore, retries controlled by **ERRSETS** necessarily involve restarting the offending statement from the beginning, even though it might have been partially completed. This requires that such recovery methods be carefully programmed to ensure that such statements will yield correct results every time.

Caution using RETRY

You have to be very careful within MegaBasic **RETRY** procedures that trap and process errors for operations you intend to retry. First, do not do anything that performs any **DOS** *operating system calls* (e.g., file **OPEN**, **CLOSE**, **READ**, **WRITE**, **TIME$**, **DATE$**, **WAIT**, **PRINT**, **INPUT**, etc), except for keyboard input and screen output and direct ROM BIOS calls. If you violate this rule, **DOS** is left in an unstable state by the system call and upon invoking the subsequent retry, the system will probably crash (a fault with **DOS** that cannot be avoided).

Second, avoid any operations that could cause a reorganization of MegaBasic memory structures because the operation being retried may be relying on an absolute address to a memory structure which, if moved during the **RETRY** procedure, can *invalidate* the address being used. Operations that can cause a memory reorganization include DIMensioning variables, accessing uninitialized variables and using a *lot* of scratchpad space (e.g., processing big string expressions, doing subroutine calls to great depths, etc).

WAIT *<number of seconds>*

Generates a time-out delay specified by the number of seconds given. During the period specified, MegaBasic is not executing and Ctrl-C will not be detected until the **WAIT** is finished. The number of seconds may include fractions of a second down to 1 millisecond. Its actual resolution is system dependent but generally the time will always be within 60 milliseconds of the time specified.

The **MS-DOS** versions implement the **WAIT** statement with additional timer accuracy that correctly resolves to within 1 millisecond. However, do not rely on the accuracy of any such timings when running MegaBasic under Microsoft WINDOWS because the timing base is unpredictable.

WAIT statements are especially useful within **RETRY** procedures to slow down the rate at which retries are performed. For example, one retry every two seconds would be sufficient to wait for a locked file to become unlocked.

WAIT is supported under multi-tasking operating systems (e.g., MP / M-86, TurboDos-86 and Concurrent CP / M) and under MS DC6. Under single-user operating systems it is supported if the system maintains a running clock with at least one-second interval updates. If your system does not support the **WAIT** statement and you need such a capability, use a **FOR..NEXT** loop with nothing inside it and an appropriate iteration count to implement delays.

Section 1: Input and Output Statements

This section describes the character stream I/O statements and console facilities of MegaBasic, which are summarized as follows:

PRINT #<dev>,<data list>	Formats a sequence of numbers and strings and sends the result to an output channel. The extensive formatting capabilities are covered in full detail.
INPUT #<dev>,<input list>	Statement for requesting and receiving data interactively from the user. Input editing is supported. Input received is validated and stored into numeric or string variables.
EDIT$ = <string exprn>	Loads the input editing buffer in preparation for an editing session controlled by the **INPUT** statement.
ENTER <input source>	Redirects console input from a different source (i.e., text file or device name). Subsequent input from device #0 is taken from that source until it runs out and reverts back to the normal console.
IOCTL #<dev>,<ctrl string>	Outputs special channel control sequences to channels that support such controls. This is highly dependent on the system configuration of the host machine.

General file operations are described in Chapter 7, Section 2, which include a number of important statements useful in the context of the current discussion (i.e., **OPEN**, **CLOSE** and **FILEPOS**). Chapter 9, Section 4 provides complete operating details on the built-in functions related to both I/O and file operations. The functions described there related to just I/O are summarized below for easy reference:

Pos(D)	Returns the column position on channel D.
Line(D)	Returns the line position on channel D.
Inchr$(D..)	Returns input characters from channel D.
Edit$	Returns the previous line input, or command tail.
Input(D)	Returns the input status of input channel D.
Output(D)	Returns the output status of output channel
Ioctl(D)	Indicates whether channel D supports ctrl strings.
Ioctl$(D,C$)	Returns control string input from channel D.

MegaBasic provides 32 I/O channels through which character streams are transferred to and from your MegaBasic program. An I/O channel is simply a connection (implemented in hardware and software) between your program and a *device* that accepts output characters one at a time, or provides input characters one at a time. This process is called stream I/O, because such data transfers usually involve many characters traveling through the device, resembling a stream of characters. Your computer console screen and keyboard is a typical example of such a device. Characters are input one at a time from the keyboard and output to the screen one at a time. Together, this input and output capability are combined into one I/O device called the *console device*.

Your program communicates through only one I/O channel at any time and because there are many I/O channels to choose from, an I/O channel must be selected for use before any information transfer can pass through it. Therefore each I/O channel is assigned a unique identifying number called its *channel number,* which is specified in each MegaBasic statement or function that performs I/O with the corresponding I/O channel. Channel numbers range from 0 to 31 and they are defined as follows:

0	Console screen and keyboard
1	Main system printer (usually output only)
2	Auxiliary I/O device (bi-directional)
3 - 31	User Defined

Channel numbers 0,1 and 2 are the built-in I/O channels which are provided by the host operating system. These channels are always present, assuming that your operating system has been implemented on your machine to support them. At any time, you can transfer information between these devices by specifying the channel number 0,1 or 2 in the MegaBasic command, statement or function requiring the data transfer. The syntax for specifying channel numbers to MegaBasic facilities will described shortly.

Channel numbers in the range 3 to 31 are set aside for user defined I/O channels. Such channels must be defined before they are used, with the MegaBasic **OPEN** statement (Section 2 of this Chapter). The **OPEN** statement creates a temporary I/O channel between your program and an arbitrary device or file already present and maintained by the operating system, and assigns a **channel** number to it for use in subsequent I/O operations. Such I/O channels become undefined and their channel numbers available for re-assignment once you **CLOSE** the channel (Section 2 of this Chapter) or your program ends (Chapter 6, Section 1).

Since channels may be either files or *physical* character devices, **you can** redirect output from your program to storage (files) or onto actual peripheral devices by simply diverting output to different channel numbers or changing an **OPEN** statement. The I/O statements themselves are defined in generic terms which lend themselves to both device and file I/O without favoring one over the other. It is the channel number itself that determines the destination of output and source of input to your program.

The actual devices that your program can **OPEN** are highly system dependent. Under the CP/M class of operating systems (i.e., CP/M-86, MP/M-86 and CCP/M), there are unfortunately no devices supported other than the built in devices (0,1 and 2). Only files can be **OPENed** on channel numbers 3 to 31 under these systems. However, the **MS-DOS** class of operating systems provide a sophisticated facility for attaching arbitrary I/O device drivers to the system at startup time, which you can access by name, somewhat like file names. Such devices can be **OPENed** by your MegaBasic program and used for I/O purposes just like any other device (or file).

Channel numbers are specified in MegaBasic commands and statements by giving the channel number preceded by a Ib-sign (#). For example the built-in channels are specified as #0, #1, and #2. The Ib-sign is needed because such channel numbers are optional and it informs MegaBasic that the number specified is to be interpreted as a channel number and not to be confused with other numbers that may also appear in the same construct. When the channel number is omitted, channel #0 is assumed by default. Therefore is it not necessary to specify a channel number when transferring data to and from the console device. A comma separates a channel number from other data or arguments that follow in the same statement.

In commands, channel numbers must be specified with integer constants (i.e., an error occurs if you specify fractional quantities or numeric expressions in this context). However no such restriction is placed on channel numbers in program statements, which can even compute channel numbers using numeric expressions if necessary. Non-integer channel numbers are truncated to the next lower integer value as they occur. Channel numbers supplied to I/O functions may also be specified as numeric expressions, but no Ib-sign (#) should be placed in front of the number, as it is in statements. Better performance results if you specify channel numbers using integer, rather than floating point, expressions, but both work.

*PRINT [#<channel>,]<data list> [,*NOMARK*]*

Causes the data list specified to be output as characters to the channel specified (or the default channel). The **PRINT** statement is the primary character output statement of MegaBasic. **WRITE** statements can also be used but they are intended for data file output. Although it goes by the name of **PRINT**, this statement is not merely a printer operation, but a general stream output statement for use with any channel number. The <*data list*> describes the items to be printed and how they are to be formatted. It consists of the following items, which must be separated with commas when more than one appears in the **PRINT** statement:

Numbers	Numeric values are converted to display codes and sent to the **PRINT** channel. Their format can be controlled via a format specification that precedes them in the data list. Numbers may be specified with numeric expressions, constants, etc.
Strings	Strings are sent to the **PRINT** channel after being formatted as needed. They may be specified using any general string expression, constant, etc.
Vectors	Vectors and vector expressions are **PRINTed** element by element and formatted accordingly. Such items must be preceded by the word VEC in order to be identified as vectors. See the material starting in Chapter 3, Section 7 for complete information on vectors.
Format Specifications	Controls the appearance and layout of numbers and strings following it in the data list. Formats consist of a percent sign (%) to indicate that a format specification is coming up, followed by a string expression (commonly a string constant) evaluating to a valid format description.
Control Specifications	These are special purpose items inserted into the data list to perform useful operations as the **PRINT** list is being processed, such as tab control and blank line generation.

MegaBasic scans the data list, printing numbers and strings as they come and attending to format and control specifications as encountered. Numbers and string expressions are evaluated, then printed in the currently defined format, which may be redefined at any point in the data list. At the beginning of the **PRINT** statement, the currently defined format is the default format, which may also be re-defined at any point.

Normally a carriage return is generated on completion of a **PRINT** statement, but this can be suppressed by terminating any **PRINT** with a comma. This allows several **PRINT** statements to contribute to the formation of a single line. Because of how frequently **PRINT** statements generally appear in most programs, the **PRINT** keyword can be replaced by an exclamation mark (!) for brevity. This notation performs the identical function that **PRINT** does.

Format Specifications

Format specifications control the appearance of numbers and strings as they are printed. For example you may want to be able to control their position on the page, restrict the number of decimals displayed, select standard or scientific notation, breakup large numbers with commas, put dollar signs in front of numbers, or justify a string on the left or right of a fixed-width field. You can specify multiple options with a single format specification. A format specification appears in **PRINT** statement preceding the values to be formatted.

Format specifications consist of a percent sign (%), indicating that a format specifier is next in the **PRINT** list, followed by a string expression that evaluates to a format description string. When MegaBasic encounters a format string expression, it evaluates the expression and remembers the result to control the format of subsequent items in the **PRINT** statement. Nothing is printed when a format string is encountered. Format strings can be virtually any length, limited only by the available scratch pad memory remaining (up to 55k). A format specification affects all strings that follow it in the data list until another format is encountered, or the data list ends.

In most instances, the format string expression will merely consist of a string constant containing some fixed format, rather than as a large, complex string expression that dynamically computes a different format based on prevailing conditions. Such flexibility will be discussed, but for now we shall confine ourselves to the simple static format case. One such example is as follows:

```
Print %“c15f2”, X, Y, Z
```

which prints the values of X, Y and Z with 2 decimal places, commas to the left of the decimal, and right justified in a field of 15 character positions. Applying this to values such as 4325, 0.3665, 5893432.567 and 0, the following display would be presented:

–4,324.00	**.37**	**5,893,432.57**	**.00**

The *f* in the format string is called *the format mode character*, which selects the type of format to be applied. There are six numeric format modes and three string format modes, which are all described on the following pages. Most format modes can be further augmented by various format *modifiers*. For example, the c in the format string above modifies the f-mode to include commas in large numbers. All modifiers will be discussed shortly.

In the coming pages, we will discuss each of the various format modes and their modifiers. Then we will show how they can be combined together to form more complex format descriptions with a minimum of effort. Understanding how to specify formats involves many things, which you should learn one at a time. Try them out as you are reading about them; make up your own examples and experiment with them until you feel comfortable with each concept.

Formatting Numbers

If you do not include any numeric format specifications in a **PRINT** statement, MegaBasic will display all numbers in the default format. You can designate any format specification to be the default, but it is normally a special format called free-form, in which numbers are displayed in the following manner:

- A single space as a separator from preceding values
- A minus sign (–) if negative, but no plus sign (+) if positive
- All leading digits and trailing decimals to full precision

MegaBasic switches to E-notation for numbers that are very large or very small, but most numbers are shown the way you would normally expect them to appear. Free-form is useful when numbers are displayed within unformatted text (like words in sentences) and for quick displays such as those needed during the test and debugging phase of program development. But free-form cannot be used for numbers that need to line-up in columns and, many times, it displays numbers with more digits to the right of the decimal than you would otherwise want.

So if free-form format is not desired, you must explicitly specify a different numeric format. Numeric formats are denoted explicitly in two forms:

```
<width> <mode> <decimals>

<width> <mode> <places>
```

<width>	total number of print columns to provide for the number. format mode character, one of the letters I, F, *E, H, O* number of trailing digits to the right of the decimal.
<mode>	Minimum number of digit places that must appear in the numeric value.
<decimals>	number of trailing digits to the right of the decimal.
<places>	Minimum number of digit places that must appear in the numeric value.

The example format string *c15f2* illustrated earlier, specifies a <*width*> of 15 column positions, a <*mode*> character of *f* (called an F-specification), and 2 trailing <*decimals*>. Each of these components are discussed in detail below.

<width>

The format field width specifies a fixed number of print columns (or positions) to use when displaying the formatted number. This width is specified as an unsigned integer from 1 to 120, and must be wide enough to accommodate the largest number to be printed in that field, including all decimals, any decimal point, commas, number signs, dollar signs, leading spaces and any other characters that will appear in the field.

Numbers are always positioned up against the right-hand side of such fixed-width fields so that they will line-up when placed in columns. Spaces (i.e., blanks) are used to fill out the left side of the field to preserve the fixed-width. If you specify a field width that is too narrow for some number to fit in, MegaBasic will display the field filled with asterisks instead of the number to indicate a programming error that should be corrected. However *no formal* error will be reported so that your program *can* continue on.

The field width is optional and if you omit it from a format, the numbers are formatted with one leading space and however many additional print columns are necessary to display the number, which may be determined by other specifications in the format string. Such variable width fields are useful within unformatted text, as in paragraphs and sentences, similar to free form.

<mode>

The format mode is a single character, in upper or lower case, which selects the type of numeric format to be used. Six different modes are available, which include fixed decimal (F), scientific or E-notation (E), integer (I), octal (O), binary (B) and hexadecimal (H). All mode characters may be preceded by the optional field <*width*>, described above, and followed by an optional <*decimals*> or <*places*> specification, which are described below. Each <*mode*> is individually discussed on the net page.

<decimals>

This is the fixed number of digits to the right of the decimal point that you want displayed in the number. For example, dollar values with pennies should be displayed with two decimals. You can specify any number of decimals from 0 to 80 and each number is displayed rounded to the nearest value with the exact number of decimals specified. All decimals requested by the format are shown even if they are zeros. This specification only applies to E and F format modes. Omitting the <*decimals*> from either of these modes is equivalent to specifying zero.

<places>

This optional value is placed to the right of the format mode character to specify the minimum number of digit places to show, even if that means extra leading zeros in front of the number (which are normally suppressed). It applies only to the integer format modes (I, H, O and B) and is usually specified to force leading zeros in front of numbers. Numbers are normally printed with leading zeros suppressed. If you specify more <*places*> than you have room for in your fixed <*width*>, a *Format Specification Error* will occur.

Numeric Format Modes

Each of the format modes are fully described below. For notational purposes, the letter w will stand for the field <*width*> value, *r will* stand for the <*decimals*> value, and p will stand for the <*places*> value. Each of these values are optional but when supplied, they must be given as unsigned integers.

Numeric Formatting Modes	
wFr	Right justifies a number with *r* trailing decimals to the right within a field w columns wide. Free-form fixed decimal layout is formed by omitting the width (w). For example: **PRINT** *%"12F3",3476.6* displays as *3476.600* preceded by 4 spaces to make a total width of 12 columns. By omitting *r-decimals* part or specifying zero causes the number to print rounded to an integer, with no decimals and no decimal point displayed.
wEr	Same as wFr except that E-notation (scientific) is used. This notation prints a base value (X.XXXXX...) followed by a power-of-ten scaling factor (E+XX or E–XX) called the exponent. The exponent always appears as the last 4 characters of the specified field. For example: **PRINT** *%"12E5",3476.6 will* display the value 3.47660E+03 with just one space in front of it. See **<mode>** above for a discussion of E-notation. When the field width is specified (w), it must be at least *r+7* to provide enough room for the entire value.

wIp	Right justifies an integer in a field w columns wide. No trailing decimals can be specified. Non-integral values are rounded prior to printing. Free-form integers are generated by omitting the width (w). For example: **PRINT** *%"12I",3476.6* displays the integer *3477* with 8 spaces in front of it, using a total of 12 display columns. By omitting the *12*, only one space is placed in front regardless of the number size. The *p-option* (places) is specified only to force a minimum number of digit places, usually to include leading zeros in the number. For example **PRINT** *%"1217",3476.6* displays 0003477 right justified in a field 12 columns wide. No leading zeros appear if the number itself takes up or exceeds the place-count specified.
wHp	Same as the preceding wIp format except the number is displayed in *hexadecimal*, i.e., base 16, which is useful in systems programming applications. No minus sign is displayed when negative numbers are formatted with this mode. Instead, negative 32-bit values are shown in hexadecimal twos-complement notation. Free-form hexadecimal numbers are displayed when both the *width (w)* and *places (p)* options are omitted. Non-integer values formatted in this manner are truncated to the next lower integer value before being displayed, rather than rounded as in the wIp format. Format modifiers, described later, have no effect within H, O and *B* format modes.
WOP	Same as the preceding wHp format except that the number is displayed in *octal*, i.e., base 8, useful is certain systems programming applications. The format mode character is the *letter O*, not the digit zero.
WBP	Same as the wHp format except that the number is displayed in *binary*, i.e., base 2, useful in applications using bit strings as well as in systems programming applications. Up to 32 columns may be required to show all the digits of some binary numbers (e.g., –1 displays as 32 places of ones).

Format Modifiers

The following set of special format modifying characters may be included within a format string to produce additional features such as dollar signs, comma grouping (e.g., 1,435,801), zero suppression, etc. Such modifiers consist of single characters which are placed within a format specification string to invoke the desired effect.

Several modifiers may appear in a format string for their combined effect; their order of appearance is of no significance. You can specify a modified free- form format by listing all the desired modifiers without specifying any format mode (i.e., no mode implies free-form). Each format modifier is described below:

Numeric Format Modifiers	
$	Places a dollar sign ($) to the left of each number printed. When a leading numeric sign (+ or –) appears with the dollar sign ($), the sign comes first, followed by the dollar sign. Be sure to provide sufficient room in your field widths (w) to allow for the dollar sign. The $ modifier applies only to the F, E and I format modes. In hexadecimal, octal and binary formats, $ includes the radix sign character on the numbers (i.e., h for hexadecimal, o for octal and b for binary). The radix sign will be appended to the formatted number whenever you specify a $ format modifier. Be sure to account for this extra character in any width specification affected by it.

C	On *F* and I format modes, C inserts commas every three places (left of the decimal) after 1000. Remember that these commas take up space in your specified widths. On the E-format mode, the C-modifier produces a variant of E-notation known as *engineering notation,* rather than insert commas. The exponent of numbers in engineering notation is always a multiple of three, and the value portion is a number from 1.0 to 999. Such values are much easier to comprehend in the same way that numbers with commas are easier to understand. When using this format option, you must be sure that the format width (i.e., number of columns) is at least 8 more than the number of decimals specified. Engineering notation is also used for numbers printed *in free-form comma* format for *large values* requiring a switch to E-notation.
Z	Suppresses trailing zeros to the right of the decimal. Trailing zeros are changed to spaces (blanks). If the format does not include a width specification (w), then these spaces will not appear in the field. The Z-modifier applies only to the *F* and E format modes. On string formats (described later), the Z-modifier suppresses trailing spaces generated on formatted strings. Note that this shortens the field and is thus primarily useful only on the last string of a printed line.
+	Indicates positive numbers with a plus sign (+), the same way as negative numbers are shown with a minus sign (–). All numbers will be printed with a numeric sign, regardless of their value. The + modifier applies only to the *F,* E and I format modes.
T	Positions the *sign* of the number, if shown, to the right of the number (called a *trailing sign),* instead to the left (a *leading sign)* . The sign will appear as the last character of the specified field. When applied to fixed-width fields, all numbers are shifted over one column to the left to provide room for the sign. Non-negative numbers in such fields therefore have one space in the last column of their field (instead of a sign). The T-modifier applies only to *F* and I format modes, but it has no effect on negative values when the A-modifier is also present.
A	Provides *accounting format* for negative numbers, which are shown in parentheses rather than given a minus sign. When applied to fixed-width fields, all numbers are shifted over one column to the left to provide room for the closing parenthesis. Non-negative numbers in such fields therefore have one space in the last column of their field (instead of parentheses). The *A-modifier* applies only to the *F* and I format modes.
N	Suppresses the display of zero values by filling the numeric field with blanks. This is useful for enhancing numeric displays that consist of mostly zeros, e.g., sparse matrices. The *N-modifier* applies only to the *F, E,* and I format modes.
*****	Changes all leading blanks of formatted numbers to asterisks. For example the format *15F2$i* would format the value 5354.249 as ******* $5354.25. This modifier has no effect on any of the *Hex, Octal* or *Binary* formats and is provided for use in check-writing applications.
#	Causes the format specification it is within to become the *default format* as well as the *current format.* This is not really a format modifier since it has no *modifying* effect on the current format. Used alone in the specification, # sets the default format to free-form numeric output (if it was not already). This may be used for a particular format that occurs frequently throughout your program. Once you make it the default format, you never again need to specify it explicitly in your **PRINT** statements, because the default format is used whenever no format is specified.
D	Selects the default format as the **current** format. All immediately preceding format modifiers are lost, so this modifier should be first when more than one is supplied. Additional specifications and modifiers that follow will alter this new current format as specified. Think of the *D-modifier* as shorthand for the default format. The default format is always *unmodified free-form* unless your program changes using the Ib-sign *#-modifier* described above.

Altering Format Attributes

The characters used for currency ($), decimal points (.) and comma separators (,) in formatted numbers can be changed during execution. This is done within format strings by following certain format modifiers with an equals sign (=) and the desired ASCII code. These codes are described in the table below:

$=n	Defines the ASCII code to use for dollar signs.	**"F2 $=33 C"**	**!1,234,567.89**
C=n	Defines the ASCII code to use for - commas.	**"F2 $=37 C=32"**	**%1 234 567.89**
P=n	Defines the ASCII code to use for decimal points.	**"F2 $=33 C=46 P=44"**	**!1.234.567,89**
G=n	Defines the comma-break grouping size.	**"F2 $=37 C=46 P=44 G=2"**	**%1.23.45.67,89**

The values after the equal sign specify the ASCII code (in decimal) to use for the respective usage. Note that P is a modifier used only for changing the decimal point character and G is used only to specify the number of digits within comma groupings (C). These changes become permanent for all subsequently formatted numbers within the same MegaBasic package from which they were reassigned, so to restore them you have to re-specify their original settings.

Automatic Numeric Scaling

Another **PRINT** format modifier is supported that shifts numbers left or right any number of places before they are **PRINT**ed. This eliminates the need to explicitly scale numbers using multiply or divide before they are **PRINT**ed, simplifying your program and making it faster (i.e., the internal scaling does not perform any multiplies or divides). This modifier consists of a < or > followed by the number of digit places to shift. To shift left, use <; to shift right, use > (i.e., the number is shifted the direction of the arrow). The following examples illustrate how this works:

Print % "15F2>2",X,Y,Z	Prints values shifted to the right two places (i.e., divided by 100).
Print % "201<3",X,Y,Z	Prints values shifted to the left three places (i.e., multiplied by 1000).
Print % "8H>3",X,Y,Z	Prints hexadecimal values shifted to the right 3 hex digits (i.e., divided by 2^12).
Print % "16B<4",X,Y,Z	Prints binary values shifted to the left 4 binary digits (i.e., multiplied by 2^4).

The scaling modifier works with all numeric format modes (i.e., I, F, E, H, O, B) and it has no effect on string formats (i.e., L, M, R). When the binary, hex or octal modes are scaled, the digits *that fall off* the end of the number are lost and no error is reported for this. When any of the decimal modes are scaled, the resulting scaled number must remain in the range of valid floating point values (a *BCD* limitation that is not in the IEEE version).

When used, scaling factors should follow the numeric format mode specification. They could be placed in front, but then a space has to separate the shift count from the format width that would follow. You can specify a scaling factor all by itself (or with other modifiers) to scale a number printed in free-form.

Multiple Formats and Format Rescan

Up to this point we have described formats that cause a series of values to be printed the same way. But suppose, for example, that your **PRINT** statement will format six numbers in three pairs, such that each pair consists of an integer and a fixed-decimal value. Since the format changes on each successive number, you would have to specify a format string preceding each of the six numbers, even though only two different formats are actually needed. Such a **PRINT** statement would appear as follows:

```
PRINT %“8i”,1,%“12f2”,X,%“8i”,J,
%“12f2”,Y,% “8i”,K,%“12f2”,Z
```

To eliminate such cumbersome notation, MegaBasic allows multiple format specifications to be packed into one format string, which are distributed in a round-robin fashion to successive values being printed. This technique lets you avoid all the redundant repetition of identical format specifications in situations like the one just described, which can be programmed as follows:

```
PRINT %“8i,12f2”,1,X,J,Y,K,Z
```

Although both **PRINT** statements produce identical results, the later is obviously easier to type and understand. The individual formats within a multiple format string must be separated from one another with commas, as shown above. Spaces may be inserted anywhere within a format string to improve readability but they have no other significance.

MegaBasic simply applies successive formats from the string to the successive values as they are encountered and displayed. If the format string *runs out* of formats before all the values have been printed, MegaBasic cycles back to the first format in the string and continues cycling through the formats until all values have been printed. This is called format rescan. If more formats are supplied than the number of values to be printed, the extra formats are never used and no error is reported.

You can specify the free-form format in a multiple format string as an *empty* format, i.e., two commas in a row with nothing in between (,,). The default format may be specified simply as the letter D, a format modifier described earlier.

Multiple formats can be useful even when every number being printed uses an entirely different format. Shorter data lists result if you define one long format string in the **PRINT** statement instead of many separate short ones, and they are generally easier to read and understand. Long format strings can be built and stored in string variables, so that subsequent **PRINT** statements need only refer to their names to apply the multiple format (e.g., **PRINT** %FMT$,X,Y,..)

Multiple format strings have the additional flexibility to intersperse line breaks and arbitrary character sequences at any point between formatted items. Line breaks and blank lines can be generated by specifying one or more slashes (e.g., / /) as one of the items in the format string. Each slash generates a carriage return, line feed sequence; two or more slashes generates blank lines. Slashes are a separate item in the format string, and as such, they must be separated from other items in the string by commas. For example, to print the vector X(*) so that 8 values are printed on each line, the following **PRINT** statement might be used:

```
PRINT %“8i,12f2,8i,12f2,8i,12f2,8i,12f2”, /, VEC X(*)
```

In the same way that slashes can be inserted anywhere, you can also insert any string constant in between formatted items. This is done by specifying a quoted string constant as one of the items in the multiple format string. Either quote character can be used (" or '). For example, to divide the lines generated by the format example above with a vertical bar (I), the following **PRINT** statement would be specified:

```
PRINT %"8i,12f2,8i,12f2,' | ',8i,12f2,8i,12f2,/", VEC X(*)
```

Any characters (or *control characters)* can be included within such constants except for the quote character used at both ends. An error occurs if the quote at either end is omitted. This item in a format string is called *a format literal,* and it can be used for printing telephone and *social* security numbers and other numbers that *contain* certain non-numeric characters within them.

When MegaBasic prints a formatted number or string, it first prints any slashes (/) and format literals that are encountered in the format string until an actual format specification is encountered. After the last data item has been printed, MegaBasic generates any slashes and literals that immediately follow the last format specification (and precede the next specification). The same action is taken if no data items are specified in the **PRINT** statement.

Format Repetition

To simplify the construction of more complicated format descriptions, the concept of format repetition can be applied. In a format string, you can cause any sequence of format items to be repeated by surrounding the sequence with parentheses ( ), preceded by the repetition count. For example, the vector print example above can be simplified using format repetition as follows:

```
PRINT %"4(8i,12f2),/", VEC X(*)
```

where the format sequence "8i,12f2" is effectively repeated 4 times. Format repetition can be nested: repetition inside of repetition. For example, the above **PRINT** statement can be augmented to print a blank line between every group of five output lines, as follows:

```
PRINT %"5(4(8i,12f2),/),1", VEC X(*)
```

You can specify up to 5 levels of nested format repetition. MegaBasic reports an error if you specify more than five, or if the parentheses are not balanced, or if you omit the repetition count. Properly designed nested format strings can be used to print entire pages using a single **PRINT** statement.

Dynamic Formatting

Computing format specifications at run-time, instead of using static fixed format string constants, is known as *dynamic formatting,* which can make use of information available at the time of the **PRINT** statement in constructing the format string. For example your format may change depending on how much data is to be printed, its range of values, and the characteristics of the channel receiving the output. If you wish to print X with commas, zero-suppression, dollar sign, right justified in a field W characters wide with D decimals, use the statement:

```
PRINT %"$ZC"+STR$(W)+"F"+STR$(D),X
```

Dynamic formatting can be a complex task that requires care and planning. User-defined string functions are useful here to hide the details of format construction and provide access to your various formatting processes by name. Functions also collect the format decision-making into centralized places, confining future changes to a limited area of your program.

Formatting Strings

Strings are normally printed exactly as given in the data list and additional spacing may be programmed as needed. Using the wide variety of string operations provided in MegaBasic, you have great control and flexibility over the format of strings. Formatting entire displays exclusively with string operations can be a very powerful way to control the appearance of your output. The STR$( ) function (Chapter 9, Section 3), which converts a number into a string form, provides all the support necessary for combining numeric and string information into formatted data ready to display.

There are, however, several simple string formats that are commonly required in many applications, and hence are provided in MegaBasic: left and right justification and centering. The format specifications for these capabilities are described below using the same notational conventions as those employed to describe the numeric format specifications earlier in this section.

String Format Modes

Each of the string format modes is fully described below. For notational purposes, the letter w will stand for the field <*width*> value. If the string does not fit into the field width specified, the right-most characters that do not fit are discarded. If the string is shorter than the specified field, it is positioned within the field according to the format mode (Left, Right or Middle), filling the unused field positions with blanks.

wL	Left justifies a string in a field w columns wide. If the string is shorter than w characters, additional spaces are output to fill out the length.
wR	Right justifies a string in a field w columns wide. If the string is shorter than w characters, the proper number of spaces required are printed, followed by the string itself.
wM	Middle justifies (centers) a string within a field of w columns. If the string is shorter than the given field width, an equal number of spaces are printed before and after the string is printed to fill out the field exactly.

The following examples should clarify their use:

"15L"	Left justifies a string in a 15-character field.
"12R"	Right justifies a string in a 12-character field.
"78M"	Centers a string in a 78-character field.

When MegaBasic encounters a number when a string format is specified, the number is printed in free-form format and the string format is then applied to the next item in the output list. When MegaBasic encounters a string when a numeric format is specified, the string is printed as-is (unformatted) and the numeric format is then applied to the next item in the output list. This allows your program to continue in the face of format type errors, and lets you insert unformatted numbers and strings into print statements with minimal affects on the format strings being applied.

When the last item of a print statement is L(eft) or M(iddle) formatted, the line will usually end with trailing spaces. If you do not wish this to occur, you can specify the Z modifier to suppress the trailing spaces on any formatted string. Note that this shortens the field and is thus primarily useful only on the last string of a printed line.

Control Specifications

Special control specifications may also appear in the <*data list*>. These are not format specifications and are not preceded by a percent (%). A plus sign (+) resets the line count for the channel to zero before proceeding and is necessary only in applications where this count is being used with the *LINES()* function. The plus sign (+) does not generate any printed characters and has nothing to do with the similar format modifier (+).

Multiple blank lines may be generated from a single **PRINT** statement by a field of slashes, similar to FORTRAN format statements. For example: *PRINT #D,///, will* generate 3 carriage returns on channel D. Slashes may be interspersed throughout the data list.

MegaBasic generates a carriage return at the end of the **PRINT** statement *unless you suppress it* by ending the statement with a comma. For example **PRINT** x displays X and a carriage return, while **PRINTX**, displays X without a carriage return so that later **PRINT** statements can continue on the same line.

TAB(P) advances the cursor to column position P prior to printing the next item, where P is a numeric expression that evaluates to a value from 0 to 255. This is accomplished by printing spaces until the desired position is reached. TAB(P) is ignored if P is less than or equal to the current position.

Printing to Files

When printing to channels 3-31, you are really transferring data to a file **OPEN**ed under the same file number. Exactly the same data is transferred to the file as would be displayed on the console if channel #0 were employed. However it can be important that the last byte printed before the file is **CLOSEd** be an appropriate end-of-file mark so that the file can be processed correctly by other programs. The MegaBasic endmark (an 8-bit value of 26, ASCII CTRL-Z) is placed automatically after each **PRINT** for this purpose if enabled. See **PARAM**(9) to use a different file endmark. You can suppress this endmark from being written to the file by typing the reserved word **NOMARK** as the last item in the data list of the **PRINT** statement you want it suppressed on. See the **NOMARK** statement for other information.

This is appropriate for later **INPUT** processing by MegaBasic programs, but typical file processing programs external to MegaBasic sometimes expect other endmarks. For example ASCII codes 0,1, 26 and 255 are common. You must handle this situation by redefining the endmark code with PARAM(9) (Chapter 9, Section 5). Note that whatever code is used cannot be part of the text printed without causing a false end-of-file condition when later processed.

INPUT *[#<channel number>,] <input list>*

This statement inputs text lines from channel 0 to 31 and stores them into program variables. If the variable is numeric, then MegaBasic attempts to convert the text line into a number. String variables receive the text line as is. If the input channel is an *interactive* device, such as the console screen and keyboard (device #0), then input can be edited using the MegaBasic line editor keys (Chapter 9, Section 5). As such, no input is accepted or acted upon by your program until you type the **ENTER** key (or carriage return). An edited input line can be as long as 254 characters or as short as zero characters (by typing only the **ENTER** key).

A simple console **INPUT** statement that requests a string, two numbers and another string (i.e., four inputs in all) might appear as follows:

```
INPUT A$,X,Y,B$
```

In this example, we omitted the channel number to select the console by default. **INPUT** can accept numbers in any form that MegaBasic recognizes as valid numeric constants. This includes signed and unsigned numbers with and without decimals, E-notation, octal, binary and hexadecimal. **INPUT** will not accept invalid numbers and automatically re-requests a new response from the user for any numeric input that is out of range or not a valid number or includes decimals on values destined for integer variables. You can input numbers into several numeric variables with a single input consisting of several numbers separated by commas or spaces. See Chapter 3, Section 2 for a complete discussion of numeric constants under MegaBasic.

You can also input a number into a string variable, but in this case, the number is simply treated as an arbitrary sequence of characters, i.e., no numeric validation is performed. String variables accept the entire line of input, even if it contains spaces, commas, numbers or words and phrases.

Input strings larger than the **DIMensioned** size of the input variable will be truncated to fit the string. Inputs into indexed string variables (or into string fields) that are shorter than the region indexed are stored left justified in the fixed-width region, padding all remaining character positions to the right of the characters input with spaces.

INPUT statements provide all the editing capabilities of the MegaBasic line editor (Chapter 1, Section 6) for each input. Although you can potentially **INPUT** data on a *hard-copy* terminal (i.e., on paper instead of a screen), the line editor assumes that a screen is being used. On a hard-copy device the editing process will cause severe misalignment of characters if any insertions, deletions or backward cursor movements are attempted.

Input Prompts

An input *prompt* is a message that is output to an interactive input channel so that the user knows that an input response is required. They also usually include additional information about the kind of input expected. The INPUT statement lets you specify an input prompt in front of any input variable in the <*input list*>, for example:

```
INPUT “Enter a number ”,X, “Type a string ”,A$
```

You can specify prompts as any string expression that does not begin with a variable or user-defined function name. This is so that MegaBasic can tell prompts and input variables apart. If your prompt is in a string variable, you can surround it with parentheses to specify the prompt variable in an input statement. MegaBasic always re-displays the prompt when invalid numeric responses are re-requested.

If you do not specify an input prompt for any particular input variable, MegaBasic automatically provides the prompt message: ? . To suppress any prompt messages, including the automatic question mark, specify a null string ( ) as your prompt. If the user types several numbers in a single input response, only the initial prompt for the first numeric input variable appears; the unneeded prompts are suppressed. For example:

```
INPUT “1st value = ”,X, “2nd value = ”,Y, “3rd value = ”,Z
```

If, in response to the input request above, you type all three values separated by commas or spaces in one input line, the second two prompts never appear on the screen and the program continues on. You could also type one input value, then type two input values, suppressing only the prompt for variable Z. This lets you skip ahead in an input sequence that you have been through many times: sort of an *expert* mode. Note, however, this only works for numeric inputs, not string inputs, and only within a single **INPUT** statement.

In order to maintain the correct column position for the console, the maximum input string you can enter is limited to 255 characters minus the length of the **INPUT** prompt. Therefore, really long prompt strings can prevent you from entering all the characters to may wish to.

Building Input from Prior Input

Input entry is usually the major *bottleneck* when using a computer program. One facility that MegaBasic provides to help reduce input keystrokes is the access to the previous non-blank input line using Ctrl-R or F5 editing keys (Chapter 1, Section 6). Once accessed, you can edit this prior input to create a new input entry. Also, the most recently entered entered input line is brought up automatically on the screen if the very first character you type is an editing control character (i.e., not an input character).

The previously entered line is not the only prior input you have access to. MegaBasic also remembers many of the most recent lines of text input through the console keyboard so that you can retrieve any of them whenever you are entering keyboard INPUT. This is particularly useful when you find yourself entering the same or similar line repeatedly because you can avoid having to retype the entire line each time. MegaBasic only remembers one instance of each line entered and keeps them in a most-recently-used order for convenient access. Lines that differ only in upper / lower case and number of spaces are treated as the same line and only the most recent *rendition is* remembered. Null lines (i.e., those without any characters) are never retained.

You access previously entered lines by typing one of several control keys at any time while you are entering a text line into MegaBasic (or into a MegaBasic program). Two keys let you move forward or backward through the line list; one key lets you return to the original line and one key deletes a line from the list. Once a line is accessed, you can immediately begin editing it without any further keystrokes. At any time you can discard your current line and start over on a different line by simply accessing another line and continuing. See Chapter 1, Section 6 for details on these control characters.

Editing Variables with Default Values

Another facility provided by the **INPUT** statement can be used to actually edit the current contents of a variable. This is especially useful for inputs where the program can *second-guess* the input response with a default response. Consider the following example:

```
INPUT “Do you wish to continue? ”,EDIT ANSWER$
```

As with any **INPUT** statement, this statement asks a question and stores the result into a string variable (**ANSWER$**). But notice the word *EDIT* in front of the variable. This causes the current contents of **ANSWER$** to be displayed on the screen with the cursor positioned over the first character, before accepting the user s response. If the user types **ENTER** (or carriage return), the variable is returned unaltered. If the user deletes or inserts characters into the display, whatever is shown when the **ENTER** key is typed is returned in the variable.

Your program can, of course, store the string Yes or *No* into the **ANSWER$** variable before issuing the **INPUT** statement. If the contents of **ANSWER$** happens to be what the user was about to type, the response need only be a single keystroke (i.e., typing the **RETURN** or **ENTER** key). If the response differs only slightly from the contents of the variable, the user can easily edit the visible entry already present on the screen into the desired response, before typing the **ENTER** key. To edit this value, the first key you type Must be an editing control key, rather than an *ordinary* input character. If the first key is an input character, the current default entry shown vanishes from the screen and is replaced by the key struck. This makes it unnecessary to delete the entry first when all you want to do is simply type a different entry without editing.

Any input variable can be edited by preceding its name with the EDIT keyword. Variables without this modifier will be input the usual way. You can edit numeric variables in this manner as well. Numbers are displayed using the default format currently in effect (explained earlier in this section) and with all leading and trailing blanks removed. Do not use a numeric format that inserts commas into large numbers because that would divide the input into several numbers (e.g., 1,234,567 is interpreted as three numbers).

Suppressing Input Echo

If edited and echoed input is not desired, use the **INCHR$**( ) function, described in Chapter 9, Section 4. Also, you can specify **INPUT** statements in three different forms to control carriage return and character echo, as follows:

INPUT	Echoes all input keys typed during each input entry.
INPUT1	Suppresses the carriage return echo after each input.
INPUT2	Suppresses all echo and editing control key action.

INPUT2 works just like the INPUT1 statement except that characters input are not echoed to the console or other specified channel. Editing control keys are not recognized and input as normal characters and input is terminated with a carriage return. This is ideal for the input of passwords or other applications where echo suppression is desirable. All three forms of **INPUT** behave in an identical manner when inputting from an *open text file:* suppressing the echo of character and carriage return input and disabling all input prompts.

Input from Files

When inputting from a non-interactive *device,* such as a file (opened under channel 3 to 31) or read-only device, the **INPUT** statement operates as follows:

- All prompts are suppressed and **EDIT** keywords are ignored.
- For each string variable being input, a complete line is read from the file. This consists of all characters from the current file position up to the next carriage return, end of file mark, or physical end of file (which ever comes first). This terminator is not included in the line input, nor in the next line input. Thus a sequence of carriage returns is **INPUT** as a sequence of null strings.
- For each numeric variable, MegaBasic reads the file from the current file position up to the end of the number. Numbers must be terminated by commas, spaces, tabs, linefeeds or carriage returns. All leading control characters and separators are ignored.

- Carriage returns and linefeeds (ASCII codes 13 and 10) are treated differently when they appear in pairs. A CR-LF sequence collapses into a single CR code (ignoring the LF). An LF-CR sequence collapses into a single LF (the CR is ignored and does not terminate the line).
- An error occurs if you attempt to **INPUT** a numeric variable and no valid number is present at the current file position. If numbers in the file contains commas, dollar signs ($) or other extraneous characters, then numeric input is not directly possible and you must input such values as strings and extract the values with string operations. Numbers with decimals input into integer variables are truncated.

An error occurs if your program attempts to **INPUT** a string when the first character is the *end-of-file mark* (26 code) or past the last file byte. You can test for this condition by testing the **INPUT**( ) function (Chapter 9, Section 4) for a value of zero before each **INPUT** line (Section 1 of this Chapter), or by trapping the error with the **ERRSET** statement (Chapter 6, Section 4). To recognized a different endmark code, use PARAM(9) to redefine it (Chapter 9, Section 5).

ENTER *<console input source>*

Redirects console input from a different, specified source text file or character device. MegaBasic takes all subsequent console keyboard input from the input file specified, until the file runs out or a **BYE, END** or **DOS** command is encountered. If the input file runs out or an untrappable error occurs, the normal keyboard is re-established so that subsequent commands are taken from the keyboard. **ENTER** can be used either as a command or as an executable statement within a program. As an executable statement, the *<console input source>* can be specified as a general string expression.

This capability is useful for *re-playing* a keystroke sequence for any purpose, such as automating a sequence of MegaBasic commands or providing automatic responses to an executing program for testing or demonstration purposes. Only DOS compatible versions of MegaBasic currently support this feature. Enter works exactly like the **DOS** *command-level* input redirection mechanism (e.g., **BASIC < CONSOLE.INP**).

EDIT$ = *<string exprn>*

Evaluates the string expression and places it into the previous-line buffer so that editing control keys can be used on it in a subsequent **INPUT** statement, for example:

This permits editing of the string produced by the concatenation of A$, B$ and C$ without prior entry of that string from the keyboard. See Chapter 1, Section 6 for further details on editing data as it is entered. The **INPUT** statement itself supports a much more powerful method for editing the current contents of program variables. The current contents of the previous-line buffer is always available from the EDIT$ function (no arguments).

Because MegaBasic maintains a list of **most-recently-entered lines, rather** than a single line, setting **EDIT$** (e.g., **EDIT$** = *string)*, adds a new most-recent line to the line list. Setting **EDIT$** several times in succession adds several lines to the list, which can be useful for pre-loading the buffer in preparation for a subsequent input entry. You cannot overflow this buffer because MegaBasic makes room for new entries by automatically deleting the oldest lines in the buffer as needed. Normally this buffer is 512 bytes long, but you can resize it at any time by setting **PARAM**(24) to any size from 0 to 4096. The buffer is always cleared to empty every time you set **PARAM**(24) (Chapter 9, Section 5)

Command-Level Arguments

Whenever you execute a program, either from the MegaBasic command level or the operating system command level, you can follow the command that starts the program with additional characters on the same line. This sequence of extra characters is called the *command tail*, and MegaBasic places it into the *old line* buffer when program execution begins so that you can retrieve them (using the **EDIT**$ function) as needed by your program, extracting any additional arguments it contains. For example, suppose you run a program from the operating system using the command:

```
BASIC Program data1 data2 data3
```

When *Program* begin execution, **EDIT$** will return the additional data typed as the string: *Program data1 data2 data3*. This command *tail* must be used before your program requests console input via the **INPUT** statement, because the edit buffer is then overwritten and its prior contents are lost. See the **EDIT$** function (Chapter 9, Section 4) for special considerations about accessing the command tail string and using **EDIT$** in general.

IOCTL #*<channel number>, <control string>*

Transmits a control string to the device opened under the specified channel number. If the device does not support control strings then no action is performed and no error is reported. Control string operations are supported only under the **MS-DOS** operating systems.

Some device drivers under **MS-DOS** and other operating systems can accept and generate special control information called I/O control strings. These strings allow programs to control the behavior of the device like baud rates, stop bits, communication protocols, character translation, internal buffering, timing characteristics, etc. Special status information may also be obtained from a device as input control strings. Control strings are passed between application programs and devices via a special I/O channel called the IOCTL channel which is provided by the device along side its *normal* character I/O transfer channel. Control strings are simply character strings that are transferred through this special I/O channel. The length and content of control strings is determined by the the design of the device driver itself; it is really a special command language designed specifically for a specific device. Its definition must therefore be obtained from documentation associated with the device and is generally a customized capability peculiar to a specific device driver.

Once your program has opened up a device under some MegaBasic channel number using the **OPEN** statement, your program can then access the I/O control string capabilities of the device. The following table lists the other control string facilities provided by MegaBasic:

IOCTL(D)	This function asks the device opened under channel number D if it can process control strings. The function result is Yes (1) or No (°)
IOCTL$(D)	This function inputs a control string from channel number D. If the device does not process control strings then a null string is returned.
IOCTL$(D,C$)	This function outputs control string C$ to channel number D and then returns an input response control string from the device, i.e., send a command and return the acknowledgement response.

Section 2: File Processing Statements

MegaBasic provides a complete set of file operations that your program can apply to accomplish any desired file processing task. Most operations refer to files either by name or by open channel number. Each statement performs a relatively simple operation, all of which are summarized in the table that follows, giving you a view of the total range of possibilities before delving into the detailed discussion of each file processing statement.

CREATE	Creates a file under some new specified name.
DESTROY	Permanently deletes files from the system. The storage area they occupied then becomes available for use by other files.
RENAME	Changes the name of a file to another unique name.
DIR and DIR$	Provideslistings of file directories and access to subdirectories and user numbers.
OPEN	Opens an existing file for subsequent access and assigns an open channel number. Files can be opened under a variety of access levels to limit the kinds of operations permitted.
OPENC	Similar to **OPEN** except that files opened for output are created automatically if they do not already exist, plus other automatic features.
CLOSE	Closes a file that has been previously opened. This ensures that all revisions made to the file while open are posted to the permanent recorded copy of the file on the storage device.
FILPOS(F)	Sets the file position of open channel E File positions can also be set within **READ** and **WRITE**.
FILESIZE(F)	Sets the absolute file size under open channel F to a longer or shorter length.
READ	Accessesinformation in an open file and transfers it to variables in your program.
WRITE	Transfers variables or computation results to specific locations of an open file. Information can written anywhere within a file or appended to the end of a file.
LOCK UNLOCK	Locks and unlocks regions of files opened in shared mode under multi-user or network environments.

This section of the manual describes the facilities in MegaBasic for accessing and storing data on files maintained by your operating system. You should be familiar with your particular operating system, its capabilities and user facilities for basic file operations.

What is a File?

A file is simply a sequence of bytes or characters stored on some mass storage device, which is maintained by the operating system as an individual data object, rather than as separate bytes of unrelated information. Files may be of any length up to some limit imposed by the physical size of the storage device and the configuration of the operating system. Dividing large amounts of data into a set of files permits easier handling of the data, similar to the way that manual systems break up data into individual files containing related information.

Since many files can be maintained by the operating system, each one is assigned it own name, unique from the rest, so that any particular file can be identified when it is needed for later access. File names are assigned by people and by computer software when the file is created for the first time, and the name assigned usually contains some indication of the purpose of the file or its contents. File names are expressed in MegaBasic programs using string expressions, and in commands as a series of characters without quotes around them. Consult your operating system manual for details concerning file names, file types and internal file structures.

Accessing Files

In MegaBasic, as in other programming languages, a file must be opened before your program can access its contents. This is similar to manual systems, where files must be withdrawn from a file cabinet drawer and opened to view before the data they contain can be accessed. Opening a computer data file causes MegaBasic to build a number of internal control structures that provide efficient, direct access to the file without having to give the file name for each operation. Your program can have up to 32 files open simultaneously. An open file is identified in your program by its open channel number, an integer from 0 to 31 which is assigned to a file when it is initially opened.

Once a file is open, you can read from it, write to it, determine or change its size, find out the date and time that it was last modified, etc. Since a file is just a sequence of bytes on a storage medium, the location of any particular byte in a file is called its *byte position*. The first byte in a file is always at byte position 0, the second byte is at position 1, and so on up to the last byte of the file. Byte position numbers are important because all data transfers occur at specific byte positions in the file. You have to specify file positions to access data located at random locations scattered about your file.

Sequential and Random File Access

For each open file, MegaBasic maintains a special byte position pointer which is called the *current file position*. Whenever you do not specify the byte position of a data transfer, MegaBasic performs the transfer at the current file position. Upon completion of the transfer, the current file position is advanced to the byte position *following* the last byte transferred to the file. Hence, if you never specify a byte position when transferring data, successive transfers are performed in ascending file locations. Accessing files in this manner is called *sequential file access*.

Another way to access a file is called *random* file access, so-called because data is read or written in non-sequential or *random* order. To access data in a file at random, you have to set the file position to the location at which you will be reading or writing. In actual practice, most programs apply a mixture of both sequential and random access methods to accomplish their tasks. For example, you might read alphabetized names from a master data file that is in *random* order, under control of an index file that is read *sequentially*. Random file access will be covered in more detail when we discuss the transfer statements themselves.

When you specify byte positions and open channel numbers using values with decimals (e.g., non-integer values like 523.736 and 0.943), MegaBasic truncates the numbers to the next lower integer before using them (e.g., 523 and 0). An *Out Of Bounds Error* will occur if negative values are specified as file positions. Also, real values specified for any integer purpose such as these is slower than using integer values because MegaBasic has to convert them to integer form before using them. This can be avoided by specifying integer expressions in all integer contexts like open channel numbers, byte positions in files, array subscripts, string indexing, etc. Integers are explained in depth over most of Chapter 3.

MegaBasic is capable of accessing very large files, up to 2,147,483,647 bytes in length. This is greater than the limiting disk size supported by most currently available microcomputers and their operating systems. Even when the ever expanding capabilities of the latest microcomputers eventually support such capacities, few applications are likely to require files of this size.

Text File Processing

The **READ** and **WRITE** statements described in this section are intended for general data file information data transfers. One common class of files that requires special handling is called text files, which contain unformatted lines of words and phrases like the page of text you are now reading. It is convenient to process such files as a sequences of lines, rather than as individual characters or fixed-length records. For this reason you should transfer text information between text files and your MegaBasic program using **PRINT** and **INPUT** statements instead of **WRITE** and **READ** statements. **PRINT** and **INPUT** have been designed specifically to deal with such files in a simple I/O and System Interaction and efficient manner, whereas you would complicate your task by using **READ** and **WRITE** to do the same thing. Chapter 7, Section 1 describes in depth how to apply **PRINT** and **INPUT** to text file processing.

Record-Oriented Files

In some applications, it can be useful to organize a file as a sequence of fixed-length records, where each record consists of a set of data fields. In MegaBasic, you can implement this by reading and writing records as fixed-length strings (see the **READ** and **WRITE** statements about this). Such strings can be built from or divided into their data field components using either standard MegaBasic string processing methods, or by accessing the record string as a structured variable, composed of the desired data fields, then accessing the various fields directly as variables. The subject of structured field variables begins in Chapter 5, Section 3.

Specifying File Names

File names in MegaBasic statements and functions are always specified by string expressions, which means they can be constructed using string computations, or pulled out of string variables. You will often specify file names as simple string constants: just a file name with quotes around it. The exact syntax of file names is defined by the host operating system, but the differences generally affect only the set of characters that are legal as file name characters (letters, digits and some punctuation). File names consist of the following four parts:

<drive code>: <pathname> <primary name> . <extension>

The optional <*drive code*> *is* a letter indicating the physical drive on which the file resides. You can omit this if the file resides on the *default drive* (see your system manual). A colon (:) separates this letter from the rest of the name when the <*drive code*> *is* given. The optional <*pathname*> specifies the directory where the file resides. The <*primary name*> *is* mandatory and consists of 1 to 8 characters. The optional <*extension*> consists of 1 to 3 characters and immediately follows the <*primary name*> with a period (.) in between for separation.

A file name cannot contain question marks (?) or asterisks (*) for matching multiple files. Periods (.) and colons (:) can appear only as separators as shown above, never within the name portions. All letters in file names may be typed in upper or lower case for the same effect. Also, no blanks (spaces) may appear anywhere within a file name; those in the above file name format exist solely for visual purposes and do not appear in actual file names.

Under **MS-DOS** you can specify file names with their directory path. This provides access to files in directories other than the currently selected directory. As with file names, path names can be typed in upper or lower case, but MegaBasic converts any lower case characters to upper case internally. The names within a pathname must be separated by backslashes (\ ) or forward slashes (/), and MegaBasic converts forward slashes to back-slashes before the name is used internally.

Any legal **MS-DOS** pathname is acceptable to MegaBasic. Hence the file .. \ x refers to the file named X in the directory just above the current directory. See your **MS-DOS** operating system users manual for complete information about file pathnames and how to specify them.

File Functions

There are a number of built-in MegaBasic functions (Section 2 of this Chapter) that provide information about files useful to your file processing applications. These functions are summarized below for quick reference.

Inchr$(F,N)	Inputs N bytes from open channel F
Input(F)	True if data can be read, False otherwise
Output(F)	True if data can be output, False otherwise
File(F$)	True if channel F$ exists, False if not
Filepos(F)	Byte position of file opened under channel F
Filesize(F)	Total file size opened under channel F
Filedate$(F)	Date of last update for open channel F
Filetime$(F)	Time of last update for open channel F
Filectrl(F)	Internal system file handle of open channel F
Open$(F)	Full name of the file open under channel F
Space(D)	Total remaining disk space on drive D
Dir$(F$)	File name strings extracted from the directory

CREATE *<new file name>*

Creates a new file of zero size on the disk. Omitting a drive reference from the file name refers to the current default drive. The file name is a string expression which must evaluate to a name not already present in the file directory. *A File Already Exists Error* occurs if an existing file name is specified.

DESTROY *<existing file name>*

Permanently deletes the specified file name from the disk and its directory. The file name is given as a string expression. **DESTROY** statements ignore file names that do not exist, rather than reporting a *File Not Found* error. If you **DESTROY** an open file, its file buffers are not flushed and the file is closed before being deleted to make it possible to easily **DESTROY** a file that has exceeded the available disk space, without causing additional *Out Of Disk Space* errors.

RENAME <old *file name*>,<*new file name*>

Changes the name of a file to a new name, and possibly moves it into another directory (on the same drive). MegaBasic reports an error if either the <*old file name*> doesn't exist (a *File Not Found Error)* or the <*newfile name*> does exist (a *File Already Exists Error).* Both file names *are* given as string expressions.

Under **MS-DOS**, **RENAME** lets you rename a file with a different directory pathname, causing the file to be physically relocated to that directory without any time-consuming copy transfers. Such moves can only occur between directories on the same drive: a *File Creation Error is* reported if you attempt to move a file across drives or network nodes.

DIR$ = <*director y pathname string expression*>

Selects the sub-directory to be used for subsequent file searches, displays, **OPENS, CREATES, DESTROYS**, etc. This statement is only supported for the **MS-DOS** family of operating systems. The string expression must evaluate to a valid **MS-DOS** pathname starting at the root directory. If any of the names in the pathname are not found, no directory change is made and a *File Not Found Error will* result. The root directory can be selected using a pathname string of / . Consult your **MS-DOS** operating system guide for further details on **MS-DOS** pathnames.

This statement also changes the current drive, as needed, in addition to changing the current directory path. You can change the drive without changing the directory path by specifying only the drive part, e.g., "D:" or "C:". Prior to MegaBasic version 5.60, **DIR$**- did not modify the current drive, requiring you to set **PARAM(2)** for that purpose. If you want to set the current directory on another drive without changing drives, you now have to save and restore the current drive using **PARAM(2)**.

When used as a string function, **DIR$** (Chapter 9, Section 4) by itself (without any arguments) will return the current pathname for the default drive. With an argument, **DIR$()** Wi11 extract file names from the directory for further use in directory scanning applications.

DIR = <*user number*>

Selects the *CP/M user nurliber* for all subsequent file operations under the CP / M class of operating systems. The user number is normally zero, but with this statement you can select any user from 0 to 15 (0 to 31 under TurboDos-86). The user number must be selected prior to OPENing any files stored under that user. Files from different user numbers may be OPEN at the same time without conflict. This statement is not supported under the **MS-DOS** family of operating systems because they use sub-directories instead of user numbers. Sub-directories can be selected by the DIR$ statement described above.

DIR *[#<output device>][, <drive>][, <file.ext>]*

Generates a file directory listing from your MegaBasic program. It displays the directory only for the currently selected user number (CP / M) or subdirectory pathname (**MS-DOS**). You can start and stop long displays using the space-bar, as described for the LIST command (Chapter 2, Section 2). **DIR** prints a copy of the file directory from <*drive*> to <*device*>. When omitted, both arguments take on the current default assigned to them.

DIR is provided so that your MegaBasic programs can list file directories; the following examples illustrate its use:

DIR	Displays all the files from the default drive.
DIR 3	Displaysall the files from drive C:
DIR #1, 2	Prints all files from drive B: to the printer.
DIR "pgm"	Displaysall the files with type .PGM from the default drive.

Because channel numbers in the range 3 to 31 perform file operations sending a directory to such a *device* involves simultaneous file requests that may not be supported by the host operating system. You should therefore avoid directing the **DIR** to channels 3 to 31 unless you know that it works properly.

OPEN *[<modifiers>] #<channel>, <file name>*

Provides program access to a file and its contents lasting until the file is *closed* or the program ends. The **OPEN** statement looks up the specified file name in the directory and if found, associates the supplied channel number (0 to 31) with the file for reference by subsequent operations involving that file. An error results if you attempt to **OPEN** a file which is not found as specified in the file directory, or if you try to **OPEN** a file under a channel number which is already assigned to another open file. A file must be **OPENed** before it can be accessed with **READ**, **WRITE**, or other file statements and file functions that refer to a file by channel number (including **PRINT** and **INPUT**).

Under the **MS-DOS** class of operating systems, you can **OPEN** any I/O device driver present on the system. Such devices are not files and any byte positioning and buffering concepts discussed below for files do not apply for open devices. These devices are normally used by **PRINT** and **INPUT** statements which require that you **OPEN** them under a number which is 3 or higher (0,1 and 2 are built-in I/O channels which cannot be reassigned). Device drivers have names which, like file names, are specified with a string expression in the **OPEN** statement to identify what to **OPEN**. Device names do not appear in the file directory and you need to know what their names are in order to **OPEN** them, a subject that should be covered in documentation relating to your particular system.

The current file position of a newly **OPENed** file will always be at byte position zero (at the beginning of the file). So if you start writing data to a file immediately after **OPENing** it, you will overwrite the previous contents stored at the beginning of the file. You must set the byte position of the file to the location you desire the file transfer to take place. For example, to append data to the end of a file immediately after opening it, you would set the byte position to the byte size of the file before proceeding with the transfer operation. See the **FILEPOS**( ) and **FILESIZE**( ) functions for further information. Byte positions can be specified within the data list of the **READ** and **WRITE** statements described in this section.

Your program should **CLOSE** files (Section 2 of this Chapter) which are no longer needed by subsequent processes. Closing a file makes its channel number available for use by another **OPEN** file and ensures that all changes made to the file are properly posted to the file directory maintained by the operating system.

Optional <*modifiers*> may be specified to control file access in the following ways:

SHARED	The file is **OPEN**ed in a mode that allows other users on the machine to also **OPEN** the same file simultaneously. Under multi-user environments, MegaBasic provides an automatic record locking and unlocking mechanism to ensure file update integrity. Under single-user environments, such transfers involving such files are unbuffered, resulting in much slower response but immediately posting file changes to disk. We will fully discuss the use of **OPEN SHARED** files later on beginning in Chapter 7, Section 2.
INPUT (or READ)	Only permits **READ** or **INPUT** operations to be performed on the open file. A **WRITE** or **PRINT** to such files causes a *Read-Only* error. In multi-user systems, **OPEN INPUT** also allows other users to also have the file open, but no locking operations are supported unless you also open the file **SHARED**.
OUTPUT (or WRITE)	Only permits **WRITE** or **PRINT** operations to be performed on this open file. This is primary useful to trap an **READS** or **INPUTS** from files which you intend to be open for output only, as the file is internally opened in *read/write* mode.
APPEND	Positions the newly opened file to the first byte position beyond its end (instead of at the beginning). A subsequent write operation will then *append* data to the file instead of writing to the beginning of the file. Like the other **OPEN** mode reserved words (i.e., **INPUT, OUTPUT, SHARED,** etc), **APPEND** can be specified in combination with any of the other modes. However, **APPEND** always implies output-only so if later the file is repositioned and read, you must be sure to also specify **INPUT** along with **APPEND** in the **OPEN** statement.

Opening a file for **APPEND** under CP/M systems may be a dangerous thing to do, because there is no way to know exactly where the end of the file really is. CP/M maintains the size of a each file in units of 128 byte blocks and unless you are always writing 128 byte records, opening for **APPEND** Will usually position the file farther than the actual end of the data.

If no modifiers are specified, the file is **OPEN**ed for input and output, and private (exclusive) access under multi-user systems. The **SHOW OPEN** command (Chapter 2, Section 5) shows you the current characteristics of all **OPEN** files.

The same file may be **OPEN**ed under more than one channel number so that several file pointers can be independently used in subsequent file operations. This can be quite useful in situations where sequential access from several different locations occurs simultaneously, such as copying data from one location to another or sorting. MegaBasic controls multiple buffer usage so that at all times any given 512-byte file segment can only be buffered by at most one buffer. This is transparent to the program and prevents any file update problems due to the multiple buffers.

A typical application of multiple file-**OPENS** is *reducing* record sizes in a file, in-place, rather than creating a separate result file. For example the following program removes all the spaces from a text file:

```
10 Rem –– Remove spaces from file F$
20 Open #8,F$; Open #9,F$; Dim LINE$(200)
30 While Input(8); Input #8,LINE$
40 LINE$ = LINE$-“ ”; Print #9,LINE$; Next
50 Close; End
```

Line 20 opens file F$ once as an input file (#8) and once as an output file (#9). Line 30 processes each line in the file until the end of file is reached. Line 40 removes all spaces from the line input and **PRINTS** the resulting line back to the file. The multiple buffers and separate file pointers make this process both easy to program and very fast. This kind of process only works when each resulting line is no longer than the input line.

A *pool* of file buffers is always available which is automatically assigned to files as needed. The buffering system is completely transparent to your programs and never needs any attention other than changing the number of buffers in the pool, using **PARAM**(1O) (Chapter 9, Section 5). This number however need only be changed to increase performance at the expense of memory. As few as 4 buffers or as many as 127 may be assigned and the number assigned is not in any way dependent on the number of files you have open now or intend to open. For example, 4 buffers can service 32 **OPEN** files or 127 buffers can service 1 **OPEN** file.

Files **OPENed** under channel numbers above 2 may be sequentially accessed by **PRINT** or **INPUT** statements and by **READ** and **WRITE** statements. The example program shown earlier does all file access with **PRINT** and **INPUT**. This facility provides efficient sequential access to text-files. See Chapter 7, Section 1 for the details (under **PRINT** and **INPUT**).

All **OPEN** files, their channel numbers, current file positions, and other characteristics are accessible and common to all programs in a multiple package system. In such a system, it may be useful to have a central routine manage the set of channel numbers which is in use and available. The **OPENS**(F) function can be employed to test a channel number for availability.

OPENC *[<modifiers>] #<channel>, <file name>*

OPENC works just like the regular **OPEN** statement except that it automatically creates files that do not exist and erases files that do exist, according to how it was opened:

OPENC	Creates the file if it dos not existalready. If it does exist, OPENC works just like **OPEN**.
OPENC INPUT	Just like **OPEN INPUT**
OPENC OUTPUT	Creates the file if it does not exist, or erases the file if it does exist (i.e., its file size is set to zero).
OPENC APPEND	Creates the file if it does not exist already and positions the file pointer to the end of the file.

OPENC is often much more convenient to use than **OPEN**, but at the expense of exposing data files to possible erasure from unintentional misuse. **OPEN** is more conservative than **OPENC**, but the files it opens must always exist and it performs no extra services.

CLOSE *[<list of channel numbers>]*

Posts all recent changes made to open files (flushes buffers and sets file sizes), then frees the open file channel number for subsequent reuse. The channel numbers of each open file to be closed are listed immediately after the **CLOSE** reserved word. Each channel number is preceded by a Ib-sign (#) and commas separate the channel numbers from one another in the list. All open files are closed when no channel numbers are specified.

For example:

CLOSE	Closes all files currently open. Avoid this form when there is a possibility that you might unintentionally close files opened by other packages.
CLOSE #N+3	Closes the channel number specified by the expression N+3.

MegaBasic does nothing if you specify a channel number that is not currently open. All files are automatically closed when your program **ENDS**, but they will remain open after a STOP statement so that subsequent **CONT**inuation can proceed with the same files available.

FILEPOS (<channel>) = <new file position>

Sets the byte position of an **OPEN** file for subsequent sequential file operations. Random file positioning may also be specified within the data list of a **READ** or **WRITE** statement, but **FILEPOS()** is sometimes more convenient and readable. When the file is being processed only with **PRINT** and **INPUT** statements, this is the only method of re-positioning available. By setting the file position to the file size in bytes (see the **FILESIZE()** function), subsequent **WRITE** or **PRINT** statements will append data to the end of the file, extending its length accordingly.

FILESIZE (<channel number>) = <new file size>

Sets the absolute size of an **OPEN** file to the <*new file size*> specified. This statement changes the amount of physical disk space that is allocated to a file and hence changes the amount of disk space remaining. The <*newfile size*> is specified as a numeric expression (integer of real) that evaluates to the number of bytes in the file (i.e., the file position of the first byte beyond the end of the file).

Setting the file size is not a commonly required operation because the **WRITE** statement automatically extends the file as data is written past the old end of the file. It is especially useful for shortening files, as there is no other way to do it. This operation is possible only if the host operating system supports it (e.g., **MS-DOS** and Xenix).

READ *[<type mode>] #<channel>, <data list>*

Reads data from an open file, specified by the <*channel number*>, into one or more program variables (string, integer or real). The <*data list*> consists of a series of items, separated from one another with commas. Except for file position expressions, all **READ** items must be variables (i.e., not expressions).

File Position (%)

Changes the file position for subsequent **READS** from the file, specified as file byte-position expression preceded with a percent sign (%). When you do not specify this position, data will be sequentially read from the current file position. The current file position becomes the position specified here, and you can re-specify it more than once in the <*data list*>. *You* can also set the file position using the **FILEPOS()** assignment statement.

Variable-length String Variable

Reads a MegaBasic *variable-length* string from the file into the specified string variable. Variable-length strings reside on files with a two or three byte header that indicates how long the string is. This entire length is read into the variable. If the variable is shorter than the string being read, only the left-most portion that fits will be transferred to the variable, discarding the rest. Such strings with headers are written by MegaBasic **WRITE** statements. A *Data Type Error* occurs if a proper string header is not found at the current file location. The target string variable can be indexed or unindexed.

Real Variable

Reads *a floating point value* from the file into the specified *real* variable. No validation of the bytes *read is* performed; whatever bytes are present in the file are read into the real variable. Hence, a real (memory image) representation is read from the file, instead of the less efficient ASCII representation (as some other **BASICS** read). **INPUT** statements should be used for reading *ASCII* numbers from files. The number of bytes read depends on the floating point precision provided by the current version of MegaBasic. Chapter 3, Section 1 discusses the internal representation of floating point real numbers. Chapter 3, Section 7 describes how to read entire arrays from the file in one **READ**.

Integer Variable

Reads the next four bytes from the file into the integer variable specified. These bytes are ordered from the low to high in the file. No data type checking is performed on integer values as they are read from the file (as done for string values described above). Chapter 3, Section 7 describes how to read entire arrays (i.e., vectors) of integer values from the file in one quick operation.

16-bit Values into Numeric Variables (@)

Reads the next two bytes from the file into an integer or real variable after performing the appropriate type conversion. An at-sign (@) must precede each numeric variable in the data list to be read in this manner. The two bytes read are interpreted as an unsigned integer value (ranging from 0 to 65535), which is converted to the proper format for the numeric variable specified (real or integer). The two bytes are ordered low-to-high on the file.

8-bit Values into Numeric Variables (&)

Reads the next single byte from the file into an integer or real variable after performing the appropriate type conversion. An ampersand (&) must precede each numeric variable being read in this manner. The byte read is interpreted as an unsigned integer value (ranging from 0 to 255), which is converted to the proper format for the numeric variable specified (real or integer).

Fixed-length String Variable (&)

Reads zero or more bytes directly from the file into a string variable. No header is interpreted from the file to control this operation. Instead, the number of bytes read is determined by the exact length of the string variable specified, i.e., its string length, not its maximum dimensioned size. You must precede each string variable to be read in this manner with an ampersand (&) to indicate fixed-length string mode.

This method gives you the most control over how many bytes are read, but you are responsible for controlling the length of the transfer by controlling the length of the receiving string variable. In general, fixed-length strings are the best way to read fixed-length records from a file. Additional steps are required to access the individual data fields within such records.

Forming a READ Statement

A **READ** statement data list is composed of the above data items in any combination. The data list may be any length as long as the entire **READ** statement fits within one MegaBasic program line (up to 255 characters). Chapter 3, Section 7 describes an additional way to read data into vector or array variables. **READ** statements are processed from left to right, performing each **READ** or file position task as encountered in the order given in the data list.

The variables read must be simple variables, array elements, string variables or string array elements. Any string variable or array element may be indexed to read data into a substring region of the variable. As data is read from the file, it replaces the prior contents of each receiving variable in exactly the same manner as if the data were stored using an assignment statement to each variable. The following statement will illustrate some of the possibilities:

```
READ #F, A,B,C, %P1 ,X,Y, %P2,Z
```

This reads A, B and C from the current position of the file, X and Y from position Pl, and Z from position P2. Simple numeric variables and individual array elements may be read from the file.

Reading Numbers

The numeric precision assumed when floating point values are read from a file must agree with the precision in effect when the data was written. For example if a file was written using 14-digit precision MegaBasic, it cannot be read by any program being run under 8,10 or 12-digit configurations of MegaBasic. The precision of floating point values read / written to files is, however, independent from the precision used for variables. PARAM(11) may be set to any precision from 6 to 18 digits to control subsequent file transfers (p. [P#,param).

Binary data can also be read into numeric variables by prefacing each numeric variable reference with an ampersand (&) for 8-bit values or an at-sign (@) for 16-bit values. 16-bit values are defined as the next two bytes on the file with the low-order byte first. Both 8-bit and 16-bit values are interpreted as unsigned integer values. Binary file operations bypass all type checking since they read whatever is presented to them. The 8-bit and 16-bit values are converted to floating point format when read from the file into real variables. Integer variables do not require this conversion and hence a much faster transfer results. The following **READ** statement illustrates the possibilities:

```
READ #F,X,A(I,J),8Y,@B(K)
```

where numeric values are read into X and A(I,J); a byte-value is read into Y and a 16-bit word-value is read into B(K). Reading vast quantities of bytes with this method is not recommended because each byte read is individually read and dispatched to each destination variable one at a time. Also, you should avoid using real variables for receiving 8 and 16 bit values because of the considerable effort that is required to convert each value to floating point representation, a process performed internally for each variable read. Large quantities of binary information should be read as fixed-length strings into string variables, as described shortly.

Reading Strings

String variables can be read as a whole or as indexed sub-strings. String data is written in a special compact format: the amount of storage taken equals the length of the string plus two for up to 255 characters, or plus three for over 255 characters. The short sequence of bytes in front of such strings is called the string header, which tells MegaBasic that a string of some specified length resides on the file. This header makes it possible to sequentially read variable length strings, one after another from the file, without having to explicitly specify and control how much data to read on each transfer (because the length is embedded in the string itself).

If the string on the file is larger than the variable it is read into, the characters that don't fit are lost just like string assignment statements). Regardless of the string variable capacity, the file pointer is always set properly to the next item in the file after reading any string.

In some applications, string headers are not suitable because you are reading pure binary information from the file that was not written as variable-length strings. An example of this is a file written by a foreign system which your program is processing by interpretation. To read pure sequences of bytes from files, bypassing all type and header controls, precede each string variable name with an ampersand (&). The number of bytes read is controlled by the current (before the **READ**) length of the string. 16-bit binary **READS** are not possible with string variables. The following example shows how this is done:

```
10 DIM A$(100), B$(167); Rem––Set your string sizes
20 Read &A$, &B$;        Rem –– Read 100 bytes into AS, 167 into B$
30 Read &A$(1,N), &B$(1,M); Rem––Read N bytes into A$, M into B$
```

It is very important to realize that the number of bytes read using ampersand string variables is equal to the length of the string variable specified in the **READ** list. This length is not necessarily the same as the DIMensioned size of that string variable. For example if we set A$="" prior to the **READ** in line 20 above, zero bytes will be read into A$ when it is read. Furthermore, this kind of **READ** never affects the length of any string variable, as measured by the **LEN** function (e.g., **LEN(A$)**). A length of a string variable can be set to any length up to its dimensioned size with a statement like: **LEN(A$)-N**, as described in Chapter 5, Section 1.

Random-Access READs

As each data item is read from the file, the file position is advanced by the number of bytes read, so that the file position is always aligned to the next data item. When randomly accessing a data file, you must specify file positions which always refer to the first byte of multi-byte data items (such as strings, 16-bit values, and floating point values. To do this you must know the number of bytes required for each data item. String and binary data types are covered above. The length of floating point values is always the same for a given MegaBasic precision: **PRECISION**/2+1. Thus the standard 8-digit precision requires 5 bytes (8/2+1 = 5). If you ever access a data item somewhere past its first byte, a Data *Type Error* will usually occur to inform you of the problem. However binary data items have no identifying characteristics to permit such error detection, so exercise great care when processing random binary files.

Numeric Representation Control

The <*type mode*> *is* an optional modifier you can place in any **READ** (or **WRITE**) statement to control the numeric type (real or integer) of numeric values read, regardless of the numeric type of the variable receiving the number. You specify this modifier as the reserved word **REAL** or **INTEGER** typed just after the word **READ** and before the #channel number expression. This modifier has no effect upon string **READS** and **WRITES** not even in the case where the strings happen to contain numeric structure fields, as such strings are treated as ordinary strings.

The **INTEGER** modifier causes all numbers to be read as integers during the modified **READ** statement, even if real variables appear in the data list. MegaBasic automatically converts these integer values read from the file into real form when the receiving variable is real, and no conversion is applied when the variable is integer.

The **REAL** modifier causes all numbers to be read as real values (**IEEE** binary or **BCD** floating point) during the modified **READ** statement, even if integer variables appear in the data list. MegaBasic automatically converts these real values as they are read from the file into integer form when the receiving variable is integer, and no conversion is applied when the variable is real.

In most applications, numbers are written on files in predominantly one type or the other, and the <*type mode*> allows you to **READ** a series of numeric values of one type into a series of variables of both types (in any mixture). This feature can be useful for ensuring the proper interpretation of stored numbers without having to be aware of the numeric type of the variables being read into. Numbers read into variables preceded by ampersands (&) or at-signs (@) are unaffected by the **REAL** and **INTEGER** type modifiers. Converting numbers between real and integer representations can degrade program performance and may involve some precision loss in the process. Chapter 3 describes both numeric types and important additional information about conversions.

One application for the **REAL** <*type mode*> occurs when a program designed with only real variables is converted to take advantage of integer variables and expressions. Files processed by such programs may be reading (and writing) numbers which, by design, would always be real numbers. If the data lists of **READ** (and **WRITE**) statements become integer data lists, the real file transfers will suddenly become integer file transfers, and erroneous results would surely follow. You can solve this problem by simply inserting the **REAL** modifier immediately after each **READ** and **WRITE** reserved word throughout such converted programs to force **REAL** number transfers for all transfers.

WRITE *[<type mode>] #<channel>, <data list> [,NOMARK]*

Writes a list of one of more data items to the open file specified by the <*channel*>. WRITE works just like the **READ** statement except for the direction of data transfer. The data listed need not be confined to just variables. Any data being written may be specified as a general expression whose computational result is written to the file. As in the **READ** statement, the data list of a **WRITE** consists of strings, numbers and file position expressions, as summarized below:

File Position (%)

Specifies the starting byte position in the file to which the next data item will be written. This is specified with a percent sign (%) followed by a numeric expression that evaluates to the desired byte position. When you do not specify this position, data will be written sequentially to the current file position. The current file position becomes the position specified here, which can be re-specified more than once in the <*data list*>. *You* can also set the file position using the **FILEPOS()** assignment statement.

Variable-length Strings

Writes the string specified as a variable-length string to the file at the current file position. Such strings are written to files with a two or three byte header that specifies how long the string is, so that the same length can be read back at a later time. The header also helps MegaBasic verify that a string is actually on the file when a **READ** request attempts to read it. Variable-length strings are useful for variable-length record structures.

Real Values

Writes the specified floating point real value to the file at the current file position. The exact memory image of the (**IEEE** binary or **BCD**) floating point value is written to the file, rather than a printable ASCII numeric representation (as written by some other **BASICS**). The number of bytes written is dependent on the floating point precision provided by the current version of MegaBasic (as described in Chapter 3, Section 1). The precision of floating point values read / written to files is, however, independent from the precision used for variables. PARAM(11) may be set to any precision from 6 to 18 digits to control subsequent file transfers (Chapter 9, Section 5). **PRINT** statements should be used for writing ASCII numeric representation to files.

Numbers should be written to files in this manner when speed is important and the numbers being written contain decimals or can span a very large numeric range. This is the most general purpose numeric file format. A number is written in this representation whenever the number evaluates to a real number. See Chapter 7, Section 3 for complete details on the differences between real and integer numbers and how to specify them. Chapter 3, Section 7 describes how to write partial or entire arrays to files in one quick step (i.e., as vectors).

Integer Values

Writes the specified integer value to the file at the current file position as a sequence of four bytes. These bytes are ordered from the low to high in ascending locations on the file. You should consider this numeric file format when speed and wide range are important and the values being written are always integer, especially those which are manipulated within programs in integer variables. This representation is particularly well suited for storing file position pointers used in complex linked file structures. A number is written in this representation whenever the number evaluates to a integer number. See Section 3 of this chapter for complete details on the differences between real and integer numbers and how to specify them. As with real numbers, integer arrays can be written to files using vector write operations, described in Chapter 3, Section 7.

16-bit Word Values (@)

Writes the specified value, which must be within the range from 0 to 65535, as a two-byte sequence to the current file position. You must indicate numbers to be written in this manner by preceding each one with an at-sign (@) to distinguish them from those written in real or integer format (described above). The two bytes written are written in low-byte high-byte order on the file. Real values written in this way are automatically converted to binary integer form before the transfer takes place. You should consider this numeric format when compactness is an important consideration and the values to be written always lie within the 16-bit unsigned integer range.

8-bit Byte Values (&)

Writes the specified value, which must be within the range from 0 to 255, as one byte to the current file position. You must indicate numbers to be written in this manner by preceding each one with an ampersand (&), distinguishing them from those written in real or integer format (described above). Real values written in this way are automatically converted to binary integer form before the transfer takes place (a time consuming process best avoided). You should consider this numeric format when compactness is the overriding consideration in your application and the values to be written always lie within the 8-bit unsigned integer range.

Fixed-length Strings (&)

Writes the string specified directly on the file at the current file position. No header describing the string is written to the file ahead of the string. The string is written exactly as specified, no more, no less, and programs that read it back are expected to determine for themselves how to many characters to read. You must indicate strings to be written in this manner by preceding each one with an ampersand (&), distinguishing them from those to be written as regular string data with headers. You should consider writing strings in this way when headers are undesirable, or for fixed-length records, or when you want to control all aspects of your data transfers at the expense of some additional program complexity.

Forming WRITE Statements

A WRITE statement data list is composed of the above data items in any combination. The data list may be any length as long as the entire **WRITE** statement fits within one MegaBasic program line (up to 255 characters). Chapter 3, Section 7 describes an additional way to write data from numeric vector or array variables. WRITE statements are processed from left to right, performing each **WRITE** or file position task as encountered in the order given in the data list. A typical **WRITE** statement looks something like this:

```
Write #F, R+S,Log(X), %P1,X,Y, %P2,ZIY-54
```

This statement writes R+S and LOG(X) at the current file position, X and Y at position Pl, and Z / Y-54 at position P2. Without any file position specifications (%), all data is written sequentially on the file. Writing past the physical end-of-file causes the file to be extended automatically.

Writing Numbers

It is of the utmost importance for you to understand that numbers are written to the file in different ways depending on whether the number evaluated in the <*data list*> as a real number or as an integer number. **Integers and reals** are written using a different number of file byte positions and their structures are interpreted in totally different and incompatible ways. Programs that expect a certain numeric representation on a file where a different one resides will produce erroneous and unpredictable results. You should be familiar with all of the material presented in Section 3 of this chapter, in order to properly express numeric values and computations so that they evaluate to the expected numeric type.

However, you can specify an optional <*type mode*> for specific **WRITE** (and **READ**) statements to force numbers being written to a specified type, regardless of how they are specified. Either of the reserved words **REAL** or **INTEGER** may be inserted between the **WRITE** (or **READ**) reserved word of the statement and the <*channel*>, to cause any particular file transfer to only write the numeric file representation corresponding to the type specified. Numeric type control only affects numbers and has no effect on strings, even if the string contains numeric structure fields.

If a number appears in the <*data list*> of the wrong type, MegaBasic automatically converts it to the <*type mode*> specified. Although too many such conversions can degrade the performance of your program, your program will transfer numbers in a predictable and expected manner. You should be aware however that these conversions have certain limitations on numeric range and precision that are also described in Chapter 3. The following two example **WRITE** statements illustrate an integer-only **WRITE** and a real-only **WRITE** to the channel number contained in variable F:

```
Write integer #F, X,Y, Z
Write real #F, X,Y, Z
```

Binary data can be written by prefacing each numeric expression with an ampersand (&) for 8-bit values or an at-sign (@) for 16-bit values. Floating point values from 0 to 65535 are converted to binary format before the **WRITE** takes place. The optional <*type mode*> has no effect on any binary format. For example:

```
Write real #F,X,A(I,J),&Y,@B(K)
```

In this **WRITE** statement, X and A(i,j) are written to the file as floating point values, Y is converted to an 8-bit value and then written, and B(E9 is converted to a 16-bit value and then written. 8-bit and 16-bit values consume only 1 and 2 bytes of file space, respectively.

Writing Strings

String expressions may be included in the data expression list and are written in a manner corresponding to the **READing** of strings. Preceding string expressions with an ampersand (&) causes the string to be written as a sequence of fixed-length bytes for the length of the string. Without the ampersand (&), strings are written with a 2 or 3 byte header in front of the string for identification purposes; ampersand (&) strings are written as pure fixed length byte sequences without any header. The following example illustrates the various ways a string can be written:

```
Write #F, A$,B$(1,J),"literal"+Q$, &C$,&D$(K,L),&Chr$(0)*15
```

This statement writes A$,B$(i,j)and"literal"+Q$tofile#Fasvariable-length strings, followed by fixed-length strings C$, D$(k,1) and CHR$(O)*15 (without headers), the later of which generates a sequence of 15 binary zeros, a topic described in Chapter 4, Section 4.

Since the elements of a **WRITE** statement may be general expressions, each data item listed in the data list is evaluated internally and this internal result is then written to the file. String data can potentially consume all the internal work space set aside for this purpose and hence great care must be exercised when very long *computed* strings are being written.

In the case where the string expression consists solely of a string variable (indexed or not), the contents of the variable are transferred directly to the file, rather than being evaluated internally as a general expression. Hence no internal memory is required for this common case. A side benefit of this implementation is that large transfers out of string variables proceed twice as fast as transfers of string expression results of comparable length.

End-of-File Marks

If file endmark generation is enabled (see the **NOMARK** statement below), MegaBasic writes an additional single byte file endmark at the prevailing file position (without advancing the file pointer) after executing each **WRITE** statement. While useful for purely sequential file usage, this often proves unsatisfactory for binary or random access operations. To prevent the generation of the end-of-file mark, you may finish **WRITE** statement with the **NOMARK** keyword, for example:

```
Write #F, A$, X, Y, Nomark
```

This same keyword may also be used as a program statement to provide global control over the generation of end-marks for subsequent write operations (see the next statement) . The endmark code generated may be redefined using **PARAM**(9), described in Chapter 9, Section 5.

NOMARK *<logical exprn>*

Normally, MegaBasic does not write endmarks during write operations. The **NOMARK** statement lets you enable or disable endmark generation at any time. A non-zero expression (logical *True)* causes file mark suppression, and a zero (logical *False)* expression brings it back again. When endmarks are enabled, the endmark code (usually an ASCII 26 code), is written to the file each time a **WRITE** statement is executed. It is written to the file at the file byte immediately following the last item written by the **WRITE** statement (without advancing the file position).

Endmarks usually *get in the way* in applications involving binary or random file operations. In other applications, usually text or other sequential processing, endmarks can be useful. CP / M-86 extends files in blocks of 128 bytes at a time. Endmarks can be useful under that operating system (and others like TurboDOS) to mark the *true* end-of-file within the last file block.

If the keyword **NOMARK** is appended to the output list of a **WRITE** statement, the endmark for that statement is suppressed, regardless of the current **NOMARK** enable / disable state. Therefore, the **NOMARK** statement affects the action of only those **WRITE** statements which omit the **NOMARK** keyword. The endmark code normally used is an ASCII 26 code, but this can be redefined using **PARAM**(9) (Chapter 9, Section 5) or the **CONFIG** utility program (Appendix C, Section 2). **NOMARK also** controls whether or not the end-mark is used to detect the end of file during **READS** and **INPUTS**.

WRITE *[#<channel number>]*

Flushes all file buffers internally associated with the open file specified, or flushes all open files if no channel number is specified. This statement has the same effect as closing the file(s), re-**OPENing** them, and setting the file position of each back to the same position it was when closed. This statement is useful in situations where, at certain points during program execution, it is desirable to ensure that the physical files on the disk are totally up-to-date. Typical applications include data base programs and multi-user environments. Files which are **OPEN SHARED** are unlocked by this operation.

READ *[#<channel number>]*

Flushes all file buffers for all open files or for the one specified, just like the **WRITE** statement described above. In addition however, this statement empties the contents of all the affected file buffers so that subsequent file operations are *forced* to perform physical disk transfers. Most programs will not need (or desire) to use this statement, which greatly slows down subsequent file operations. It is provided specifically for multi-user and network environments where files may be serving several processes simultaneously. It is not possible to update multi-access files properly unless the users of such files always deal directly with the file itself, rather than through private buffers in memory. This statement simply re-starts the buffering process from scratch, each time it is invoked.

Shared OPEN Files

When your program **OPENS** a file under multi-user and network versions of MegaBasic no other process is permitted to open that file with write-privileges until it is **CLOSED**. Conversely, if the desired file is in use by another process when your program attempts to **OPEN** it, a *File Busy Error* occurs, which can be trapped with ERRSET as a type 26 error. Once the file has been successfully **OPEN**ed, your program has exclusive access to that file. NOTE: MegaBasic exclusive **OPENS** *do allow* other processes to open the same file for *read-only* access. Conversely, files open for *read-only* access can also be opened by other processes for *read write* access.

Occasionally, especially in large data base applications, you may wish to **OPEN** a file without excluding other processes from **OPENing** it and without being prevented by another's access in progress. This is called a non-exclusive or **SHARED** open file, which is specified by an **OPEN** statement of the form:

```
OPEN SHARED #<channel number>,<file name>
```

Any file, not already exclusively **OPENed** (non-**SHARED**), may be **OPENed** for **SHARED** access. Such files can be tricky to deal with, however, because if several processes are accessing and modifying the same area of the file simultaneously, the outcome can be highly unpredictable (depending on the order in which the different processes access / modify the file contents). Depending on your system, you may have to set Param(22) for correct operation (Chapter 9, Section 5).

The general approach that you must use in shared file access applications is to perform one *transaction* at a time. A transaction in this context is the set of all operations on shared files that must all be performed and completed successfully to bring about the desired outcome. For example, inserting a new record into a data base is one transaction that may involve updating several index files and one or more master files. Such operations will fail if other users can modify any of these related file structures in the middle of this transaction. Furthermore, external access involving any data that is in the process of changing will not be valid either. Hence, the following outline summarizes the correct way to implement such a transaction:

- Identify and lock all potentially writeable components required in the desired transaction. Do not proceed until all components are locked or you abort the transaction.
- Now that you have exclusive access to all required file regions, perform the necessary accesses and updates that are needed.
- Unlock every component locked in the first step.

MegaBasic makes use of the record locking capabilities of the multi-user system to give your program temporary exclusive access into critical areas of the **SHARED** file (regions where potential modifications could disrupt the intended sequence in progress). Although this locking mechanism has been designed to be as automatic and transparent as possible, there are inherent difficulties with the very concept of shared files which require the MegaBasic programmer to fully understand the actions invoked by **READ** (or file **INPUT**) and **WRITE** (or file **PRINT**) statements.

Explicit File Locking

The **LOCK** statement lets you explicitly lock an open file region and **UNLOCK** lets you unlock it, both of which have the same syntax:

```
LOCK #<file number>, <lock specification>
UNLOCK #<file number>, <lock specification>
```

where the <*lock specification*> can be one of the following:

<start>	Specifies a 1-byte lock at the specified file location.
<start>: <length>	Specifies a lock at the specified file location over the <length> specified.
<start> TO <end>	Specifies a lock at the specified file location up to the ending position specified.

LOCK and **UNLOCK** give you complete low-level control over the locking used on **SHARED** files. They do nothing if applied to files not opened in **SHARED** mode. However, you are responsible for unlocking everything you lock in *exactly* the same manner it was locked and for remembering to do so before closing the file. **LOCK** and **UNLOCK** affect neither the current file size nor its position for subsequent **READS** or **WRITES**. Retriable *Suspended File Access* errors are generated if the file is already locked by another process.

The automatic locking performed by MegaBasic on **SHARED** files is still active and can be used along with the **LOCK** and **UNLOCK** statements. However, since these direct locking statements provide more control over locking, **PARAM**(26) can be set to 1 to disable the automatic locking of MegaBasic (or to zero to re-enable it again). **PARAM**(26) is local to each MegaBasic package. Setting **PARAM**(22) to –1 will disable automatic locking over all packages in the application.

Automatic Locking Under MegaBasic

Files under MegaBasic are *stream-oriented*, rather the *record-oriented*. This means that all files are viewed as a continuous sequence of bytes, over which the programmer may impose logical records using file position addressing techniques. Thus a string or number on a file is read or written as a short sequence of bytes at some specific byte location within the file. This approach to file operations provides the greatest possible flexibility, but it leaves the notion of *records up* to the individual programmer to define. For example, records of 100 bytes can be accessed by record number R using the following **READ** statement:

```
READ #F, %R*100, A$,B$,X,Y,Z
```

Notice that we position the file at the byte location derived by multiplying the record number by the record length. The record itself, in this case, consists of two strings and three numbers.

In multiuser and local area network applications, record locking is generally provided by letting an application inform the operating system that one or more regions of a file will be temporarily *locked* to prevent other users from accessing them while they are being changed. After completing the transaction, the user *unlocks* the locked regions and continues on.

In a record-oriented file model, records can be locked and unlocked as they are accessed. But a different approach must be used in a stream-oriented file model because the *records* are conceptual, rather than physical. Since the operating system typically limits number of individual file locks per process (usually less than 64), locking each byte is not feasible. Instead, MegaBasic defines records, for record locking purposes, in the following way:

- The beginning of a *record* is defined as the first file byte accessed by a **READ** or **WRITE** statement or the first byte accessed **after a** change in file position.
- The end of a record is defined as the last byte accessed by the **READ** or **WRITE** statement before the end of the **READ** or **WRITE** statement, or the next file position change, which ever occurs first.

Thus each **READ** and **WRITE** statement can access one or more *records*, delimited by the statement boundaries and / or file position changes. For example, the following **READ** statement accesses three *records:*

```
READ #F, A$, %123,X,Y,B$, %9999,C$,Z
```

Notice that several variables may be accessed in one record and that you could read the beginning of a record without reading the rest of it. It is important to realize that this notion of records is defined for the purpose of record locking and it may not correspond to the logical records that a program uses when accessing the file.

When your file is **OPEN SHARED**, all **READS** and **WRITES** (**INPUTS** and **PRINTS** too) implicitly lock and unlock the file blocks in a predetermined way. Your programs are never concerned with the actual lock / unlock operations themselves. Your program is strictly concerned only with its **READ** and **WRITE** statements. Internally, MegaBasic performs **READS** and **WRITES** in the following sequence of steps:

Shared Read Sequence

- Unlock all locks currently active throughout this file.
- Reposition the file as needed, then lock the first byte of the current record.
- Read the actual record directly from the file. No buffers are used on any file operations of file opened in **SHARED** mode. If there are more records to read, go back to step (2).

In other words, a **READ** statement first unlocks all areas currently locked in the file, then locks each *record* and reads it into the program variables. Lack of buffered support for **SHARED** transfers greatly slows down the transfer but this is essential for correct operation in a shared access system. The **WRITE** sequence is the converse of the **READ** sequence:

Shared Write Sequence

- Reposition the file as needed then lock the first byte of the current record.
- Write the record directly to the file, i.e., without any file buffering. If there are more records to write, go back to step (1).
- On completion of the write statement, unlock all records of this file that are currently locked.

In other words, a **WRITE** statement locks each record before writing it, and writes each record directly to the file without buffering. On completion, all records currently locked are unlocked, including both the newly locked records and any records already locked before the write operation began.

Notice that **WRITE** is the converse of **READ** in all respects. In most programs most of the time, the scheme for **READS** and **WRITES** as defined above will provide all the locking / unlocking features necessary. For example reading some values, using them in a computation, then modifying and writing them back to the file will always operate correctly, even if other processes are trying to do the same thing. This policy also tends to minimize the number of file locks active at any given instant.

At first glance, it may appear that locking only one byte of a record would not be sufficient for reliable operation. However, shared access is an inherently cooperative process that would not work without agreement by all competing processes. As such, locking the first byte of a record is equivalent to locking the entire record as long as every file access abides by one very simple rule: always access records from the beginning of the record, i.e., do not access the middle of a record without accessing the beginning. This is a very reasonable rule that is easy and natural for any process to observe. Such processes already have to cooperatively access files through the locking primitives provided by the operating system, so one additional minor rule is in no way unreasonable.

Multiple File Locks

In some situations, a transaction involves several regions of one or more files, all of which have to be locked during the transaction process. For example, deleting records from linked data structures stored on a file may requires numerous **READS** and **WRITES** to perform one logical modification to the file, requiring a continuous lock on all file regions involved until the operation is complete. To do this, the file transfer statement is augmented with the keyword *LOCK* to indicate that locked records are to remain locked. This disables the unlocking phase of the **READ** and **WRITE** statements that was described earlier. Such statements will then appear as follows:

```
READ LOCK #<channel number>, <transfer list>
WRITE LOCK #<channel number>, <transfer list>
INPUT LOCK #<channel number>, <transfer list>
PRINT LOCK #<channel number>, <transfer list>
```

PRINT and **INPUT** statements can also transfer data to and from files and as such, they should be thought of as **READS** (**INPUT**) and **WRITES** (**PRINT**).

We should emphasize that this mode of operation is the exception rather than the rule and overusing it can lead to poor performance for competing processes or exceed the maximum number of locked regions permitted by the operating system, resulting in a *Too Many File Locks Error*. If you require many parts of a file to be locked simultaneously, you should consider **OPEN**ing the file in exclusive (un**SHARED**) mode, instead of **SHARED** mode.

Always minimize the total number of locked records at any given instant. This is done by following the policy of locking what you need to, use the locked data immediately, then perform all necessary updates (**WRITES**) and leave the file in a completely unlocked state – all as one action to be done without interruption. MegaBasic itself can only maintain a total of 64 locks at any given time over all **OPEN** files collectively. The operating system may support more or less than this number, so your programs must be designed to limit the use of this extremely scarce resource to avoid premature program termination (by errors).

Some operating systems, notably Concurrent CP/M and TurboDOS 86, cannot lock files down to the byte level. Instead, they only support locking of file *blocks*, typically 128 bytes each. Therefore when MegaBasic locks the first byte of a record under these systems, it is actually locking the appropriate block in the file that contains the desired byte. This can sometimes cause unexpected delays due to the locking of nearby regions unrelated to the transaction at hand. This cannot be avoided in such systems.

Retry Control for Blocked Resources

When MegaBasic attempts to lock a record, in preparing to read or write that record, a *Suspended File Access Error will* occur if that record is already locked by another process. You should consider this error to be a normal, expected event in applications that access shared files. However, setting up an **ERRSET** trap for every file access operation in the program can get pretty tedious (and bulky). Therefore MegaBasic provides a special mechanism that lets you provide your own automatic response in these situations.

The **RETRY** statement (Chapter 6, Section 4) defines a procedure to be invoked when resource access is temporarily blocked by other processes. This include the system printer, exclusively **OPEN**ed files (un**SHARED**), locked records, locked disk drives, etc. This error recovery mechanism is supported in addition to the **ERRSET** recovery mechanism already provided for all trappable errors. A **WAIT** statement (Chapter 6, Section 4) can be used for timed delays that do not consume **CPU** cycles can be invoked to control the retry process.

Section 3: System Interface Statements

These statements provide access to system memory and hardware ports 0 to 65535 and permit control of various MegaBasic system parameters. See Chapter 9, Section 5 for the discussion of additional functions **FREE()**, **EXAM()**, **INP()** and variable addressing [ ]. This Section covers the statements summarized below:

SEG	Defines the default physical memory segment to use when the segment portion of a segment:offset address is omitted.
FILL	Stores data directly into physical memory locations.
EXAM	Reads data directly from physical memory into program variables.
OUT	Sends 8-bit values through physical machine ports.
CALL	Invokes a machine code subroutine using a **FAR CALL**. Machine registers can be set before the call and retrieved after the call. Your own machine subroutines can be accessed as a MegaBasic package using another method described in Chapter 10, Section 5.
CALL	Invokes a machine code subroutine using a software **INT** number Machine registers can be set before the call and retrieved after the call.
CALL#	Defines a MegaBasic procedure that is access externally through a software **INT**errupt.
DOS "cmd"	Executes an operating system shell command.
PARAM	Provides access to many MegaBasic internal control variables, some of which can be altered, all can be read.

Memory access should only be used by qualified programmers and even then avoided whenever possible. It is very easy to corrupt the machine code of MegaBasic to produce unpredictable and even disastrous results. Furthermore, programs which rely upon such techniques will be highly machine dependent and potentially very difficult to move to other machines or operating systems. These operations are intended for limited use by systems programmers to perform actions which would otherwise be impossible.

To accommodate the segmented addresses of the 80x86 **CPU**s, memory addresses required by the **FILL, EXAM** or **CALL** functions and statements may be specified two different ways:

<segment address>: <offset address>
or
<offset address>

The first form specifies both address components, each of which may be given with a arithmetic expression, and represents a complete absolute memory location. The second form specifies only the offset portion of the complete address and the segment portion is the default segment, defined by an earlier SEG statement (described below). Keep this in mind whenever using **FILL, EXAM** or **CALL.**

Many of the parameters and results of these system interface statements are in strictly integer form. In MegaBasic you can generally specify numbers in either real or integer form, but real numbers are converted to integer when the context in which they are used demands it. Better performance results if you always specify numbers in the same numeric type as used by the application.

SEG [<variable name>]

MegaBasic variables do not reside at fixed memory locations because their segments are physically relocated from time to time during program execution to efficiently allocate large memory blocks. The **SEG** statement sets the default segment address as used by the **FILL, EXAM** and **CALL** statements. This statement sets the default segment address to that of the specified variable (even if it changes), or to the standard control segment if omitted.

This default is local within functions, procedures and **GOSUBS**, and cannot therefore be changed by invoking a sub-program of any kind. The default segment is applied whenever a memory address does not contain an absolute segment override and will always be correct no matter how much internal reorganization occurs.

FILL *<starting address>, <data list>*

Stores a list of data values directly into sequential memory locations. The data list is identical to that of a **WRITE**# statement and the data is stored into memory in the identical format, summarized as follows:

- Floating point (real) values are stored in the prevailing floating point format native to the executing MegaBasic (**BCD** or **IEEE** binary floating point format). Control over precision using **PARAM**(11) is not supported: the normal internal precision is always used.
- Integer values are stored as a **sequence of four bytes which repre**sent the 32-bit twos-complement integer value used by MegaBasic. These bytes are ordered from low to high in ascending memory locations.
- Numbers specified in either real or integer form can be **FILL**ed as 8-bit or 16-bit unsigned binary integers by preceding each such value in the data list by an ampersand (&) or an at-sign (@), which converts and stores the values in 8-bit or 16-bit (low-high order) unsigned binary integer format, respectively. An *Out Of Bounds Error will* occur if any numbers so specified lie outside the range of 8-bit and 16-bit values.
- String values are stored in packed string format with headers unless preceded by an ampersand (&), which stores the string in binary format without string headers. See the **READ**# (Section 2 of this Chapter) and **WRITE**# (Section 2 of this Chapter) statements for further information.

EXAM *<starting address>, <variable list>*

Loads string or numeric variables from absolute sequential memory locations. **EXAM** loads variables from memory the way the **FILL** statement stores values into memory. Its variable list is identical to that of the **READ**# statement (Section 2 of this Chapter). Refer to the FILL, **READ**# and **WRITE**# statements for further information.

OUT *<port number>, <byte value>*

Sends an 8-bit value (0..255) out through the hardware port specified. No status interrogation is performed and the transfer takes place immediately. Any 8086 port number from 0 to 65535 is permitted. The **OUT** statement will accept either numeric or string data for output through **CPU** ports. For example: **OUT P,C$** will output the first character in string variable C$. Any general string (or numeric) expression may be specified, however only the first character of the string is **OUT**put. If a null string is specified, an undefined value is **OUT**put.

CALL #<int> [,<register$> [,<result register$>]]

Invokes any of the 8086 software interrupts numbered from 0 to 255. An actual software interrupt instruction implements the call, rather than simulating it. Calling an uninitialized interrupt number will likely *crash or halt* the system. The machine code interrupt subroutine being called *must* termInate with an *8086 IRET instruction*. This statement permits machine register access on both the call (the first register string) and the return (the result register string variable). Register values are specified concatenated together as a string of characters, positionally defined in the string as follows:

Position	*Register*	
1	AX	AH
2		AL
3	BX	BH
4		BL
5	CX	CH
6		CL
7	DX	DH
8		DL
9	SI	
11	DI	
13	BP	
15	ES	
17	DS	
19	*CPU Flags*	

Use **BIT()**, **ASC()**, **CHR$()**, **FILL** and **EXAM** to pack / unpack your desired values to / from the string arguments. **CALL** is 80x86 **CPU** dependent and other MegaBasic versions using different microprocessors may use different but similar conventions. To send and receive all the registers, the string arguments specified must be the full 20 byte length (i.e., ten 16-bit registers). Shorter strings access fewer registers, e.g., a length of 5 bytes would access registers AX, BX and CH.

The **CPU** register string includes the *Flags* register as bytes 19 and 20 of the register string. Because of the critical nature of the **FLAGS** register, you cannot set any of the flag bits and attempts to so will be ignored (without any reported error). Your program can only examine the **FLAGS** in a returned register string. Usually only the ***CARRY*** *flag* will be of interest and it is returned in **BIT(REG$(19),7)**.

Although there can be good reasons for doing so, passing absolute addresses of MegaBasic variables to external routines for access and / or modification can be *very risky*. This is because MegaBasic variables are moved from time to time to allow efficient management of the available memory. The very act of using a CALL statement can cause a shift in memory addresses. The following steps can be used to minimize this difficulty.

- Set the register string to chr$(0)*20, which forces a memory shift if one would have occurred.
- Setup the register string with your absolute addresses, taking care to use no user-defined functions or complex expressions that might cause another memory shift in the process.
- Make the **CALL**

Interrupts which have been set up by the **SERVICE** statement (described later) cannot be accessed by **CALL**# statements using the same MegaBasic program that set them up. However, they can be invoked from programs running under other copies of MegaBasic in the same machine. An *Interrupt Service Error* results if this rule is violated.

CALL *<seg>:<offset> [,<register$> [,<result reg$>]]*

Executes an *8086 FAR CALL* to the subroutine at the memory location specified by the numeric address expression *<seg>:<offset>*. The **CPU** register contents can be communicated to and from the routine using the same conventions as specified for the *interrupt* **CALL**# *statement* described above. The machine code subroutine being called must terminate with an 8086 **FAR RET** *instruction.*

DOS *[<command string expression>]*

DOS statements without any arguments exit MegaBasic, or execute operating system commands specified as a *string expression* and then returns to MegaBasic. Thus you never need to return to the operating system to invoke some operating system service. From MegaBasic, you can **COPY** or **TYPE** files, display **DOS** directories, run batch files, execute programs written in C, **COBOL**, **FORTRAN**, assembler, **PASCAL**, or any other language, so long as the program can be run from the operating system *command level.*

Since the command string is a string expression, you have to surround it with quotes, but you can also specify it using an arbitrary string expression. Omitting the command string altogether will exit MegaBasic and return you to the operating system command shell (however the **END** statement is preferred).

You specify the operating system commands exactly as if you were typing them at the operating system *command level,* for example:

DOS "copy C:*.*B:*"	Copies all files from drive C: to drive B:.
DOS "type "+T$	Types the text of the file named in string variable T$ on the console screen.
DOS ""	Enters the *MS-DOS command shell* while preserving the current program state within MegaBasic. At this point you can enter DOS commands for as long as you want to. Afterward to resume where you left off, just type the DOS EXIT command to get back into MegaBasic.

Consult your **MS-DOS** operating system manual for full information about the available **DOS** commands and how to specify them. We describe several important ways to use the **DOS** statement below:

- Your programs can request **MS-DOS** commands from the user with an **INPUT** statement for immediate execution by the **DOS** statement. Hence, your program can always stay in control without giving up any capabilities.
- If the command string is a null string, MegaBasic invokes the command processor for command execution from the console, which allows the user to enter as many **MS-DOS** commands as desired. To return to MegaBasic, the user must type the **MS-DOS** command **EXIT**, which exits the command processor and returns **back** to MegaBasic.

- Your program can build a **MS-DOS** *batch* file and then execute it with a **DOS** command. This is done by simply **PRINT**ing them to an **OPEN** file whose name has the extension *.BAT* (e.g., *BATCHFIL.bat)*. You execute a batch file by giving its name as an **MS-DOS** command (e.g., DOS "**BATCHFIL**"). Upon completion of the last command in the file, MegaBasic regains control and your program continues on its way. See your **MS-DOS** manual for further information about batch files and batch commands.

- Sometimes you may want to redirect the output of an executing batch file. Given a file or device name in string variable OUT$ and a batch file name in variable **BATCH$**, the following MegaBasic statement will redirect all console messages to the message destination specified:

  ```
  DOS "COMMAND > " + OUT$ + "IC " + BATCH$
  ```

  Be sure that your batch file ends with an **EXIT** command to return to MegaBasic. Other wise, it enters the shell command level and waits for a command while screen output is redirected away from the screen.

DOS shell commands come in two flavors: internal and external. *Internal* commands are those *built into the command shell,* while *external* commands are those in *separate files,* i.e., those with **.EXE**, **.COM** or **.BAT** file extensions. External commands always return a termination or *exit code* which your program can access from **PARAM**(19) immediately after the **DOS** statement. See the **END** statement (Chapter 6, Section 1) for more information on exit codes.

There is one important restriction that you must be aware of in using the **DOS** statement to execute **DOS** system commands: **never execute** a program which stays resident in memory after it terminates. An example of this type of program is the **PRINT** utility included in the standard set of **MS-DOS** utilities. Such programs will likely appear to operate correctly for a while, but later on after MegaBasic regains control and / or terminates, the system will probably crash or lockup with a memory allocation error at some point.

You can use most resident programs without any problems by making them resident *before you bring up MegaBasic.* For example, you can install the **DOS PRINT** utility before you get into MegaBasic, then later invoke **PRINT** from MegaBasic through a **DOS** command. This is necessary as a result of the memory allocation mechanism used within the **DOS** operating system; it is not a bug in MegaBasic.

The **DOS** statement relies on the **MS-DOS** command processor residing on a disk file (usually named **COMMAND**.com), which is temporarily brought into memory to execute each command. You may notice a slight pause between giving a **DOS** command and its actual execution (while loading the command processor). MegaBasic determines the name of the current command processor by reading it from the set of **MS-DOS** environment strings, available to all programs running under **MS-DOS**.

If MegaBasic cannot find the command processor on the disk under the name specified in the environment, a *File Not Found* error will occur. This can happen if you set the default drive to a drive which does not contain any command processor file. To avoid this problem, your **MS-DOS** system should employ a **CONFIG**.sys configuration file that contains the command:

```
SHELL = C:\COMMAND.COM /P
```

which specifies that the command processor is always found in the root directory on drive C: no matter what the default drive happens to be. This is especially useful on fixed-disk systems where you only need one copy of the command processor on the system. The command processor is usually re-loaded when most programs terminate, so if the system cannot find it, you have to re-boot from scratch. See your **DOS** operating system manual for other information about **CONFIG**.sys and the **SHELL** command.

The **DOS** command is also supported under the Xenix operating system version of MegaBasic. However, there are some differences that you should read about in Appendix B.1. MegaBasic does not support it under any of the **CP/M** operating systems, nor under the TurboDOS and Convergent Technology operating systems.

SERVICE #*<inter rupt number>,<proc label>*

Sets up an 8086 interrupt to access a MegaBasic subroutine (**PROC**). The interrupt number may be a value from 0 to 255; the procedure label identifies a procedure (**PROC**) which is executed when the specified interrupt is invoked. The procedure specified must contain exactly one string argument variable which communicates the **CPU** register contents to and from the interrupt caller and follows the positional assignment conventions defined for the **CALL** statement described earlier. On **RETURN**, the **CPU** registers are set to the current contents of the string variable.

As with the **CALL** statement described earlier, the register string is limited by the length of string variable used to communicate the register values. Registers defined past the end of a string variable shorter than 20 bytes will retain their contents present when the interrupt was invoked.

This statement is intended to provide an interface between your MegaBasic programs and other arbitrary programs in the same 8086 system address space. Other programs can invoke the specified **PROC** by merely executing a software interrupt instruction corresponding to the interrupt number specified. The **CPU** register values are stored in the string variable prior to beginning the **PROC** execution and on a **RETURN** statement the contents of that string variable are placed into the **CPU** registers and passed back to the interrupt caller. By using one of the registers to pass *a function number,* one interrupt *can* branch to *any* number of *separate* routines.

Up to 16 separate interrupt numbers can be independently set up using separate **SERVICE** statements. An interrupt number already defined can be redefined with another **SERVICE** statement. The set of defined interrupts is cleared each time the **RUN** command is invoked.

Although technically possible, this statement is not intended for hardware interrupt service routines written in MegaBasic statements. Such use is highly system dependent and involves hardware prioritizing which, without careful planning, can lead to hanging up the system for indefinite periods of time. The response times for MegaBasic **SERVICE** routines are usually much longer than required by most hardware interrupt applications. SERVICE routines are therefore most suitable for major activities such as transaction processing, database searches, etc.

In multi-tasking or multi-user systems, MegaBasic **SERVICE** routine requests can potentially occur during another routine execution. In such cases, MegaBasic will block further requests until the completion of the current routine. Only one **SERVICE** routine can be executing at any given instant. This means that a routine cannot itself invoke another routine (using **CALL#**) without causing the system to wait forever, unless the routine is managed by another copy of MegaBasic. If a **SERVICE** routine requires another **SERVICE** routine in the course of its operations, it need only invoke it as an actual **PROC** instead of a **CALL#**. In single-user systems, a *Interrupt Service Error* results if a **SERVICE** interrupt is invoked while MegaBasic is executing another.

SERVICE *[<memor y size required>]*

Exits MegaBasic and returns to the operating system, but leaves the current program and its variables intact for subsequent access via interrupt calls. This statement allows exiting MegaBasic after one or more interrupt routines have been setup so that other programs which use them can be subsequently executed. Notice the absence of the lb-sign (#) in this statement as compared to the earlier **SERVICE** statement.

MegaBasic normally takes up all the memory in your machine, leaving nothing for other programs to use. Under **MS-DOS**, you can release all unused memory back to the operating system, except for a fixed number of bytes specified in the **SERVICE** statement; a default value of 4096 bytes is assumed if the argument is omitted. Under CP/M-86, you must reduce the maximum memory limit of MegaBasic itself to some fixed total number of bytes, using the **CONFIG** utility program.

PARAM (<exprn>) = <exprn>

The **PARAM**(**P**) statement allows control of several internal execution factors. It may be used on the left side of an assignment statement (=) to assign new values, or accessed as a function to determine current **PARAM()** values. See the discussion of the **PARAM()** function in Chapter 9, Section 5 for complete information.

Section 4: Logical Interrupts

This section describes a mechanism allowing processes external to a running MegaBasic application to asynchronously invoke procedures within it. This mechanism, called the *logical interrupt system*, posts external *interrupts* into MegaBasic which are subsequently *acknowledged* upon completion of the MegaBasic application statement currently executing. An *interrupt service* routine, implemented in MegaBasic statements, services the logical interrupt, and when it returns, the original program resumes execution at the statement that was interrupted.

The purpose of the logical interrupt system is two-fold. First, to provide fast, real-time, event-driven response to external events for applications such as industrial process control, instrumentation and communication. Second, to provide a limited multitasking capability for special applications under single-user operating systems, such as **MS-DOS**. As we will describe below, logical interrupts have to be triggered by *external processes*, which are usually driven by *hardware* interrupts.

INTERRUPT <*logical Int*>, <*proc*>[,<*priority*>][,<*max post*>]

This specifies the MegaBasic procedure to be called when an external process triggers the logical interrupt number. Its parameters are as follows:

- Logical interrupt number from 0 to 31.
- Name of (or pointer to) a MegaBasic procedure that processes logical interrupts invoked by a process external to the running MegaBasic program.
- Optional priority level number that defaults to a priority level equal to the specified interrupt number. Priority levels do not have to be unique and may range from 0 to 255. When more than one interrupt is pending simultaneously, the one with the numerically highest priority is serviced next.
- Optional maximum number of pending interrupts that can be outstanding without causing an *overrun* (i.e., being lost). It defaults to 1 if omitted. Buffered interrupts are discussed in detail shortly.

A user-assigned *Intel 80x86* interrupt vector is used as an entry point into MegaBasic, which is divided into a set of 32 *logical* interrupts by setting AL register to a value from 0 to 31 before invoking the 80x86 software interrupt number. The term *logical interrupt is* used to distinguish them from **INTEL** *software interrupts* because they provide an *idealized* interrupt system instead a *physical* one. To supply application-specific information, the invoking process may optionally pass additional values in ES and BX when calling the 80x86 interrupt, which the MegaBasic interrupt *service* routine can access when it begins executing.

Interrupt Service Procedures

An interrupt service procedure is simply a MegaBasic procedure with *zero formal parameters*. They can be defined locally or in external packages and implemented in either MegaBasic code or machine code (in assembler packages). Generally what the procedure itself does should be kept as short as possible. A typical interrupt procedure might do nothing more than access the posted interrupt information (via the **INTERRUPT** function coming up) and add it to a queue for processing by the *foreground application*.

When an interrupt occurs, all interrupts with the same or lower priority become disabled until the procedure returns. However if an enabled interrupt with a higher priority occurs, it will be serviced as soon as the next MegaBasic statement finishes executing. As soon as the higher priority interrupt procedure returns, the lower priority procedure resumes execution, re-enabling the intervening priority levels. The user must ensure that in such a case, any common data structures used by the various interrupt service routines are accessed in a re-entrant manner. The **LOCAL** statement is useful in such cases to protect variables that need to be preserved.

Avoid operations that wait for something to occur; after all, the whole idea of logical interrupts is the elimination of wasteful polling mechanisms. For example, **INPUT** statements, **INCHR$()** function calls or anything else that waits for keyboard input should *not be used* inside logical interrupt service routines. **INPUT** statements that display prompts and accept edited input are not re-entrant, so logical interrupt service routines must avoid using them to avoid conflict. This is because interrupting an interactive input statement to execute another such **INPUT** statement will leave the original **INPUT** in an unpredictable state. The **INCHR$()** function is, however, fully re-entrant so it can be used within a service procedure without restriction.

The foreground application should avoid operations of indeterminate duration because logical interrupts are usually only serviced between MegaBasic statements and excessive statement execution time will delay interrupt response. Logical interrupts are also serviced while waiting for single character input (as in interactive **INPUT** statements and **INCHR$()** function calls) and during **WAIT** statement delays.

It is possible to block logical interrupt servicing by calls to the operating system or to other resident software that wait or execute indefinitely. This includes *shelling-out* to the operating system, direct **CALLS** to system device input (and sometimes output) functions, and MegaBasic multi-character inputs (e.g., **READ**ing serial devices, **INCHR$()** with multi-character requests).

Interrupt service procedures should not be used to process *extremely rapid* events, due to the relatively long periods that interrupts cannot be acknowledged (i.e., during statement execution). For example, one-at-a-time byte transfers at 38.4k baud could be too fast for this type of system. However events like *output buffer empty, input buffer almost full, machine tool sequence complete, timer expired, concentrator available, mouse moved, hot-key pressed, pressure threshold reached* and *message waiting* can be handled very efficiently.

Buffering Logical Interrupts

An individual logical interrupt number can handle multiple interrupts without necessarily causing an overrun condition. This is done by buffering the external interrupt post requests up to a maximum count, specified by the optional fourth parameter on the **INTERRUPT** statement, as described earlier. For example if you set <*max post*> to 4, up to 4 interrupts could be posted and pending on that interrupt without causing an overrun.

An *overrun* condition can only occur if the <*max post*> limit is exceeded or there is no more space in the interrupt posting buffer pool. There is a system-wide limit of 64 pending interrupts, so it is possible to use up all the interrupt capacity on only a few interrupts if you over-commit the interrupt capacity too far.

The <*max post*> parameter defaults to 1, which leads to interrupt overrun condition if a second interrupt is received before the first one is processed, unless you specifying a higher number. Interrupts stay posted until the processing procedure **RETURN**s, however while a logical interrupt is being serviced it is temporarily given one extra posting.

Interrupt Control

Variations of the **INTERRUPT** statement are used to enable and disable logical interrupts and to select an 80x86 INT number for access by the external processes. Logical interrupt capabilities provide a complete system for supporting real-time, asynchronous event processing. The **INTERRUPT** statement earlier merely defines a logical interrupt number, it does not enable it. Three other statements are available to enable, disable and terminate interrupts:

INTERRUPT [<logical int>], ON	Enables specified interrupt
INTERRUPT [<logical int>], STOP	Disables specified interrupt
INTERRUPT [<logical int>], END	Clears interrupt definition

If the interrupt number is omitted from the above statements, all currently defined interrupt numbers are selected by default and modified accordingly. MegaBasic reports an error on any attempt to enable or disable an undefined interrupt number. When an interrupt occurs on a disabled interrupt number, the event is still posted but not acted on. Later, if the interrupt number is re-enabled, the posted event is serviced immediately. An interrupt can be redefined by clearing (**END**) its current definition and redefining the same interrupt number in an **INTERRUPT** statement with different procedure and control settings.

Interrupt Control Information

The **INTERRUPT** function is provided so that an **INTERRUPT** *service* procedure can access the register values passed by the external invoking process, along with status information about a logical interrupt. This function is specified as follows:

INTERRUPT(***<data selector> [, <interrupt number>l)***

where the <*data selector*> chooses the value from the available set and the optional <*inter rupt number*> specifies the logical interrupt number (0 to 31) for which the data applies. If the <*inter rupt number*> *is* omitted, then the logical interrupt currently being serviced is assumed. Minus one (–1) is returned if the interrupt is *not* being serviced, or if either argument is out of range. The <*data selector*> argument may take on the following integer values:

INTERRUPTpriority level currently defined		
0	**Interrupt priority level currently defined**	
	Current interrupt status, as defined below:	
	Bit 0	Logical interrupt is defined
	Bit 1	Interrupt service is enabled
1	**Bit 2**	An external interrupt is pending
	Bit 3	Interrupt currently being serviced
	Bit 4	Not serviced in time (overrun)
2	**Value contained in the BX register when posted**	
3	**Value contained in the ES register when posted**	
4	**Number of interrupts pending on this interrupt**	
5	**Posting limit currently defined for this interrupt**	

The interrupt status value returned for selector 1 contains a number of bit flags that indicate the current state of the corresponding interrupt. These bits are most easily accessed using the & operator, e.g., **INTERRUPT(l)&4** indicates the status of bit 2.

The *overrun flag* indicates that the interrupt has been posted more than once before being serviced by the interrupt procedure. This flag is cleared only when an **INTERRUPT ON** or **STOP** statement is invoked. If overrun is a result of too many events on one logical interrupt, you can redistribute the events over multiple logical interrupts to lessen the burden or specify a higher <*max post*> limit. Overrun can also occur when your program executes operations of indeterminant duration, such as multi-character **READ**, **INCHR$** and direct **CALLS** to similar operations in the operating system and other resident software, so avoid such actions while logical interrupts are enabled.

Assigning the 80x86 Software Interrupt

The 8086 **INT** number must be defined as one of the 256 (0 to 255) hardware interrupt vectors provided by the **8086 CPU**, using the following statement:

INTERRUPT = *<cpu interrupt number>*

Defining it with the **INTERRUPT** = statement causes that interrupt vector to be linked into the MegaBasic logical interrupt system. It is this hardware **INT** that is called by external software to invoke the MegaBasic logical interrupts. Without defining it, no execution path to your logical interrupts exists. In multi-tasking systems where more than one incarnation of MegaBasic is executing, each of the MegaBasic tasks may assign a different 80x86 **INT** so that each one independently accesses separate MegaBasic programs.

Only one 80x86 **INT** vector is used by the logical interrupt system. If you re-define it as a different number, the prior **INT** vector is restored to its earlier contents and the new vector is set to point into the logical interrupt system. A defined interrupt vector is only restored when MegaBasic exits or when a program defines another 80x86 **INT** number. The pointer in the interrupt vector points to the following structure in the MegaBasic process segment:

```
CALL FAR   <entry seg>:<entry offset>
DW         <INT vector offset>
```

The **CALL FAR** is a 5 byte instruction that is followed by a word containing the offset in the interrupt table (segment 0000) of the currently defined interrupt in use. This allows external processes to determine for themselves if the interrupt vector they are about to call has been defined. This word will not match the interrupt vector offset if the interrupt vector was not setup by a MegaBasic **INTERRUPT** = <*INT*> statement. The following assembly code can be used to make this test:

```
MOV BX,Offset INTNUMB*4      ;Point BX to interrupt vector to test
PUSH DS                      ;Point DS to interrupt table segment
XOR AX,AX
MOV DS,AX
LDS Sl,Dword ptr [BX]        ;set DS:SI to interrupt vector contents
CMP BX,[SI+5]                ;compare vector offsets
POP DS                       ;Restore original DS
JNE NOTSETUP                 ;Branch it not equal to error recovery.
```

Posting Interrupts

One of the 256 interrupt vectors is reserved for use as the interface that external events signal to MegaBasic that some event has occurred. This **INT** number is defined by the **INTERRUPT** = <*number*> statement described earlier. The external process places the *logical interrupt number* in AL, sets BX and ES to an optional value to be *posted* and then calls the reserved interrupt number. This invokes a *short* routine within MegaBasic that posts the event in the internal interrupt control tables maintained by MegaBasic and then *immediately* returns.

Except for the carry flag, all **CPU** registers are preserved by the event interrupt. Invalid or undefined logical interrupt numbers in AL are ignored. Hardware interrupts are disabled during this posting operation. The posting call returns with carry set if posting fails for any reason (e.g., undefined interrupt or exceeding the posting limit), and returns with carry cleared to indicate a successful post.

It is only when MegaBasic finishes whatever statement it is currently executing, that it services a posted interrupt (assuming it has a *higher* priority than any interrupt currently being serviced). When an interrupt service procedure returns, it will resume program execution at the point it left off, as long as no other interrupts are pending.

Background Processing under MS-DOS

To provide a more complete environment for developing, testing and using the logical interrupt system, the **MS-DOS** version of MegaBasic takes advantage of the multi-tasking *hooks* provided by the **MS-DOS** operating system (as implemented on **IBM PCS** and compatible computers). These *hooks* allow background processes to execute concurrently with **DOS** shell commands and other programs unrelated to MegaBasic.

In the discussion that follows, we shall refer to executing MegaBasic logical interrupt service procedures as *background* processes and refer to all other operations *as foreground* processes.

A **DOS** statement lets you enter a *nested* invocation of the **COMMAND**.com command shell to either execute an immediate command and return, or to enter its interpretive command level for an indefinite period. During this time, if logical interrupts are invoked by interrupt driven processes, they will be acted on as if the MegaBasic program itself is running, providing a limited but effective form of concurrent processing.

Concurrent operation under **MS-DOS** works by rapidly passing the cPu back and forth between the foreground and background processes. To get from the foreground to background, **DOS** invokes INT 28h during busy loops and INT lCh on every so-called *timer-tick* (18.2 times per second). To get back to the foreground process, the background process need only return from the INT 28h or INT lCh that called it. MegaBasic does this automatically when the logical interrupt service procedures have all completed, when a predefined *time-slice* is exhausted, or when an **INTERRUPT WAIT** statement is executed (a topic discussed shortly).

Normally, the MegaBasic **DOS** statement temporarily releases all unused memory to the system so that the invoked command will have the maximum memory available for its own execution. However since this leaves nothing for the background processes to execute with, a second optional parameter to the **DOS** statement must be specified to reserve some working memory for MegaBasic operation, for example:

```
DOS "",5000
```

This statement enters the shell command level and leaves 5000 bytes of free memory for any background processing, beyond the memory already in use by the program. The memory size parameter may be any number of bytes up to the available free space, minus about 16k for the command shell process. It should generally be limited to the smallest amount of memory under which the background process can fully execute without running out of memory.

When a background process gains control during a **DOS** statement invocation, it normally retains control until it and all other outstanding logical interrupts pending is serviced and its associated interrupt procedures have all returned. In some situations, particularly when the background process is waiting for some event to take place, the background process may take up too much time before giving the foreground a chance to execute. Therefore, a special statement can be issued to give the foreground process immediate control before continuing with the next background statement. This statement is simply:

```
INTERRUPT WAIT
```

After executing this statement, the foreground process resumes until it passes control back to the background process, which continues with the statement immediately following the **INTERRUPT WAIT** statement. In background processes that wait for things to happen, as in background modem transfers, **INTERRUPT WAIT** statements should be invoked in busy waiting loops so that the background process doesn't monopolize the **CPU** unnecessarily while it waits. When no **DOS** statement is in progress, **INTERRUPT WAIT** statements are ignored when executed (i.e., they do nothing).

Background processes must never **STOP** or **END**, either directly or indirectly due to an error, while a **DOS** statement is in progress. To do so will leave the system with an active **COMMAND**.com shell running a foreground process that can neither be removed nor resumed. In such a state, you can edit and save your program to repair bugs, but the machine will crash when you exit MegaBasic. Furthermore, all available memory, except the amount you specified in the **DOS** statement, will be unavailable for any purpose. The only viable option at this point will be to re-boot the computer. To minimize the likelihood that this occurs, the following additional extensions are provided:

- MegaBasic does not recognize Ctrl-C from the console when a **DOS** statement is in progress. This is to prevent a Ctrl-C typed into a foreground process from inadvertently reaching the background process and stopping it.
- MegaBasic does not execute **DOS** statements while a *higher-level* **DOS** statement is still active and attempts to do so will be ignored. The design of **MS-DOS** does not support such an operation and permitting it would immediately crash the system.
- The **DOS** *function* can be tested at any time to determine if a **DOS** statement is currently in progress. The **DOS** function returns 1 if a **DOS** statement is in progress, 0 if no **DOS** statement is in progress, or –1 if the **DOS** statement has completed but logical interrupts serviced before its completion have not yet finished.

All logical interrupts serviced during a **DOS** statement must be completed (i.e., their service procedures must return) before the **DOS** statement is really finished so that the statement following the **DOS** statement can be executed. Until this happens, Ctrl-C and further **DOS** statements will remain disabled.

Automatic Background Processing

Up to now, we have discussed background processes that are initiated by external interrupts. However, MegaBasic also supports background processes that are invoked automatically by the **DOS**. To use this feature, an additional third (and optional fourth) parameter is appended to the **DOS** statement to enter the foreground process:

DOS *<command>,<memory size>,<int number>,<time slice>*

where <*int number*> is the logical interrupt to invoke while the **DOS** statement is in progress. It may be any of the 32 logical interrupts, but it must defined and enabled before you issue the **DOS** statement (otherwise nothing will happen). When the **DOS** statement completes, the program resumes with the statement that follows it and the background process invoked by the interrupt specified will no longer be invoked.

The <*time slice*> argument may optionally specify the maximum amount of time to be given to the background process without yielding to the foreground process. This is called the time slice limit and it is specified as an integer number of milliseconds from 0 to 65535. If you omit this argument, a default of 65535 is used (about 65 seconds). When a time slice is used up, MegaBasic generates an automatic **INTERRUPT WAIT** so that the foreground process can resume. **INTERRUPT WAIT** statements can still be used to further break up a time slice as needed, but they are not necessary in most background applications if you specify an appropriate time slice. The interval specified is rounded up to the nearest system *timer-tick* provided under the host system.

If the automatic interrupt is the only logical interrupt being used, you do not have to define any 8086 software **INT** in order for it to operate. The automatic interrupts are invoked through the **DOS INTS** lCh and 28h and therefore no other vectors are needed. All you have to do is:

- Define the logical interrupt with its procedure and priority
- Enable the logical interrupt
- Issue a **DOS** statement that reserves enough memory and specifies the logical interrupt number to be invoked repeatedly by the **DOS**.

To see how all this is put together to form an actual background MegaBasic program, an example background process that occasionally displays the time of day now follows:

```
10 Interrupt 1 ,TEST,4; Rem -- Define a logical interrupt
20 Interrupt on; Rem -- Enable logical interrupt 1
30 Dos "",5000,1,10; Rem -- Enter DOS, begin background
40 End; Rem -- Done upon user "Exit"

50 Def proc TEST
60 Repeat; I += 1; If not I mod 1000 then Print time$,dos
70 next; If dos<1 then Return
80 Proc end
```

Procedure **TEST** prints the current time of day on the screen whenever variable I increments to a multiple of 1000. In line 70, the background status is tested so that the procedure only returns after the user types **EXIT** in the **DOS** command level (terminating the background process). The apparent effect of all of this is that the program executes concurrently along with whatever you happen to be running in the foreground. To terminate the background process, type the **EXIT** command from the **DOS** command level.

Background Process Termination

A MegaBasic background process can terminate at any point, either normally or through an untrapped error. What occurs is that when the process terminates (e.g., via **END**, **DOS**, **STOP** or an error), control is permanently passed back to the foreground. However, the background program remains suspended in memory until you give the **EXIT** command from the **DOS** *command shell*. At that point, the background process *formally* terminates, releasing all its memory, leaving you again at the **DOS** command shell prompt.

Background Time-Slice Control

Ideally, you never want to experience any *visible* delays in a foreground process due to excessively long time-slices taken by a background process. There are two ways that background processes can begin their time-slice: through the *timer-tick* (**INT 1Ch**) and through the **DOS** *idle loop* (**INT 28h**). In general running off the timer-tick is *more visible* in the foreground process than running off the idle loop, particularly for computationally intensive background processes. Knowing which method began the current time-slice, enables you to use **INTERRUPT WAIT** to shorten unnecessary time-slices.

Hence **INTERRUPT(2)** returns 0 or 1 in *background processes* to indicate the *source* of the current time-slice: from the *idle-loop or* the *timer-tick*, respectively. For example, suppose your background process has a loop where is spends a lot of time. An **INTERRUPT WAIT** if **INTERRUPT(2)**>0 at the top of each iteration allows only one loop iteration during *timer-ticks*, but a full time-slice during *idle loops*. **INTERRUPT(2)** normally returns the BX register value that is posted by *logical interrupts*, a register not set by background processes.

Chapter 8 User-Defined Subroutines

One of the most powerful aids for controlling complexity of software systems is the principle of modular design. As programs become longer, they also become more complex and difficult to work on. To maintain the simplicity of small programs, large problems can be broken down into a set of smaller component problems which, by themselves, are easier to solve. If some of these component problems are still too complex to deal with, they can be further subdivided until the subcomponents become manageable. Corresponding to each of these sub-components is a program module designed to solve its problem. This is the technique of modular design, also called *divide and conquer*, which is implemented in programming languages using the construct called *subroutines*. The concepts and techniques you use to build and use subroutines are covered in this chapter, as summarized below:

Subroutine Statements	Summary of all MegaBasic statements involving user-defined functions and subroutines.
Elements of Subroutines	Essential concepts and related ideas necessary to building subroutines.
Types of Subroutines	The different mechanisms involved in the three types of subroutines supported under MegaBasic: **GOSUBS**, procedures and functions.
Communicating with Devices	Detailed descriptions of all available techniques for passing information between subroutines and their calling references.
Recursive Programming	How subroutines can invoke themselves as a powerful means for reducing the size and complexity of the solution being implemented.

In MegaBasic, as in most programming languages, modules are implemented with *subroutines*, sometimes known as *subprograms*. In their simplest form, subroutines consist of some sequence of program statements that can be invoked from elsewhere in the program by merely referring to them by name. This program sequence is not some random collection of statements, but rather a coherent solution to one problem or subproblem. To hide the details of the subsolutions required to implement them, subroutines usually refer to other subroutines for that subsolution. Hence large programs tend to be structured as hierarchies of subroutines where most of the actual details are carried out at the *lowest levels*.

This chapter discusses the most powerful feature of MegaBasic: subroutine construction and usage. It describes the *collection* of features that let you create your own additions to the language as you see fit, extending its capabilities and tailoring its facilities toward your own special needs. These features permit previously developed programs to be used as the building blocks of new, larger programs, which, on completion, become the *primitive components* of still larger systems.

To put it simply, a subroutine is nothing more than a section of your program that performs a *specific task*, that has been set aside and given a *name* for you to refer to (and execute) from *anywhere* in your program. As you develop programs from subroutine building blocks, you will gradually accumulate your own useful set of subroutines that can be used to build new programs with far less effort than earlier ones. In the sections that follow, we assume that you understand how to build and use programs, and concentrate on the new concepts about subroutines.

Chapter 10 covers the concept of *packages*, which lets you collect many of your subroutines and global data variables into external libraries that can be accessed by your main program as MegaBasic *extensions*. Subroutines can also be developed in assembler (Chapter 10, Section 5) and packages with one or more assembler routines can be accessed by your program like any other package.

The **`LIBRARY.PGM`** file included with the MegaBasic release contains many examples of useful functions and procedures which illustrate how subroutines can be built, documented and made accessible for general use by other programs you write. Meaningful names, line-labels and identifiers are used to clarify their usage and understanding and they may be freely applied to your own programs without any further permission from the author. Refer to these routines for more examples of the concepts described in this section.

Section 1: Subroutine Statements

MegaBasic provides three types of user-defined subroutines: **GOSUBS**, functions and procedures. A **GOSUB** is simply a means for re-using a section of program lines from any place in the program. User-defined functions are used like variables for data in string or numeric expressions, except that they represent computed results rather than stored values. Procedures are like **GOSUBS**, except that they are invoked by name instead of by program line location, and they can supply argument list parameters. This section describes all the MegaBasic statements used to create subroutines, summarized as follows:

GOSUB	Invokes a sequence of program lines which returns back when a **RETURN** statement is encountered.
ON..GOSUB	Selects and executes a **GOSUB** from a list of **GOSUBS** using a computed index into the list.
RETURN	Causes the currently executing **GOSUB**, function or procedure to *return* to the program location from which it was originally called.
LOCAL	Creates temporary string or numeric variables for use within a **GOSUB**, function or procedure. These variables *disappear* after the subroutine executes its **RETURN** statement.
DEF FUNC	Defines the name, arguments, entry point and result type of a user-defined function.
FUNC END	Defines the end of a user-defined function.
DEF PROC	Defines the name and argument structure of a user-defined procedure.
PROC END	Defines the end of a user-defined procedure.
ARGUMENT	Statement that accesses arguments from an *open-ended* argument list.

You can communicate data to procedures and functions through an argument list, which is used in computing their intended task. Functions always compute a single result which is used in the expression that invoked the function.

GOSUB *<label>*

Short for ***GOTO*** *Subroutine,* a **GOSUB** statement transfers program control to the line specified (by line number or line-label) as with the **GOTO** statement. The line transferred to must be in the same program (package) as the **GOSUB** statement. When a **GOSUB** subroutine has finished its work, the program resumes execution at the statement following the **GOSUB** statement. To signal this termination response, a companion statement called a **RETURN** must be executed. MegaBasic keeps track of the statement following the **GOSUB** so that when a **RETURN** statement is encountered, control returns to the right place. There may be any number of **RETURN** statements within the body of the **GOSUB** subroutine, and each of them will resume execution at the same point in the program.

GOSUBS are best visualized as program blocks that perform a procedure as an operational unit. Although the body of a **GOSUB** has no obvious structure required by

MegaBasic language syntax, it is important to treat it as a unit by clearly defining its entry and exit points and using them in rigidly controlled ways. You should not jump out of a **GOSUB** using any type of **GOTO** statement, for the purpose of bypassing the **RETURN** statement. The **RETURN** mechanism that remembers where to continue after each **GOSUB** returns cannot take such exits into account and *unpredictable behavior* may result. You can use a **GOTO** to instead of a **RETURN**, if that **GOTO** jumps somewhere that *eventually* does execute a **RETURN** statement, like another **GOSUB**.

MegaBasic provides several mechanisms that depend on well-defined block structured **GOSUBS** to be useful. Error processing structures (**ERRSETS**) and **LOCAL** variables are local within **GOSUBS**, meaning that changes made to them do not propagate back up through to the program when the **GOSUB** returns. You can define your own **LOCAL** variables within **GOSUBS** which may be used in any way whatsoever without affecting anything outside that **GOSUB** (discussed below). Awareness of these features is necessary for proper programming of **GOSUBS** and **ERRSET** processing.

New programs should favor user-defined procedures over **GOSUBS**, because they perform the same function in a cleaner way and have additional features that make them far more versatile. **GOSUBS** will always be supported in MegaBasic for compatibility with existing programs that use them.

ON *<exprn>* GOSUB *<line list>*

Selects and invokes one **GOSUB** from a list of **GOSUBS**, according to a computed number that specifies which **GOSUB** in the list to invoke. The *<line list> is* a series of line numbers or line labels, separated by commas, which identify each of the **GOSUBS** from which the computed choice is made. The *<expr n> is* a real or integer expression which, after being truncated to an integer, must evaluate to the position in the *<line list>* corresponding to the desired **GOSUB** line number. This integer must be from 1 to the length of the *<line list>*. Upon completion of that **GOSUB** (after its **RETURN** is executed), program execution resumes at the statement that follows the *<line list>*. This is sometimes referred to as a computed **GOSUB** because it executes a **GOSUB** based upon a computed value.

RETURN *[<expr n>]*

Directs program control to the statement following the most recent **GOSUB** or procedure call, or returns a string or numeric expression result from a user defined function. The *<expr n> is* only specified in **RETURNS** from functions, which is the only way a function result is made available.

Before the actual return, **RETURN** restores the states of the previous **ERRSET** structure and of any **LOCAL** variables (see **LOCAL** statement below) to their states at the time of the **GOSUB** or user function call. A **RETURN** statement is permitted anywhere within a subroutine, even inside a **FOR** or **WHILE** loop. **RETURN** statements may appear in as many places within a subroutine as needed.

When you use a **RETURN** statement to return from a user-defined function, you must follow the **RETURN** reserved word with an expression that provides the result value of the function. The data type of this expression must match the data type of the user-defined function, i.e., string functions must return a string, numeric functions must return a number. A *Data Type Error will* occur if such a type mismatch is specified.

Numeric functions are defined as either integer or real functions, which means that the numeric result that they return is always a number of the type defined. However, if you

specify a real result for an integer function (or vice versa), MegaBasic will automatically convert the expression result to the proper numeric type (i.e., *integer-to-real* or *real-to-integer).* This kind of type mismatch is permitted and no error is reported for its occurrence, unless for some reason the type conversion could not be completed (e.g., a real result too large to fit into an integer representation). Nonetheless, you should always try to provide numbers in the proper representation for the context in which they are to be used for the most efficient implementation.

LOCAL *<list of string and scalar variables>*

Creates temporary simple string and numeric variables (not arrays) which may be used freely for any purpose within any **GOSUB**, user-defined function or procedure. **LOCAL** statements generally appear as one of the first things done within subroutines that use them, rather than in the middle somewhere. Global variables of the same name which already exist are protected but inaccessible until the subroutine executes a **RETURN** statement. Scalar and string variables may be listed separated by commas after the **LOCAL** keyword, and can then be used for unlimited local working storage.

Since local variables carry their previous value after the **LOCAL** declaration, they may be employed for passing data parameters to **GOSUBS**. On **RETURN**, their prior values are restored, and program execution resumes. Used within recursive procedures (Chapter 8, Section 5) to create temporary working variables, this is a particularly useful and powerful tool.

Subscripted variables cannot be **LOCALized**. **LOCAL** declarations are permitted only within subroutines and using them directly inside **FOR** or **WHILE** loops will cause an error. **Re-DIMensioning** of a **LOCAL** string, although not recommended, is permitted as long as its previous string value will fit (i.e., it *can* be restored) upon **RETURN**.

String variables of any length can be localized, but since their previous contents are saved on the scratchpad until a **RETURN** is executed, really long strings can easily use up all the storage set aside for this purpose. Hence **LOCAL** strings should be used sparingly and with great care to avoid such problems. The amount of local storage available at any instant can be obtained from the **FREE(2)** utility function. The following example illustrates how **LOCAL** variables are confined to subroutines:

```
10 X = 99999; Y = X; TEXT$ = “text string”
20 Gosub 50; Print TEXT$,X,Y;
30 End

50 Local X,TEXT$
60 X – –1; Y = X; TEXT$ = “######”
70 Return
```

When this program is run, the **PRINT** statement in line 20 will display the line: *text string 99999 –1.* Notice how the **GOSUB** was unable to modify both X and TEXT$ (because they were declared **LOCAL**) but was able to modify Y because it was not declared **LOCAL**.

When they are only being employed to prevent interference with variable values, **LOCAL** variables are not needed within subroutines which are always called from other packages (Chapter 10). All variables within external packages are implicitly local to that package unless explicitly declared **SHARED**. Such variables are physically different from those in other packages that happen to have the same name.

DEF *[<modifiers>]* FUNC *<name> [(<arg list>)] [=<exprn>]*

Defines a user-defined function, including its name, its list of arguments and its mode of operation: single line or multiple line. A function name may be any legal variable name (Chapter 1, Section 5). Such names do not have to begin with the letters *FN* as in other **BASICS**.

The **DEF** statement must appear as the first statement on the line in which it appears and it cannot be preceded by a line label. Chapter 8, Section 3 contains further details on user-defined functions. The optional set of <*modifiers*> *is* used to specify the access scope (**SHARED**) of each function, and the result type (**STRING**, **INTEGER** or **REAL**) of numeric functions. These modifiers are described as follows:

SHARED	Indicates that the function may be invoked from external packages in a multiple package program. See Chapter 10, Section 2 for further details about this.
INTEGER	Specified only for numeric functions that return an integer result, rather than a real result. All numeric functions are real unless you declare them to be integer, or if the leading letter of its name has already been classified as integer by a **DEF** statement.
REAL	Specified only for numeric functions that return a real result, rather than an integer result. This modifier need only be supplied if an earlier **DEF** statement has already classified the leading letter of the function name as integer, because functions are normally real by default.
STRING	Specifies that the function returns a string result. This is not required if the function name end with a dollar sign ($) or if the leading letter of the function name has been declared as a string already. Its presence with such names is, however, permitted and and may improve program readability.

Since numeric functions always return either an integer or a real result, you must somehow indicate the result type of each function. This is may be done explicitly using the above modifiers in the **DEF** statement of the function, or implicitly by allowing the default type to be imposed by omitting the type from the **DEF** statement. Without the word **STRING**, **INTEGER** or **REAL**, a function s result type is implied from its name. However, it helps program readability if all functions explicitly declare their result type. The rules for assigning data types to names are discussed in Chapter 3, Section 1, Chapter 4, Section 2 and Chapter 5, Section 1. See the **RETURN** statement (Chapter 8, Section 1) for more information about returning the result of a function.

When a function is activated by using it in an expression, an argument list is supplied which defines values for each of the argument list variables in the order given by the **DEF FUNC** statement. Such values may be specified by a general expression, but they always appear within the function itself as the contents of the argument list variables.

The optional argument list consists of a sequence of unindexed string or numeric variable names separate by commas and enclosed in parentheses. These variables, called formal arguments, serve during actual use of the function to hold the data passed to the function so that they can be processed to form the ultimate result returned. If no

argument list is specified in this definition statement, then no arguments can be passed to the function when it is actually used.

Argument list variables may be either string, integer or real variables, which corresponds to the argument type to be passed through that argument in the list. You should specify numeric argument list variables that possess the appropriate integer or real type for the context in which they are to be used. This avoids unnecessary type conversions which can degrade the performance of your program.

An argument list has several other optional advanced features which are discussed in Chapter 8, Section 4. These features include passing data in both directions through the parameter variables, optional parameters that take on default values when omitted, and open-ended parameter lists that can have any number of parameters.

If the =<*expr ession*> at the end of the **DEF** statement is omitted, then a multiple line function is defined whose procedural definition must follow. The main body of a multi-line function consists of a sequence of statements that includes at least one **RETURN** <*expr n*> statement and physically ends with a **FUNC END** statement (described below). The following example illustrates a useful multi-line function:

```
40 Def string func NTH(N); Local P
50 If N mod 100 > 3 and N mod 100 < 21
   then Return str$(N)+“th”
60 P = min(N mod 10,4)*2+1; Return
str$(N)+“thstndrdth”(P:2)
70 Func End
```

Typing this direct statement	**For l=1 to 10; Print NTH(I),; Next**
produces this output	**1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th**

FUNC END

Used as the last statement of a multiple-line function to indicate where its **DEFinition** ends. Unlike the **DEF** statement, which must be the first statement on a line, the **FUNC END** statement may appear anywhere on a line as long as it is the last statement of the function. Single-line functions do not use the **FUNC END** statement, but they are mandatory in multi-line functions.

DEF [SHARED] PROC *<proc name> [<arg list>]*

Defines the name and argument structure of a user-defined procedure. Without any arguments, a procedure is virtually identical with a **GOSUB**. To use a procedure you merely type its name, along with any required arguments (and you do not type the word ***PROC*** or **GOSUB** in front). Procedures are permitted anywhere that a MegaBasic statement is expected and in fact, appear so much like statements that you may have a hard time telling them apart. The **SHARED** modifier is needed only to allow access to the procedure from other packages in a multiple package program (see Chapter 10 for details).

The <*ar gument list*> of a procedure is identical with the <*ar gument list*> of a function except that it is not enclosed in parentheses. Function arguments require parentheses to separate them from surrounding expression terms. Procedures cannot be invoked from

within expressions and therefore the parentheses are not needed (and, in fact, MegaBasic reports an error if they are used). See Chapter 8, Section 4 for important details concerning the more advanced features of argument lists.

Procedures may have any name that would be legal as a variable name as long as the name is not used for another purpose elsewhere in your program. You can even assign procedure names that end with a dollar sign ($) or a percent (%), although such names are normally reserved for strings and integers. Procedure names do not represent data, so they do not have a data type as variables and functions do.

The body of a procedure consists of any sequence of program statements which ultimately must lead to a **RETURN** statement. More than one **RETURN** may appear in a procedure if needed. The very last statement of a procedure must be a **PROC *END*** (see below). Procedures are almost identical with multiline user-defined functions, except that they are not used in expressions and do not return a result via the **RETURN** statement. The following useful procedure illustrates some of these concepts:

```
100 Rem *** Sort VALUE(L) through VALUE(H) with Quicksort
105 Def shared proc SORT @ VALUE,L,H; Local T,l,J
110 REPEAT; T=VALUE((L+H) div2); I=L; J=H
115 While J>-L and VALUE(J)>T; J =1 ; Next
120 While l<=H and VALUE(I)<T; I += 1; Next
125 If l<=J then
   [Swap VALUE(I),VALUE(J); I +- 1; J = 1;
    If l<=J then 115]
130 If J-L<H-I then Swap H,J else Swap L,l
135 SORT VALUE,L,H; L=l; H=J; Next if L<H
140 Return; Proc end
```

This procedure sorts the contents of a range elements in a numeric array into ascending order. Once defined, its use is as simple as:

```
SORT ARRAY,FIRST,LAST
```

Notice how the procedure call appears as if it were a standard MegaBasic statement. Procedures are specifically designed to encourage the definition of your own *new* additions to the working set of MegaBasic facilities. Their similarity to statements is intentional so that you do not have to remember and apply a separate set of rules to use them. Read Chapter 8, Section 4 for important additional features of argument lists which make procedures more versatile.

PROCEND

Used as the last statement of a user-defined procedure to indicate where its **DEFinition** ends. Unlike the **DEF** statement, which must be the first statement on a line, the **PROC END** statement may appear anywhere on a line as long as it is the last physical statement of the procedure.

ARGUMENT *<list of variables>*

Reads a sequence of values from an open-ended argument list into a set of variables. The variable types must match the actual argument types being read. An error occurs if any types mismatch or if there are more variables specified than the number of actual arguments remaining. The **ARGUMENT()** function will tell you if more actual arguments remain. Open-ended argument lists are a special feature of MegaBasic procedures and functions described on Chapter 8, Section 4.

Section 2: Elements of Subroutines

Subroutines in MegaBasic are provided as a set of related program constructs which are very simple and natural to use, yet they provide tremendous generality in their application. Their effective use, however, requires that you understand the concepts and motivation behind them. Several types of subroutines are supported under MegaBasic, but they all involve the following ideas in varying proportions:

Invocation by Name

A subroutine requires some means of identifying it. Hence each subroutine has a name of some sort. All subroutines are invoked by merely referring to their names. These names generally refer to the location in the program where the subroutine is defined. Procedures and functions can even reside in other external *packages* (described in Chapter 10).

A Single Entry Point

A subroutine must have some well-defined point at which it begins execution. Any subroutine that seems to have more than one entry point should really be treated as several subroutines, one for each entry point. This concept is enforced by the fact that subroutines are invoked by name, as described above.

One or More Exit Points

As with any program, a subroutine must at some point complete its assigned duties and terminate. However since subroutines are invoked by programs or other subroutines, they should not stop the entire execution process. Instead, subroutines must terminate and then resume execution at the statement following their invoking reference. A special subroutine termination statement called **RETURN** is used for exactly this purpose. While a subroutine is executing, it may *decide* at any point that it has finished, so MegaBasic allows a **RETURN** statement to appear anywhere (even within loops, **CASE** statements and other *block* structures).

Communication of Input Data

Subroutines would be rather useless if they could only use one particular set of input data. MegaBasic provides various mechanisms for communicating input data to subroutines at the time that they are invoked. All of these methods ultimately reduce to passing input data to the subroutine through a well defined set of variables *known* to the subroutine. MegaBasic provides a wide variety of methods for passing information through argument lists (Chapter 8, Section 4).

Communication of Output Results

The existence of a subroutine is justified only if it produces a result or causes some effect that is useful in some way. Various mechanisms are provided in MegaBasic for communicating these results directly to other places in the program that need them. For example, function subroutines return a single value back to the invoking computation directly, whereas procedure subroutines return results through argument or global variables. Subroutines can also generate results which, instead of communicating with the program, communicate with *external* files or devices.

Independence, Isolation and Information Hiding

It is highly desirable to be able to use subroutines as building blocks without having to know how they work (e.g., you don't need to understand the molecular structure of bricks in order to use them to build a house). Subroutines must therefore effectively *hide* their internal details from the context in which they are used. This is called *information hiding,* which is a primary reason subroutines are so important in modern software development.

Another aspect of information hiding is the need to isolate the actions of subroutines so that they do not have any obscure or otherwise unplanned effects on the surrounding (invoking) context. The *package* mechanism of MegaBasic provides an effective barrier between external subroutines and their local references. Other mechanisms MegaBasic provides for isolating subroutines include: **ERRSETS**, **LOCAL** statements, argument lists, etc.

Section 3: Types of Subroutines

MegaBasic supports three different types of subroutines: **GOSUBS**, functions and procedures. The facilities provided for defining and using them are covered earlier in this Chapter and you should be somewhat familiar with that material before reading on. The only difference between **GOSUBS**, functions and procedures, other than superficial syntax differences, is the manner in which input and output information is conveyed between the subroutine and its caller.

An extremely important means for communicating input data to and output results from functions and procedures is a language construct called an argument list. Argument lists are fully discussed later in Chapter 8, Section 4, and the discussion below glosses over them to avoid the extra detail.

GOSUB Subroutines

The standard method for building program modules in all **BASICS** is the **GOSUB**, which is simply a sequence of program steps which can be invoked from anywhere in the program, without having to be typed in repeatedly. A **GOSUB** is universally accepted in all **BASICS** with substantially the same meaning and form. The **GOSUB** itself is any sequence of program statements that eventually terminates with a **RETURN** statement. It is invoked by specifying its beginning line in a **GOSUB** statement, such as:

```
GOSUB 1010
```

which simply says *begin execution at line 1010 and come back when a* **RETURN** statement is encountered. In MegaBasic, you can assign names to individual lines and such lines can then be referred to by name as well as by line number. If line 1010 were to be named ***SORT:*** then the above **GOSUB** reference could be stated as follows:

```
GOSUB SORT
```

Such names have an immense effect upon the readability of your program and their use is highly recommended. The naming rules are described in detail on Chapter 2, Section 5. There is nothing at all special about the program statements performed within a **GOSUB**, except that they must eventually execute a **RETURN** statement. Since it is a fixed sequence of statements which only perform one specific set of actions, applying a **GOSUB** to different situations can be rather cumbersome. Take, for example, the following simple sort **GOSUB:**

```
1000 Rem *** Sorts the N values of array TABLE
1010 SORT: For l=1 to N-1; For J=1+1 to N
1020 If TABLE(I)>TABLE(J) then Swap
TABLE(I),TABLE(J)
1030 Next J; Next l; Return
```

To apply **GOSUB SORT** to any set of numbers involves loading the numbers you wish to sort into array **TABLE()**, setting N to how many numbers it contains, then invoking the subroutine with the statement: **GOSUB SORT** (or **GOSUB 20**). It is easy to see that in some situations, just setting up the variables in preparation for using a **GOSUB** might well require more effort than what the **GOSUB** itself performs. **GOSUBS** are useful in simple applications, but MegaBasic provides a much more powerful and flexible construct, called a procedure, which you should use when new programs are developed. MegaBasic supports **GOSUBS** primarily to support existing programs that use them.

Procedure Subroutines

Procedures differ from **GOSUBS** in three ways. First, their definition begins with a **DEF** statement that gives the procedure a name and creates a set of channels through which information is communicated to and from the procedure, called arguments. Second, procedures are invoked by stating their name followed by any required input data (no **GOSUB** prefix is used). Procedure calls therefore appear quite similar to built-in MegaBasic statements. Conversely, new statements may be added to the built-in set by defining them as MegaBasic procedures. Third, your programs can refer to procedures defined in other programs if necessary, which further enhances the view that procedures are language extensions. This particular topic will be covered later on in Chapter 10. The preceding sort **GOSUB** has been rewritten as a procedure for the following example:

```
1010 Rem *** Sorts the N values of array TABLE
1015 Def proc SORT @TABLE,N
1020 For l=1 to N–1; For J=1+1 to N
1030 If TABLE(I)>TABLE(J) then Swap
TABLE(I),TABLE(J)
1040 Next J; Next l; Return
1050 Proc End
```

The only difference with the **GOSUB** is that line 1015 has been added to define the procedure name and its input data. The input data is communicated to the procedure as a list of arguments, a subject to be covered in the next chapter. **SORT** as defined by this procedure can be invoked with different sets of input data without any additional changes, as follows:

```
SORT NTBL,200
SORT KEYS,LENGTH-UNUSED
SORT VECTOR,M*N
```

where NTABLE, KEYS and VECTOR are all one-dimensional arrays containing the data to be sorted. Communicating data via this mechanism provides tremendous flexibility for applying procedures to varying situations. Procedures are therefore recommended over **GOSUBS** in all significant applications. A good rule of thumb is to use **GOSUBS** only for very small subroutines which are called from nearby lines of a single larger routine. **GOSUBS** in existing programs can be easily converted to procedures which have no arguments to begin with, and later enhanced with argument lists as needed. (You can also leave in all **GOSUBS** if you don't want to bother converting them into procedures.)

Defining Procedure Subroutines

Procedures are defined with three components: a definition header statement, a body of statements that performs the desired task, and a terminating statement. The definition statement and structure of procedures is similar to that of functions, as follows:

```
DEF PROC <procedure name> <optional argument list>
Any number of MegaBasic statements that
eventually executes a RETURN statement
PROC END
```

The *<procedure name>* may be any legal variable name which is not used anywhere else in the program (i.e., it must be unique). A dollar sign ($) is permitted to appear as the last character of a procedure name but since procedures do not have a type associated with them, it does not imply a string type (as in variables and functions). This statement must appear as the first statement on the line in which it appears. The **DEF** and **PROC** keywords must appear as the first two words in this header.

The *<optional argument list> is* a list of variables, separated with commas, through which input data and output results may be communicated. This construct permits the procedure to deal with one set of variables that represent any data being communicated with its user. This important topic is thoroughly treated Chapter 8, Section 4. Procedure argument lists of both their definition and their references are never surrounded by parentheses as they are in functions, which gives procedure calls the appearance of MegaBasic statements.

The procedure body of statements is exactly like the body of a **GOSUB** designed for the same task. It must have at least one **RETURN** statement so that it can continue execution in its *calling* routine when finished. The very last statement of a procedure definition must be **PROC END**, which defines the end of the procedure definition.

Function Subroutines

Functions differ from procedures in two ways. First, functions are invoked from within string or numeric expressions, rather than as statements in themselves. Second, functions must return a single result value (string or numeric) back to their invoking expressions, as part of their final termination. This result is returned by specifying it in the terminating **RETURN** statement (Chapter 8, Section 1). Functions are identical with procedures in all other respects.

By defining your own functions, frequently used computations can be programmed once, and later referred to by name as often as necessary anywhere else in your program. This centralizes its internal implementation details in one place in the program, so that if the computation is modified in the future, all places that use it are automatically updated. Furthermore, properly designed functions can be independently used without knowledge or consideration of their internal workings, freeing you to solve the problems at hand instead of being side-tracked by lower level details.

As with built-in functions, user-defined functions are named and include an argument list. Any name legal for assignment to a variable is also legal as a function name. As with variables, function names must reflect the data type that is returned. A dollar sign ($) ending the name indicates that the function generates a string result; functions named without a dollar sign must generate a numeric result. Unlike most **BASICS**, function names do not have to begin with the letters *FN*. Once assigned as a function, a particular name cannot be used for any other purpose. Name formation in MegaBasic is covered in detail on Chapter 1, Section 5.

The argument list of a function must be enclosed in parentheses and may contain string or numeric argument expressions. Both the number of arguments and their data type must correspond to your definition of the function (described below). The argument list doesn't appear if no arguments are required for its operation. Argument lists are discussed in Chapter 8, Section 4.

MegaBasic supports two different forms of function definition. The simplest and most common form is called a single-line function. Functions of this type merely define an expression (string or numeric) which is evaluated when the function is invoked. Such functions are then applied as shorthand for the expression whenever required.

For more complex applications, multi-line functions can be defined. These are not restricted to just one line or one fixed result expression. Multi-line functions may contain as many statements as necessary to compute the desired result. Because no limit is imposed on their size or content, such functions may perform many complex computations, alter global data structures, or perform input and output transfers prior to returning their ultimate result.

Defining Single-Line Function Subroutines

Single line functions are completely defined by a single program statement which must fit within one program line, up to 255 characters long, and has the following form:

DEF*[<type>]* **FUNC** ***<name> [(<argument list>)] = <expression>***

The <*name*> specifies the unique name by which the function is referred to throughout the program. The optional <*argument list*> lists in parentheses the variables through which the argument data is passed to the function. The <*expression*> specifies a string or numeric expression combining the arguments (with possibly other data) into a new result which is passed back to expression invoking the function.

An optional <*type*> may be specified on numeric functions to control the result type of the function. You may specify one of the words **STRING**, **INTEGER**, or **REAL** for this option. If you omit the <*type*> option, the function result type is derived from the function name itself. Names ending with a percent sign (%) are integer; names ending with an exclamation mark (!) are real; and names ending with a dollar sign ($) are string. Otherwise the function result type depends on earlier **DEF** statements (Chapter 5, Section 1) which may have assigned a numeric or string type to the leading letter of the function name. In the absence of any type declaration, a function will return a real result by default. We recommend that all function definitions explicitly include its full type declaration as a matter of programming style and general readability.

The result <*type*> merely ensures that the result is always of the desired type specified, regardless of the actual numeric computations performed within the function and the result expression. MegaBasic will automatically convert the numeric result of a function to its defined type whenever the type of the **RETURNed** result differs. A *Data Type Error* wi1l be reported if you attempt to return a string result from a numeric function or return a numeric result from a string function.

Single-line functions provide a simple way to combine data using a complex expression, for example the definition:

```
Def real func MODULO(N,M) = N–INT(N/M)*M
```

The variables N and M are function arguments which will be used to represent the data presented by an actual reference to the function, such as in:

```
MODULO(X–1 7,SQRT(Y))
```

When this reference to **MODULO** is made, the formal arguments, N and M, are set to the values expressed by X–17 and SQRT(Y) respectively. N and M are then used within the expression given in the **DEFinition** of **MODULO**, i.e., N–INT(N / M)*M. This expression is evaluated and the result is passed back as the *value* of **MODULO**, which may then be invoked within a *higher level* expression, as in:

```
X = LOG( MODULO(X–17,SQRT(Y)) + MODULO(R/S–10,T) )
```

This is a complete assignment statement which sets X to the logarithm of the sum of two different references to **MODULO** in the same expression. Numeric argument definition variables, like N and M above, have no relation to variables of the same name used outside the function definition, because they exist only during the active execution of the function. The argument list has some important properties and options which you should read about in Chapter 8, Section 4.

Defining Multiple-Line Function Subroutines

Multiple line functions permit construction of functions with any number of statements. Similar to procedures, the definition has three parts: the **DEF** statement similar to the above, the main body of the function procedure, and a **FUNC END** statement to terminate the definition:

```
DEF [<type>l FUNC <function name> [(<argument list>)]
  Any number of MegaBasic statements which
  eventually executes: RETURN <result exprn>
FUNC END
```

The only difference between single and multiple line function **DEF** statements is the absence of the equal sign (=) and <*expr ession*>. Instead, the main body of the function immediately follows the **DEF** statement. This main body consists of whatever series of program statements are necessary to perform the desired task and return the result (except that it cannot include another **DEF** statement).

To pass the result back to the expression that invoked the function, a multiline function executes a **RETURN** statement specifying an expression that computes the desired result. Any number of **RETURN** statements may appear within a multiple line function (just like a procedure), and when any one of them is executed, its result expression is computed and passed back to whatever expression invoked the function, causing the program to resume from that point.

The <*r esult exprn*> must compute a numeric result for numeric functions and a string result for string functions. As described for single-line functions, an optional <*type*> can be specified to define a numeric function result as **STRING**, **INTEGER** or **REAL**. You should understand the material presented in Chapter 3 in order to make the best choice of real and integer numeric types.

Illustrated below is a small multi-line function to test a string to see if it is a valid number:

```
10 Input “Enter a numeric string –– ”,A$
20 If VALSTR(A$) Goto 10
30 Print “Not a valid number, try again”
40 Goto 10

100 Def integer func VALSTR(TRY$)
105 Errset 115
110 E=VAL(TRY$); Return 1
115 Return 0; Func end
```

This simple but useful function logically tests a string for whether or not it correctly represents a numeric constant. It returns 0 (false) if an error occurs when **VAL**(N$) attempts to convert N$ to a number, or returns 1 (true) if successful. Hence **VALSTR**(“234O17”) returns 1 and **VALSTR**(“$7,235.98”) returns 0. **VALSTR( )** is defined as an integer function because its logical result is returned faster.

Multiple line functions must physically terminate with a **FUNC END** statement. You cannot define other functions within function definitions, but you can define them in terms of other functions by employing user-defined functions as components to compute higher level results.

Side-Effects Produced by Subroutines

User subroutine references invoke the MegaBasic subroutines within a larger context of *higher-level* program operations or expressions. When subroutines return, this execution context resumes where it left off and the program continues. Ideally, this context should be unaltered by the act of calling the subroutine, other than the result intended. However there are two areas where subroutine calls can potentially upset program integrity in non-trivial and unobvious ways. These are known as side-effects.

The first side-effect is the problem of global variables changed within the subroutine and is one of the most frequently encountered sources of programming errors when programming in **BASIC** with subroutines. In the example below, variables outside a function subroutine are affected by the function call, causing the **FOR..NEXT** loop in line 200 to continue forever:

```
200 For l=100 to 1 by –1
210 A(l)=SUMM(I); Next; End
800 Def func SUMM(N); T=0
805 For l=1 TO N; T=T+B(I); Next
810 Return T; FUNC End
```

The problem in this example is that the same loop index variable (I) is used by two different but nested loops, a condition which is easily hidden by the function itself. You must ensure that this kind of situation never happens in your programs by restricting potentially harmful variable accesses. Two methods are available for controlling variable access:

- Every time you use a variable, find out how and where it is used elsewhere in the program and make appropriate changes as needed, and
- Define temporary variables used in subroutines as **LOCAL** variables with the **LOCAL** statement.

Data stored in variables must be preserved for the term of its usefulness. Variables which contain long-term data must be protected from unintentional use, especially in large programs. You may safely obtain temporary storage by using available local function argument variables (if any), or by temporarily creating new variables with the **LOCAL** statement (Chapter 8, Section 1). Both methods can be employed, but using **LOCAL** variables is preferred.

The second type of side-effect is rather obscure but you should be aware of it. **READ** or **WRITE** statements containing user function calls which in turn perform their own **READS** or **WRITES** to the same file, can upset the current file position causing the original **READ** or **WRITE** to access the wrong file position. For example if you directly **WRITE** the result of a function that itself accesses the same file, the data will be written at the file position left by the function call rather than the position specified by the original **WRITE** statement. Or suppose that a **READ** statement includes a user function that **CLOSES** the file in its procedure. Such an operation would produce highly unpredictable results.

Awareness of this side-effect is essential to prevent it from occurring. There are two ways to avoid this problem. At the *call* level, you can always store the function result in a variable for subsequent use in the **READ** or **WRITE** statements. At the function level, you could save the file positions of all files accessed by the function and restore them just before returning back to the caller. This is an excellent solution because it hides the details within the function and the caller does not have to know anything about it.

Section 4: Communicating with Subroutines

Useful subroutines take some input data, do something with it, and produce some effect or result. Inputs must be accessible to the subroutine and the set of output results produced need to be posted in some way useful to its caller. Except for the **RETURN** <*expression*> of function subroutines, all data communication between subroutines and their application context is through variables. The various techniques to do this are presented below.

Global Variables

In MegaBasic, like all **BASICS**, all variables may be accessed throughout the program. This type of access is called *global* access and such variables are referred to as *global variables*. Variables that contain the results of one statement are therefore accessible to any other statement that chooses to use them. The output of one subroutine becomes the input of another. This kind of data communication has no limitations other than the ability of the programmer to manage the relationships of the variables involved.

However, as more variables come into play, the task of managing variables can become more difficult. Each use of a variable must be checked out by a thorough examination of its uses throughout the program. The searching capability of the **LIST** command (Chapter 2, Section 2) and the **NAMES** command (Chapter 2, Section 3) is useful for this. The **XREF** command (Chapter 2, Section 5) generates a cross reference listing of the various identifiers showing where they are referenced. Each global variable should be documented and its purpose restricted to a single use.

Meaningful names assigned to variables, lines, functions and procedures are an important means of managing your program. Depending on the application, a naming standard can be useful to indicate the purpose of each variable by characteristics embedded in its name. For example, by naming truly global variables with names of six or more characters and restricting the names of variables used for temporary storage to five or less characters, it will be more difficult to accidentally use a global variable for temporary purposes. In the long run, such self-imposed standards can save you a significant amount of debugging time.

The problems associated with global variables are an unavoidable result of the unrestricted access to variables, a standard feature of **BASIC** itself. Without restrictions on global variable access, programs larger than 40 or 50k bytes become difficult to develop and maintain. Fortunately, MegaBasic has some automatic mechanisms to handle these sorts of problems: *argument lists* and their related *local variables*.

Argument Lists

As seen above, communicating data through variables has some rather severe shortcomings. Instead of having to explicitly assign values to the variables required by the subroutine, MegaBasic supports a mechanism for automatically assigning them, called an argument list. When invoking a subroutine that uses an argument list, the data you wish to communicate is listed after the name of the subroutine without necessarily being stored in any particular variables. When the subroutine begins execution, this input data becomes available as a list of variables corresponding to each input argument.

The following example illustrates how this works. Suppose that you need to compute the nearest multiple of one number to another. This simple function would be defined something like this:

```
Def func NEARMULT(V,MULT) = round(VlMULT)*MULT
```

The argument list of this function is defined with two numeric variables, V and MULT, which are called *the formal arguments* of the function. The actual work of the function is performed by the expression to the right of the equals sign (=), which uses the two arguments in a computation that determines the result desired. When invoked, **NEARMULT** uses any pair of actual arguments that you wish to submit to it for calculation and they do not even have to be stored in variables, for example:

```
NEARMULT(XIY,Z–2)
```

MegaBasic sets V to the result of quotient X / Y, sets **MULT** to the difference Z–2, and then begins function execution. These two argument expressions are called the *actual arguments* of the function. Even though **NEARMULT** only *knows* how to operate with two fixed variables (i.e., V and **MULT**), it can accept any arbitrary pair of input numbers.

Subroutine arguments can be strings as well as numbers and when strings appear as actual arguments, their corresponding formal arguments in the argument list definition must be string variables. In other words, the data types of the actual and formal arguments must agree. MegaBasic assumes that the definition is correct when they don't agree, and reports an error in the reference to the subroutine. For example, if you supply a string expression as one of the arguments to the function above, MegaBasic reports a *Data Type Error.*

You can independently specify formal argument variables with several optional capabilities. Different argument passing modes are supported to let you pass actual arguments as values, as variables or as pointers. This is covered in the next few pages. The number of arguments specified in subroutine references normally matches the number defined for the subroutine. However, you can also specify optional arguments along with default values that are *passed* in the event that the corresponding actual argument is omitted. This powerful feature is described later on in this chapter.

Argument Passing Modes

MegaBasic supports a number of different ways that arguments can be passed to subroutines called argument passing modes. Each mode has certain advantages over the others in specific applications. They are specified independently for each formal argument by the presence or absence of a single special character in front of the formal argument variable in the subroutine **DEF** statement. Much of the remainder of this chapter is devoted to describing these argument modes, which are summarized below:

Value Arguments	Unless you specify otherwise, arguments are passed by *value*. In other words, only the value is passed to a subroutine. Changing the value of such an argument has no effect on the subroutine caller.
Variable Arguments	An at-sign (@) indicates an argument that lets you pass an actual variable to the subroutine. Such variables may contain input values, but they may be altered by the subroutine in order to pass output data back as well. The actual arguments specified can only be names of variables, i.e., no expressions, arrays subscripts or string indexing.
Copied Arguments	A percent sign (%) indicates an argument that simply receives an input value for subsequent use by the subroutine. This is similar to a *value argument*, but the formal argument variable is not restored upon returning from the subroutine.
Pointer Arguments	An asterisk (*) indicates an argument that extracts the *pointer* to the actual argument and passes that value through the formal argument variable instead. The topic of pointers is discussed in Chapter 5, Section 4.

Value arguments are common in most applications because of their generality and simplicity. The other argument modes have very important uses and can simplify and speed up certain types of processing. The concepts and motivation for each mode will now be discussed in full detail.

Value Arguments

Formal arguments defined in subroutines are nothing more than simple variables, which may actually have other uses in the program (outside the subroutine). When you pass data to a procedure or function through the variables listed in the subroutine definition, the following steps are performed internally:

- All of the actual data values to be passed are completely evaluated, as each argument may be expressed by a general expression.
- The current contents of the variables listed in the argument list *definition* are saved so that upon **RETURN**, they can be restored to their original values as if nothing happened.
- The previously evaluated *actual data values* to be passed are then copied into the corresponding argument variables for subsequent use within the function. Numeric arguments are converted to the same type as the argument variables receiving them.

MegaBasic preserves the contents of all formal argument variables during the entire execution life of a subroutine. In other words, the contents of each formal argument variable defined by a subroutine is the same before and after invoking that subroutine, regardless of what information is passed through it. Once a subroutine **RETURNS**, its formal argument variable values disappear and are forever lost. Because they effectively exist only within the context of subroutine execution, such variables are called *local* variables. The following procedure illustrates an example of this:

```
10 LIMIT = 999999,; Print LIMIT; COUNT_TO 10;
Print LIMIT; END
20 Def proc COUNT_TO LIMIT
30 For LIMIT = 1 to LIMIT; Print LIMIT,; Next
40 Return; Proc end
```

When **RUN**, this program produces the following output:

```
99999912 3 4 5 6 7 8 910 999999
```

Although **LIMIT** is altered within procedure **COUNT_TO**, its value on the outside of the procedure is unaffected. The **LIMIT** variable inside the procedure is, for all intents and purposes, totally separate and distinct from its namesake outside of the procedure. Other variables used within subroutines that are not formal argument variables will be treated as *global* variables, and their values persist after the subroutine terminates.

The fact that you can always temporarily store information in local variables without ever worrying about overwriting *global* information *is* an extremely useful and simplifying mechanism. It means that you can develop subroutines which can be used in any context without fear that they may alter surrounding variables in unplanned ways. Local variables are so important that MegaBasic includes a **LOCAL** statement (Chapter 8, Section 1) which effectively creates local variables for whatever temporary (local) application you desire.

The following program uses a **LOCAL** statement to implement the previous example using a **GOSUB** instead of a procedure:

```
10 COUNT = 999999; Print COUNT,;
GOSUB 20; Print COUNT;
End
20 Local COUNT;
For COUNT = 1 to 10; Print COUNT,;
Next; Return
```

When RUN, this program produces the following output:

```
9999991 2 3 4 5 6 7 8 910 999999
```

Even though the **COUNT** variable is modified within the **GOSUB**, it remains unchanged from the view of the routine which called the **GOSUB**. Use local variables to isolate the inner workings of your subroutines, confining their effects on the *outside world* to only well-defined and documented sets of intended output result variables.

String variables may be passed by *value* as localized variables but remember that their prior string value is saved in internal scratchpad memory until the **RETURN** statement is executed. This can quickly use up the 55k bytes (approx.) that they possesses if used indiscriminately. Furthermore, the formal variable argument itself must be of sufficient capacity to contain the actual input argument string. A *Length Error* results if the input argument exceeds the size of the formal variable.

String variables used solely for local purposes should be assigned an initial value of a null string ( ) during program start-up (initialization). This will prevent the unnecessary saving of a full variable of spaces ( ) onto the scratchpad stack when the string is declared local or used as a local parameter in an argument list.

Variable Arguments (@)

The standard method of passing input data to subroutines via local variables does not address the other side of the problem: subroutine output. Local variables only communicate input values and by definition cannot be employed to pass computed results back to the *user* of the subroutine. Furthermore, it is neither possible nor feasible to communicate array variables as local variables, but in many situations array communication is nonetheless highly desirable. For example, if you develop a subroutine to sort the contents of an array, you most certainly would like to apply that subroutine to any array without having to reprogram it for each.

Variable arguments are specified by placing an at-sign (@) in front of the corresponding formal argument variable in the subroutine **DEF** statement. This form of argument not only passes the contents of a variable, it passes the variable itself. MegaBasic temporarily changes the identity of the formal argument variable so that it *becomes* the actual argument variable itself. This change of identity remains in effect until the subroutine terminates. The simple example below illustrates these concepts with a procedure that increments numeric variables:

```
10 X=6;
   While X<16; INCREMENT X; Print X,; Next;
   End
20 Def proc INCREMENT @ VBL
30 VBL = VBL+1; Return 40 Proc end
```

Running this program produces the following output:

```
7 8 91011 1213141516
```

Although **INCREMENT** only deals with a variable named **VBL**, it is called with a different variable named X. Inside this procedure, **VBL** becomes X and any alteration to **VBL** is really an alteration to X. As shown above, the special nature of this argument is indicated in the argument definition of **VBL** by placing an at-sign (@) in front of it.

This method of communicating variables is called *passing by address* in many languages, due to the internal mechanism employed to implement it. Passing by address is a powerful feature of **FORTRAN**, **PASCAL**, C, PL/1, **ALGOL**, and many other compiled languages. There are a number of ground rules that must be obeyed in order to use this feature, which follow below:

- An at-sign (@) must immediately precede each formal argument variable in the **DEF** statement which is to be passed by address. The at-signs are never to appear in the calls to this subroutine.
- Actual arguments to be passed by address must appear as a variable name only: no subscripting, no indexing, no arithmetic expressions, no parentheses. This means that array elements and indexed string variables cannot be passed by address.
- The formal variable in the definition and the actual variable passed in the calling sequence must agree in type, i.e., both must be string variables or both must be numeric variables.

- Either integer or real variables may be passed through a formal argument variable of either type. The formal argument variable does not impose its numeric type onto the argument when passing by variable, as is done when passing by value.
- Prior dimensions of the formal argument variable in the **DEF** statement do not control what may be passed and are, in fact, irrelevant. This is because the entire variable and its current size attributes are passed along with the rest of the variable. However all references to the passed variable within the body of the subroutine must agree with its current data type definition. The **DIM()** function (i.e., not the statement) provides information about array variables which can be useful in implementing subroutines that process arbitrary arrays.
- Nothing but variables may be passed in the actual arguments: no functions, no procedures and no expressions, nothing except single unadorned variable names.

Passing variables by address is important for two reasons. First, you are able to define subroutines which can process any variable which is passed to it, whereas with passing by value you do not have any access to the variable, only its value. This also means that results of a subroutine may be communicated back to the calling program via the variable. Second, no matter how big the variable is, you can pass it as an argument in only a fraction of a millisecond (impossible when passing a large string by value).

The **DIM()** function (Chapter 9, Section 5) allows subroutines to determine for themselves the dimensions of variables passed to them. DIM(X) returns the number of dimensions currently defined for variable X; zero is returned if X is scalar or undefined. DIM(X,N) returns the upper limit of dimension N of array X. An error results if X is not an array or N is outside the range 1 to **DIM(X)**.

Copied Arguments (%)

In some ***BASICS*** (e.g., North Star **BASIC** string arguments), the *passing by value* mode of input argument communication does not restore the prior contents of the argument variable on **RETURN**, i.e., local variables are not used. The net effect of using this mode is identical to storing input values into a set of variables with assignment statements prior to calling a **GOSUB** that uses them. When you pass data by copying, the following steps are performed internally:

- All of the actual data values to be passed are completely evaluated (as each argument may be expressed by a general expression).
- The previously evaluated data is then copied into the corresponding argument variables for subsequent use within the subroutine. Numeric arguments are converted to the same type as the argument variables receiving them.

This passes the input data but loses the prior contents of the formal argument variable. An occasional program designed with this mode in mind may rely on the side-effect values left in those variables (a questionable practice).

To invoke this type of argument passing, a percent sign (%) may be placed in front of any formal argument variable to indicate this mode is to be used, similar to the way that an at-sign (@) is used to indicate *passing of variables*. This mode is simply the *passing by value* mode without the local argument variable.

Two reasons for using this mode should be mentioned. First, if the formal argument variable contains a very large string, the execution time and internal memory space

required to *localize* that variable may be undesirable. *Copied* arguments are not localized and therefore execute faster and require no internal scratchpad storage. Secondly, variables in external packages (see Chapter 10) are local unless explicitly declared as **SHARED**. Such variables, if not used for any other purposes within the external package itself, can be used as *copied* arguments while retaining the properties of local variables.

Pointer Arguments (*)

Chapter 5, Section 4 describes a MegaBasic capability called *pointer variables. You* should read that section for complete information about pointer variables and pointer arguments. The discussion below is intended to introduce you to pointer concepts and their use within the context of subroutine argument lists.

A pointer is a special number that represents a variable, function, procedure, or line label. Each named MegaBasic *object* has an associated pointer. If you have a pointer, you can use it to access the *object* to which it refers without having to use its name. Since such numbers can be stored in variables, your program can then use variables to refer to other variables. MegaBasic pointer variables are almost identical with the pointer facilities implemented in the *C programming language.*

By placing an *asterisk (*)* in front of a formal argument variable, you define that argument as a *pointer argument*. When an actual argument is passed though such an argument, MegaBasic *extracts* its pointer value and passes it (by *value)* to the subroutine. In other words, the pointer is passed instead of the actual argument itself. *The formal* pointer argument variable must be an integer because pointers themselves are 32-bit integer values.

Specifying a real or string variable in as a formal pointer argument is reported as a *Pointer Variable Error.* The actual argument passed through a pointer argument can be any named entity, including: scalar string, integer or real variables, any array name or array element reference, or the name of a procedure, function or line label. Pointer arguments therefore let you pass *objects* rather than expressions. This is similar to variable arguments, but pointers are much more general (though also more difficult to use).

In order to use the pointer argument within the subroutine, you need to know how to access the *object* it points to. This subject is, again, covered in Chapter 5, Section 4. The purpose of pointer arguments is to provide an automatic pointer extraction mechanism as part of the argument list mechanism in order to *hide* this implementation detail from the caller. In addition pointer arguments let you pass, among other things, subroutine names and array elements as parameters to subroutines.

Optional Parameters and Default Values

You can define argument lists of user-defined functions and procedures to permit the omission of parameters from the calling argument list. In other words, such functions and procedures can be called with a different number of arguments on different references. This capability is useful for constructing MegaBasic subroutines with special purpose arguments that only need to be specified in the references that actually need them. Proper application of this capability can simplify your programs, as well as make them run faster and be more readable and maintainable. The following function **DEF** statement illustrates how to create a function with 1 or 2 arguments:

```
Def real func LOGARITHM(V,BASE=10) = log(V)llog(BASE)
```

This function computes the logarithm of V to any **BASE** specified or to base 10 if not specified. Notice the **BASE**=10 expression in the argument list. This syntax specifies the default value to use for parameter **BASE** if no second argument is specified. Any argument defined in this manner may be subsequently omitted from the argument list when calling the subroutine. For example **LOGARITHM**(2) returns Log 2 base 10, **LOGARITHM**(3,12) returns log 3 base 12. With this general idea in mind, the additional rules for defining and using default parameters are listed below:

- Parameters can only be omitted from right to left, i.e., you cannot omit parameters out of the middle, unless you also omit all parameters to the right of them. Be sure to define default values for every parameter allowed to be omitted.
- When all parameters are omitted from function references, you must also omit the parentheses that surround the argument list (i.e., don't leave an empty parenthesis *shell).*
- Default values, when specified, immediately follow the formal parameter variables in the subroutine argument list definition with an equals sign (=) separator. A default value may only be a constant or a simple variable (no subscripts or indexing) and must match the data type of its corresponding parameter variable. General expressions as default values are thus not supported.
- When a simple variable is the specified default value, its effective value is its contents *prior to the evaluation* of any earlier parameters. This is because MegaBasic evaluates the *entire* calling argument list *before* binding the values to the formal parameter variable list in the definition.
- Default values may also be assigned to parameter variables that are passed as variables (preceded by @) or passed by copying (preceded by %). The default value of an at-sign (@) variable must be a simple variable, never a constant. Default values for percent sign variables (%) are identical to default values for regular (local) parameters.

Generally, it is convenient to use default values that can be identified by their values as default values. For example, a null string ( ) is a useful default value for a string parameter that would never be a null string. When the subroutine tests this parameter and discovers that it is null, a different course of action for the omitted parameter can be taken.

Because optional parameters may only be omitted from right to left, you should carefully design your argument list definition so that the optional parameters are ordered from most often required to least often required (left to right). In this way, the references to these subroutines will tend to use the minimum number of calling arguments throughout your program.

Default parameters are passed to the subroutine more quickly than specified arguments. However you should remember that no matter how many arguments you actually supply when calling your subroutine, MegaBasic always evaluates all of the arguments in the defined formal argument list. Therefore unnecessary optional parameters that are never needed in practice should be avoided for best performance. Also, the addition of default values to subroutines definitions does not in any way slow down subroutine calls that include all the arguments.

Default values that change with program conditions are implemented by specifying the default values with simple variables. For example, suppose that whenever you omit a

certain parameter you want it to default to the last value actually specified for it. If this parameter is the variable **LAST,** you might implement this type of default using the parameter definition: **%LAST=LAST**. The percent causes **LAST** to retain its value after the subroutine terminates. Hence if you omit this parameter, it simply takes on its current value, i.e., the value it had upon return from the subroutine (assuming nothing else altered it).

One of the really important applications of optional parameters is the extension of existing MegaBasic packages in an *upward compatible* manner. Suppose that you have a package that is used by many people, for instance, a data base function library. As time goes by, you discover that some of the functions it contains can be modified to cover a wider range of applications, but you are prevented from realizing these expanded capabilities because a change in their argument structure would make the package incompatible with existing programs using it. You can probably see by now that appropriately defined optional parameters are the key to expanding such functions. All existing programs using the package can execute as before, while at the same time new or upgraded software can take advantage of the extended capabilities they provide by using the additional parameters.

Open-ended Argument Lists

A totally different argument list structure is also supported to let you define functions and procedures that can be called with any number of arguments from one call to the next. Furthermore, the argument type of each argument (string or numeric) may also vary on different calls. This capability is useful for defining subroutines that can more easily adapt to the context from which they are called. MegaBasic has several built-in functions that already operate in this manner, such as the MAX( ) and MIN( ) functions and the **PRINT** statement. To show you how to define such functions and procedures, we have defined below a user-defined version of the MAX( ) function called **MAXIMUM**:

```
100 Rem ***Return the maximum value from a list of values
110 Def func MAXIMUM(...); local X,Y; Argument X
120 While argument; Argument Y; X = max(X,Y); Next
130 Return X; Func End
```

This function illustrates all the essential features of the open-ended argument list capability. Notice that the formal argument list definition in the **DEF FUNC** statement consists of three dots (...) instead of variable names. This simply indicates that the function will be called with an open-ended arguments list and that all argument handling will be done within the body of the function.

Since no formal argument variables are provided in such functions, a special statement to obtain the actual arguments from the call has been added, called **ARGUMENT**. ARGUMENT works very much like the **READ** statement for reading values from **DATA** statements into variables, except that **ARGUMENT** reads the calling argument list into variables. For example, ARGUMENT A,B,C reads the next three arguments from the calling argument list into numeric variables A, B and C; ARGUMENT A$,X reads a string argument into A$ and a numeric argument into X. In line 110 of the example above, an **ARGUMENT** statement is used the read the first value of the calling argument list into variable X.

The **ARGUMENT** statement only works directly inside the subroutine and cannot be used from within other lower-level subroutines. In other words, you cannot have open-ended arguments *read* by calling other subroutines.

An open-ended argument list is, by definition, a list of some unspecified number of value expressions, separated with commas. Your program needs some way to determine when the list has been exhausted, so that it can stop reading arguments. You could use **ERRSET** to trap the error that results from trying to read past the last argument with the **ARGUMENT** statement. However the **ARGUMENT** statement can be used as a function (no arguments) that returns *True (1)* if there is one or more unread arguments remaining on the calling argument list, and *False (0)* if no more arguments. Line 120 in the example tests the **ARGUMENT** function in the **WHILE** condition to determine the presence of additional unread arguments and if true (1), executes the loop to evaluate the next one and process it.

MegaBasic permits your function or procedure to return without reading all the arguments that it was called with. Such unread arguments are never evaluated, so no computational effort is wasted on them. A good example of this is the **AMONG()** function, which returns the position of a number in a list of numbers, or zero if it matches none of them:

```
100 Rem *** Return the position (1,2,..,n) that X match a value
in a list
110 Def integer func AMONG(...); Local X
120 Argument X; I = 1
130 While argument; argument Y;
   If X–Ythen return l; 1+~1; Next
140 Return 0; func end
```

This function returns a result as soon as the first argument is found to equal any one of the other arguments, or when the list is exhausted. For example, **AMONG**(323, 567,12, 87, 323, 999) = 4, **AMONG**(–34, 8256,7) = 0, etc.

Up to now, we have examined open-ended argument lists of numbers for functions. In the example below, we have defined a procedure that prints each of the arguments listed, regardless of their type (string or numeric). Notice that error trapping is needed to do this because a *Type Error will* occur if a string (numeric) argument is read into a numeric (string) variable. By trapping the error, the argument can be re-read using a different variable, allowing the process to continue.

```
100 Rem *** Procedure that displays any sequence of arguments
110 Def proc PRINT_LIST...; local V, V$
120 While argument
130 Errset 140; Argument V; Print V; Goto Next
140 Errset 150; Argument V$; Print V$; Goto next
150 Errset #99, “Bad Arguments”
160 Next; Return; Proc end
```

Section 5: Recursive Programming

You are free to employ the subroutine you are defining within its own definition. Known as recursion, such subroutines must ultimately reduce down to a result without reference to themselves in order to terminate in a finite amount of time. Otherwise they continue to invoke themselves until all the subroutine control space in the machine is consumed, ending in a *Scratchpad Full Error.* Recursive subroutines often split a problem into several smaller but similar problems, then call themselves to solve each of these.

A simple application of recursive programming is the process of determining the greatest common denominator of two integers (**GCD** for short). It can be shown mathematically that **GCD**(n,m) = **GCD**(m mod n, n), where 0<n<m. This equation defines **GCD** in terms of itself, making it a recursive definition. The following program shows how it is implemented in MegaBasic:

```
10 Rem -- Recursive Function for the
20 Rem -- Greatest Common Denominator
30 Def func GCD(VAL1 ,VAL2)
40 If VAL1 > VAL2 then Swap VAL1 ,VAL2
50 If VAL1 then Return GCD(VAL2 mod VAL1 ,VAL1)
60 Return VAL2; Func End
```

Line 30 names the function and defines its arguments. Line 40 ensures that **VAL1** is not greater than **VAL2**, because we will be taking the remainder of **VAL2/VAL1** using the **MOD** operator. This is justified because the **GCD** of two numbers is the same regardless of their order. Line 50 tests **VAL1** to see if it is non-zero and if so, returns the result in terms of another **GCD** evaluation, i.e., it calls itself. If **VAL1**=0 then the **GCD** must be the value of **VAL2**, i.e., the **GCD** of zero and any other value is that value.

One of the important aspects of recursive programming illustrated above, is the essential property that the problem is ultimately reduced down to a result which is **NOT** recursively defined. The **GCD** function repeatedly reduces the original numbers supplied into smaller numbers which have the same **GCD**. When one of these numbers eventually becomes zero, the final result is known and does not required further recursive processing.

The above example also illustrates how hard it is to guarantee the a recursive subroutine will eventually terminate in all cases. Upon careful examination of it, you may notice that if **VAL1** or **VAL2** is negative (less than zero), then **GCD** will never reach a point of termination and ultimately uses up all remaining scratchpad space and terminates the program by force (i.e., by an error).

Many recursive programs can actually be reformulated as iterative procedures (i.e., implemented with loops) rather than recursively. Iterative implementation usually executes a little faster and consumes less memory. This being the case, why do we bother with recursive programming? The answer is that certain problems are more naturally defined recursively and their solution can be much more obvious and simply specified in a recursive manner.

The key to recursive programming is in recognizing that certain problems can be *naturally* defined in terms of themselves and to avoid recursive solutions to those problems that are not. As with all tools, selecting the one most suited to the problem at

hand leads to the quickest solution. Recursive programming is a *surgeons scalpel,* not to be confused with a *construction pile-driver.*

Another important consideration in constructing recursive subroutines is knowing when to use local variables and when to use static variables. Static variables are those that keep their contents after the subroutine returns, and local variables are those whose contents *live* only as long as the current subroutine invocation.

Variables used to store temporary or intermediate resulting within the recursive subroutine should usually be local variables. This is because you do not want a recursive re-entry into the same subroutine to overwrite the contents of such variables before they are used. Temporary variables having no further use before a recursive re-entry do not have to be local. Static variables can be used to store data that is never modified (i.e., read-only), or for input/output data that the subroutine modifies along the way.

A subroutine does not have to call itself directly to be recursive, it is also recursive if it invokes a different subroutine that, at some point, invokes the original subroutine again. This is known as indirect recursion and is not really any different from direct recursion except that it is less obvious. It sometime arises accidentally in larger programs when the programmer calls a subroutine that eventually get back to the same point in the program before returning. This can be a difficult situation to diagnose and the TRACE RET command (Chapter 2, Section 4) can be useful for this.

Chapter 9

MegaBasic Built-in Function Library

This section provides a complete description of all the built-in functions in MegaBasic. These should be studied for utility in your particular applications since there may be several that already do what you have in mind and they run many times (even hundreds of times) faster than similar procedures implemented in more primitive **BASIC** statements. This function set has been carefully selected to cover the widest variety of applications with a minimal number of separate function entities (when applied in combination). Chapter 3, Section 7 covers how to apply most of these functions to entire vectors of numbers (instead of just scalars). For easy referral, the built-in functions have been grouped into the following subsections:

Arithmetic Functions	Simple arithmetic conversions and transformations including: rounding, truncation, comparison, absolute values, etc.
Mathematical Functions	Transcendental functions and other transformations for mathematical and scientific applications.
Character and Bit Function	Functions for combining, searching and transforming character and bit strings.
File and Device I/O Functions	Functions providing information to support access to files and serial I/O devices.
Utility and System Interface	Functions for directly accessing memory, hardware ports and various MegaBasic internal parameters.

MegaBasic supports a *relaxed* parentheses convention: parentheses are *optional* around the argument lists of any numeric function having a *single numeric argument*. For example, Log Sin X is the same thing as Log(Sin(X)). *You may not omit* parentheses from around any argument list for string functions or from those which contain (or may contain) more than one argument in the list. When an expression follows a function without parentheses, only the first term of the expression is recognized as the argument. For example the expression Sqrt X * Y is evaluated as (Sqrt X) * Y, not as Sqrt(X * Y). There is a small speed improvement for reducing parentheses, but this feature is really supported to help simplify complex arithmetic expressions.

Unless otherwise noted, parameters to functions can be arbitrarily complex expressions. Numeric arguments can evaluate to either an integer or real data type. MegaBasic always converts numeric arguments to the type (integer or real) internally required by the particular function being invoked, regardless of the type actually provided. You can improve the performance of your program by specifying numeric arguments in the type (integer or real) most *natural* to each function or statement programmed. The description of each function provides the details necessary to take advantage of such numeric type considerations. A syntax summary of all built-in functions now follows:

ABS(<*numeric exprn*>)
ACOS(<*numeric exprn*>)
ASC(<*string expression*>)
ARGUMENT
ASIN(<*numeric exprn*>)
ATN(<*numeric exprn*>)
BIT(<*string variable*> *[<bit position range>]*)
CARD(<*string*> *[<bit range>]*)
CEIL(<*numeric exprn*>)
CHR$(<*ASCII code*> *[,<last ASCII code>]*)
CHRSEQ$(<*range*>,<*range*>,...)
COLLAT$(< *numeric exprn*>)
COS(<*numeric exprn*>)
DATE$
DIM(<*variable name*> *[, <dimension number>]*)
DIR$
DIR$(<*prior file name string*>)
EDIT$
ELAPSE *[* **(<*mode*>)** *]*
ENVIR$(<*name or sequence number*>)
ERRDEV
ERRLINE
ERRMSG$
ERRPKG$
ERRTYP
EXAM(<*memory address*>)
EXP(<*numeric exprn*>)
FILE(<*file name string exprn*>)
FILEDATE$(<*open file number*>)
FILECTRL(<*open file number*>)
FILEPOS(<*open file number*>)
FILESIZE(<*open file number*>)
FILETIME$(<open file number> *)*
FIND*(<mode><vbl$><cmp><expr$>,<step>,<count>, <step>, <count)*
FRAC*(<* ***numeric exprn*>)**
FREE(<*numeric exprn*>)
INCHR$*(<device> , <length>, <break set, <echo flag>, <time out***)**
INDEX
INP$(< ***port number*>)**
INP*(<port number>)*
INPUT(<*device number*>)
INT*(<* ***numeric exprn*>)**
INTEGER(<*numeric exprn*>)
INTERRUPT(<*selector*> *[,<interrupt number>*])
IOCTL$(< *channel number*> *[,<output control string*>])
IOCTL(<*open channel number*>)

LEN(<*string expression*>)
LINE(<*device number*>)
LN(<*numeric exprn*>)
LOG(<*numeric exprn*>)
MATCH(<*search string*>,<*match string*>,<*starting location*>,<*count*>)
MAX$(<*string1*>, <*string2*>,... <*stringN*>)
MAX(<*list of numeric expressions*>)
MIN$(<*string1*>, <*string2*>,... <*stringN*>)
MIN(<*list of numeric expressions*>)
MOD(<*numeric exprn*>,<*modulus exprn*>)
OPEN$(<*file number*>)
ORD(<*string*> [<*bit range*>])
OUTPUT(<*device number*>)
PARAM(<*parameter number*>)
PI
POLY(<*real value*>, <*real array name*>, <*polyn degree*>)
POS(< *device number*>)
REAL(<*numeric exprn*>)
RESEQ$(<*target string*>,< *stepsize*>)
REV$(<*string expression*>)
RND(<*numeric exprn*>)
ROTAT$(<*string exprn*>,<*rotation distance*>)
ROUND(<*numeric exprn*> [,<*significant digits*>])
SEG(<*variable name*>)
SGN([<*sign exprn*>~ <*value exprn*>)
SIN(<*numeric exprn*>)
SPACE(<*drive number*>)
SQRT(<*numeric exprn*>)
STR$(<*numeric expression*> [,<*format string*>])
STRUCT(<*field*> [,<*selector*>])
SUBDIR$(<*prior subdirectory name string*>)
SUM(<*vector expression*>)
TAN(<*numeric exprn*>)
TIME$
***TRAN$(*<*string*>,<*original chars*>,<*translated chars*>)**
TRAN$(<*string*>:<*translation map vbl$*>)
TRIM$(<*string expression*>)
TRUNC(<*numeric exprn*>)
VAL(<*string expression*>)
[<*variable name*>]

Section 1: Arithmetic Functions

The functions described in this section perform simple arithmetic operations on numbers such as rounding, truncating, absolute values, remaindering, random values, sign transfer, etc. Following the summary below, each function will be described in detail.

Int(X)	Returns greatest whole number no greater than X.
Ceil(X)	Returns lowest whole number no less than X.
Trunc(X)	Returns the integer portion of X.
Frac(X)	Returns the fractional portion or X.
Round(X)	Returns the nearest whole number to X.
Round(X,P)	Returns X reduced to P significant digits.
Mod(X,Y)	Returns the positive remainder of X / Y.
Abs(X)	Returns the positive value of X.
Sgn(X)	Returns the sign of X (–1, 0 or +1).
Sgn(X,Y)	Returns Y with the sign of X.
Min(X,Y,..)	Returns the minimum value among a list of values.
Max(X,Y,..)	Returns the maximum value among a list of values.
Rnd(X)	Returns uniformly distributed random values.
Sum(V)	Returns the sum of the elements of vector V.

INT(< *numeric exprn*>)

Returns the greatest whole number less than or equal to the numeric expression. For example: **`INT(3.4) = 3, INT(.2) = o, INT(–7) = –7, INT(–1.3) = –2`**. This function is only useful with real numbers, since **`INT()`** of an integer is the same value. **`INT()`** always returns a result of the same numeric type (integer or real) as the argument supplied. **`INT`** has sometimes been known in other languages under the name **`FLOOR`**.

A common use of **`INT()`** is finding the greatest multiple of a number that is not greater than another. The expression **`INT(X/Y)* Y`** returns such a multiple of Y with respect to X. This expression is so frequently encountered that it can also be expressed in MegaBasic as **`X INT Y`**, which is computed much more efficiently. In this context, `INT` is being used as an operator (like + and –), rather than as a function.

CEIL(< *numeric exprn*>)

Returns the smallest whole number greater than or equal to the numeric expression. For example: **`CEIL(3.4) = 4, CEIL(–2.13) = –2, CEIL(1) =1.`** This function is only useful with real numbers, since **`CEIL()`** of an integer is the same value. **`CEIL()`** always returns a result of the same numeric type (integer or real) as the argument supplied.

One common use of **CEIL()** is finding the smallest multiple of a number that is not less than another. The expression **CEIL(X/Y)*Y** returns such a multiple of Y with respect to X. This expression is useful enough that it can be expressed in MegaBasic as **x CEIL Y**, and computed much more efficiently (especially with integer operands). Using this syntax, **CEIL** is being used as an operator (like + and –), rather than as a function.

TRUNC*(<numeric exprn>)*

Returns numeric expression value with any fractional part removed. For negative numbers this is equivalent to the **CEIL()** function and for positive numbers this is equivalent to the **INT()** function (described above). For example: **TRUNC(3.4) = 3/ TRUNC(–2.71) = –2, TRUNC(7) = 7**.

This function is only useful with real numbers, since **TRUNC()** of an integer is the same value. **TRUNC()** always returns a result of the same numeric type (integer or real) as the argument supplied. Like the **CEIL()** and **INT()** functions, the expression **TRUNC(X/Y)*Y** can be expressed as **x TRUNC Y**, which is computed much faster with integer operands.

ROUND *(<numeric exprn>)*

Rounds the number specified to the nearest whole number. For example: **ROUND(3.4) = 3, ROUND(3.5) = 4, ROUND(6.8) = –7, ROUND(–2.5) = –2**. This function is only useful with real numbers, since **ROUND()** of an integer is the same value. **ROUND()** always returns a result of the same numeric type (integer or real) as the argument supplied.

In many applications it is desirable to round a number to the nearest tens-place or hundreds-place or some other multiple of 10 (or other moduIus). **ROUND()** can easily be generalized to round in this manner using the expression **ROUND(X/Y)*Y**. For both efficiency and convenience, this computation may be expressed in MegaBasic as **x ROUND Y. For example: 36498 ROUND 10 = 36500, 57.382 ROUND .1 = 57.4**.

ROUND(*<numeric exprn>,<significant digits>)*

Returns the numeric expression value rounded to the number of significant digits specified by the second argument expression, which must be 1 or greater. For example: **ROUND(1.6483,3) = 1.65, ROUND(S72096,3) = s72000**. This function is especially useful in **PRINT** statements for limiting the significant figures of values that can span a very large numeric range. Such numbers would normally be formatted using exponential notation (E-format), which is usually only appropriate in scientific / engineering applications and undesirable for business applications.

Another important application for this function is the case of comparing two numbers to see if they are approximately-equal. Such a comparison is frequently required when two floating point calculation results are compared for equality and minor differences in the last few trailing digits are considered insignificant. For example, we can compare X and Y out to the first 5 significant figures with the expression: **ROUND(X,5) = ROUND(Y,5)**.

MOD*(< numeric exprn>,<modulus exprn>)*

Returns the smallest non-negative number which when subtracted from the first argument produces an exact multiple of the second argument. For example:

```
MOD(34,17) = 0, MOD(13,5) = 3,
MOD(–13,5) = 2, MOD(X,0) = 0.
```

MOD() accepts either real arguments or integer arguments. If they are not both of the same numeric type then one argument is converted into the type of the other before calculating the modulo. **MOD()** returns a result in the same numeric type as the type of its arguments (after making them the same). **MOD()** may also be used as an operator: **X MOD Y =MOD(X,Y). MOD()** executes faster with integer operands than with real operands.

MOD() has zero precision loss for all arguments in both integer and floating point modes. This is important because a major use of **MOD()** is in range reduction applications. **MOD(X,Y)** requires time proportional to the magnitude difference between X and Y, and it is extremely fast for magnitude differences under 10^{10} because no multiplies and divides are used. It is somewhat slower when X and Y have an extremely large magnitude difference (e.g., 10^{50} **MOD** 10^{-50}) because it uses the method of successive scaled subtraction. Although **MOD()** introduces no error into the result, a meaningful result requires absolute accuracy in the original X and Y values, especially for large magnitude differences.

FRAC(*<numeric exprn>*)

Returns the difference between the number specified and the next lower integer, i.e., the fractional portion of a number. This function is only useful with real numbers, since **FRAC()** of an integer is always zero. **FRAC()** always returns a result of the same numeric type (integer or real) as the argument supplied. For example: **FRAC(3) = 0, FRAC(4.23) = .23, FRAC(–7.2) = .8, FRAC(3.4) = .4.**

ABS(*<numeric exprn>*)

Returns the positive (absolute) value of the numeric expression specified. For example: **A85(–25.3) = 25.3, ABS(17.1) = 17.1, ABS(0) – 0. ABS()** always returns a result of the same numeric type (integer or real) as the argument supplied.

SGN(*<numeric exprn>*)

Evaluates the numeric expression and returns –1 if it is negative, 0 if it is zero, and+l if it is positive. For example: **SGN(–4.5).=l,SGN(o) = o,SGN(352l)=l**. This result always has an integer type, regardless of the numeric type of the argument.

SGN(*<sign expm>,<value exprn>)*

Returns the value of the second argument with the sign of the first argument. The sign of the second argument and the value of the first are both ignored. For example: **SGN(4,10) = –10, SGN(0,–34) = 34, SGN(–3,–99) = -99**. This function returns a result of the same numeric type (integer or real) as the second argument. No type conversions are performed by this function, regardless of the type of either of its arguments. **SGN** may be used in an operator context (e.g., **X *sgn Y)*** to perform the same operation (Chapter 3, Section 5).

MIN*(<list of numeric expressions>)*

Returns the minimum value among a list of expression values. The expressions must be separated from one another with commas. For example: **MIN(2,1) = 1, MIN(45,Z987, –12,0,34) = –12. MIN()** returns an integer result only if all expressions listed evaluate to integer values. If one or more expressions evaluate to real values, the result will also be a real value. The INDEX function returns the sequence number of the minimum value returned by **MIN()**. When two arguments are used and the **INDEX** feature is not needed, you may want to use the **MIN** operator instead of the **MIN()** function (e.g., **X MIN Y** instead of **MIN(X.Y)**) because it is somewhat faster.

Any argument of **MIN()** can be a vector or vector expression if you precede it with the **VEC** reserved word. **MIN()** operates on vectors as if each of their elements was listed as a separate argument, so that afterward, the **INDEX** function returns the sequence number of the *argument* that was selected.

MAX(*<list of numeric expressions>)*

Returns the maximum value among a list of expression values. The expressions must be separated from one another with commas. For example: **MAX(2,1) = 2, MAX(4S,2,987,–12,0,34) = 987. MAX()** returns an integer result only if all expressions listed evaluate to integer values. If one or more expressions evaluate to real values, the result will also be a real value. The **INDEX** function returns the sequence number of the maximum value returned by **MAX()**. When two arguments are used and the **INDEX** feature is not needed, you may want to use the **MAX** operator instead of the **MAX()** function (e.g., **X max Y** instead of **MAX(X,Y)**) because it is somewhat faster.

Any argument of **MAX()** can be a vector or vector expression if you precede it with the **VEC** reserved word. **MAX()** operates on vectors as if each of their elements was listed as a separate argument, so that afterward, the INDEX function returns the sequence number of the *argument* that was selected.

SUM(*<vector expression>*)

Computes the sum of the elements of a vector or vector expression computation. The **VEC** reserved word is not used in **SUM()** because **SUM()** only operates on vectors. **SUM()** operates on either integer or real vectors and executes as much as 12 times faster than an equivalent interactive implementation. See Chapter 3, Section 7 for complete information on specifying and using vectors.

RND(*<numeric exprn>*)

Returns a pseudo-random number sequence uniformly distributed over the interval 0...1, not inclusive. The value of the argument expression controls the method of computation:

Zero argument	Returns the next number in the current sequence. Omitting the argument to also specifies zero.
0 <argument < 1	Defines a new starting seed based upon the argument.
Negative argument	Defines a new starting seed, based upon a quasi-random hardware condition.

The argument to **RND()** is used only as a real value, hence an integer argument will be converted to real before any computation begins. The result produced by **RND()** is, of course, a real value. **RND** without arguments is equivalent to **RND(0)**.

Sequences of random numbers are useful in computer simulations of realworld systems, probability studies, or any application requiring some element of chance. **RND()** produces only values uniformly distributed over the interval from 0 to 1, but you can transform its value for other random distributions, using simple calculations. Several of these methods are summarized below:

- **Uniform distribution within the interval (low,high):**

  LOW + RND * (HIGH–LOW)

- **Exponential distribution about a mean of M:**

  M – LN RND – 1.

- **Nonnal distribution with a mean M and a standard deviation S:**

  M + NORM*S, *where NORM is a teal function defined as:*

```
Def real X1, X2, S
Def real func NORM; Local S,X1,X2
Repeat; X1 = rnd; X2 = rnd; S = X1*X1 + X2*X2
Next if S>=1
Return X1 *SQRT(–2*LN SIS)
Func end
```

An excellent source of additional information about these random distribution methods and many others can be found in the book: The Art of Computer Programming, Volume 2, by Donald E. Knuth, published by AddisonWesly Publishing Company.

Section 2: Mathematical Functions

The functions described below all use real argument values to produce real result values. When integer arguments are supplied to them, MegaBasic automatically converts them to real form. Most of these functions are the so-called transcendental functions, which include trigonometric functions, logarithmic and exponential functions and square-root. Such functions return, out of necessity, approximations to the desired value rather than exact answers. MegaBasic computes these functions with accuracy out to the last digit of the prevailing floating point precision in most cases. For a small percentage of cases, the least significant digit will be off.

Sqrt(X)	Returns the square root of X.
Log(X)	Returns the common logarithm of X.
Ln(X)	Returns the natural logarithm of X.
Exp(X)	Returns e raised to the power of X.
Pi	Returns the constant pi to full precision.
Sin(X)	Returns the sine of X radians.
Asin(X)	Returns the radian angle of X expressed as a sine.
Cos(X)	Returns the cosine of X radians.
Acos(X)	Returns the radian angle of X expressed as a cosine.
Tan(X)	Returns the tangent of X radians.
Atn(X)	Returns the radian angle of X expressed as a tangent.
Poly(X,C(),D)	Evaluates the polynomial of X using a coefficient array.

SQRT(<*numeric exprn*>)

Returns the square-root of the expression value specified. An out of bounds error will occur if a negative argument is supplied to this function. For example **`SQRT(9) = 3, SQRT(1) = 1, SQRT(5.7) = 2.38746727`**, etc.

LOG(<*numeric exprn*>)

Returns the logarithm, base 10, of the expression value specified. An out of bounds error will occur if a negative or zero argument is supplied to this function. For example: **`LOG(1000) = 3, LOG(l) = 0, LOG(.123) = –.9100948886`**, etc.

LN(<*numeric exprn*>)

Returns the logarithm, base e (2.7182818...), of the expression value specified. An out of bounds error will occur if the argument specified evaluates to a negative or zero value.

EXP(<numeric exprn>)

Returns the constant e (2.7182818...) raised to the power specified by the numeric expression argument. An out of bounds error will occur if this argument is above +145.06286 (or +709.72683689 in **IEEE**), because it produces a result too large to represent in MegaBasic floating point representation. If the exponent is below –147.36549 (or –708.39641846867 in **IEEE**) then the result underflows and a zero result is returned without any reported error.

PI

Returns the constant pi (3.141592..) rounded to the prevailing precision of MegaBasic.

SIN(<numeric exprn>)

Returns the sine of the angle specified by the numeric expression. This angle must be expressed in units of radians, not degrees. To obtain the sine of an angle expressed in degrees, you must multiply that angle by the constant pi / 180 before taking the sine.

ASIN(<numeric exprn>)

Returns the angle in radians corresponding to the sine specified by the numeric expression argument. An out of bounds error will occur if the argument is less than –1 or greater than +1. The result returned is always an angle in radians between –pi / 2 and +pi / 2. To convert this radian result into degrees, you must multiply it by the constant 180 / pi.

COS(<numeric exprn>)

Returns the cosine of the angle specified by the numeric expression. This angle must be expressed in units of radians, not degrees. To obtain the cosine of an angle expressed in degrees, you must multiply that angle by the constant pi / 180 before taking the cosine.

ACOS(<numeric exprn>)

Returns the angle in radians corresponding to the cosine specified by the numeric expression argument. An out of bounds error will occur if the argument is less than –1 or greater than +1. The result returned is always an angle in radians between 0 and pi. To convert this radian result into degrees, you must multiply it by the constant 180 / pi.

TAN(<numeric exprn>)

Returns the tangent of the radian angle specified by the numeric expression argument. To obtain the tangent of an angle expressed in degrees, it must be converted to radians before taking the tangent, by multiplying it by the constant pi / 180.

ATN(<numeric exprn>)

Returns the angle in radians corresponding to the tangent specified by the numeric expression argument. The arctangent is computable for any real argument. The result returned is always an angle in radians between –pi / 2 and pi / 2. To convert this radian result into degrees, you must multiply it by the constant 180 / pi.

POLY*(<numeric exprn>, <real array vbl>, <degree>)*

Evaluates a polynomial on the first argument using a coefficient array (second argument) and a polynomial degree (third argument). The coefficient array must contain all the coefficients in ascending order and be *specified as an array reference to coefficient*$_0$ (the constant term). Invalid evaluations result if the degree specified extends past the end of the coefficient array.

For example to evaluate the polynomial: $1+3X+5X^2+7X^3+9X^4$, we first store the coefficients 1, 3, 5, 7 and 9 into array positions **`ARRAY(0...4)`**, then call **`POLY(X, ARRAY(0), 4)`** to evaluate this 4th-degree polynomial for a value of X.

Multi-dimensional arrays may be employed with the understanding that their last dimension index specifies the coefficient sequence position, and prior dimension subscripts serve to select one sequence of many. This follows from the sequential coefficient access employed by **`POLY()`** and the organization of array storage elements. For example **`POLY(X,C(I,J),5)`** evaluates a 5th-degree polynomial using the coefficient list C(I,J) on X, where I selects the sequence and J specifies the low-coefficient position of the six coefficients in the sequence (coefficient 0 to 5).

Section 3: Character and Bit String Functions

This section describes all functions related to character string processing and information about strings. A character string is a sequence of characters (or bytes) that is handled as a single data object, rather than as a multitude of separate characters. Such a data object has various properties which can examined and manipulated: they have a finite length, its characters are arranged in a particular order, their contents may represent words or numbers or other abstract variable length data.

Since each character in the string is stored in an 8 bit memory byte, strings can also be thought of as sequences of bits, otherwise known as bit strings. Each bit of a bit string can take on one of two values: 0 or 1, which can represent *Yes* or *No*, *True* or *False*, *On* or *Off*, membership or non-membership, or any other dual-valued relationship. Chapter 4 of this manual covers characters, strings and bit-strings in depth.

Len(A$)	Returns the character count in string A$.
Rev$(A$)	Returns A$ in reverse order.
Trim$(A$)	Trims the spaces from both ends of A$.
Min$(A$,B$,...)	Returns the minimum string among a list.
Max$(A$,B$,...)	Returns the maximum string among a list.
Asc(A$)	Returns the ASCII code of the first byte of A$.
Chr$(X)	Returns one-byte string of ASCII code X.
Chr$(F,L)	Returns the ascending sequence of ASCII characters from F to L.
Chrseq$(<ranges>)	Returns a string of multiple specified ASCII code ranges.
Str$(X,F$)	Converts number X into a string representation using optional format F$.
Val(A$)	Converts string representation of a number to a number.
Collat$(X)	Returns the string of X that can be string-sorted.
Match(A$,B$,S,N)	Returns a position in A$ where B$ is found.
Find(A$>,B$,W,N)	General purpose string searching facility.
Tran$(A$,B$,C$)	Translates characters in A$ and B$ to those in C$.
Reseq$(A$,N)	Resequences A$ from row-column to column-row order.
Bit(A4,I:W)	Accesses bit ranges in A$ as numbers.
Rotat$(A$,N)	Rotates string A$ by N bit positions (left or right).
Ord(A$,I:W)	Returns the first bit position in A$ set to one.
Card(A$,I:W)	Returns the count of bits set to 1 in A$.

Some of these string functions return a numeric result instead of a string result, and as such, may be used only in numeric expressions and not in string expressions. Except for the **VAL( )** function, which always returns a *real result*, all of the *numeric-string* functions return *integer results*. Considerable savings in computation time can be realized by avoiding unnecessary conversions between real and integer.

LEN(<*string expression*>)

Returns the length of the string expression. For example: **LEN("ABCDEFG")** returns 7, and **LEN("")** returns 0. **LEN()** requires more execution time and memory space when the string is not a simple string variable or constant because it has to completely evaluate the string expression before it can determine the length.

The length of a string variable (as opposed to its dimension) may be arbitrarily set using: **LEN(S$) =** *<length>*, where S$ is any string variable or string array element and<*length*> is any integer value not greater than the current dimension of the string. *New* characters created by extending the string are accessible but their particular values are unpredictable.

REV$(<*string expression*>)

Returns the string expression evaluated with its characters in reverse order. For example: REV$("abcdefg) = "gfedcba".

TRIM$(<*string expression*>)

Returns the string supplied after stripping all leading and trailing spaces. Although both ends of the string are trimmed, you can trim the spaces from one end only by appending a non-blank character to the opposite end, trimming the result, and then removing the non-blank character appended. For example the following expression trims *only the trailing* spaces from A$: TRIM$("*"+A$)(2)

MIN$(<*string1*>, <*string2*>,... <*stringN*>)

Returns the minimum string expression value listed, and sets **INDEX** to the sequence number of the one returned within the list. The strings are compared on the same basis as used in all MegaBasic string comparisons: from left to right according to the ASCII collating sequence. **MIN$()** is the string counterpart to the numeric **MIN()** function and operates in an identical fashion.

MAX$(<*stringl*>, <*string2*>,... <*stringN*>)

Returns the maximum string expression value listed, and sets **INDEX** to the sequence number of the one returned within the list. The strings are compared on the same basis as used in all MegaBasic string comparisons: left to right according to the ASCII collating sequence. **MAX$()** is the string counterpart to the numeric **MAX()** function and operates in an identical fashion.

CHR$(<*ASCII code*>[,<*last ASCII code*>])

Converts an ASCII code (any numeric value from 0 to 255) into a one-character string corresponding to the code supplied. The ASCII code conversion table in Appendix D.3 provides a complete listing of ASCII codes and their corresponding characters. Examples: **CHR$(65)="A~, CHR$(57)="9"**.

By supplying the second (optional) ASCII code parameter, **CHR$()** will return a string consisting of an ascending sequence of characters corresponding to all ASCII codes ranging from the first code given up to and including the second code given. If the second parameter is below the first parameter then a null string is returned. Examples: **CHR$(48,57)="0123456789", CHR$(87,43)="".**

In order to make this capability easier to use with printable characters, any argument to the **CHR$()** function can be either a number or a string. For example, **CHR$(65)** and **CHR$("A")** both return the letter **"A"; CHR$(48,57)** and **CHR$(0,9)** both return the string "0123456789". Only the leading character of a string argument is used and characters after the first one are ignored.

CHRSEQ$(*<range>,<range>,...)*

Returns a series of ascending ASCII character sequences, equivalent to a concatenation of multiple **CHR$()** functions. Each <*range*> consists of a single ASCII value or two ASCII values separated by a colon (:) to specify a range of ASCII values. For example, the following two expressions are equivalent:

```
Chrseq$(1 ,4,65:90,255)
Chr$(1 )+Chr$(4)+Chr$(65,90)+Chr$(255)
```

Notice that a colon is used to separate the end-points of an ASCII range in **CHRSEQ$()** functions and a comma is used as the separator in **CHR$()** functions. As with **CHRS()**, an ASCII value in any argument to **CHRSEQ$** can be specified by either a number or a string. For example, **CHRSEQ$(65:90,97:122)** and **CHRSEQ$**("A":"Z","a":"Z") both return the same result, but the string arguments make the purpose of the function much more readable. **CHRSEQ$()** is provided for applications that require so many character sequences that programming them with **CHR$()** is tedious and inefficient.

STR$(*<numeric expression> [,<format string>]*)

Converts any numeric value into a printable character representation of the number in string form (i.e., the character sequence you might type to enter the value from the keyboard). Examples: STR$(123.70)="123.7", STR$(–34E2)="–3400".

Without the optional format string, the value is converted using the default format currently in force. By including a second parameter string expression that evaluates to a valid numeric format (Chapter 7, Section 1), the numeric value is converted according to the format given. Formats to generate commas, fix-point, E-notation as well as hexadecimal, octal and binary conversions are supported along with many other options. For example: STR$(23.87,"6F1")=" 23.9", STR$(1234567.8,"CI")=" 1,234,568".

VAL(*<string expression>)*

Converts a character string representing a valid numeric value into the actual value it represents. This is the opposite of the **STR$()** function described earlier. The string may contain leading or trailing spaces (or line-feeds), but must be an otherwise valid numeric constant. The constant specified by the string may be any legal number as recognized by MegaBasic. This includes numbers in *E-notation*, as well as binary, octal and hexadecimal notation. The rules governing the formation of numeric constants are described in Chapter 3, Section 2, for example: VAL " 92E3") = 92000, VAL("–0012.430 ") = –12.43, VAL("7FFFh") = 32767.

A numeric string may be terminated by a comma, tab, linefeed or carriage return ASCII codes and characters beyond this point are not scanned or otherwise validated. Therefore the **VAL()** argument string may contain multiple numeric fields, of which only the first field is converted. By using the method described below, such multiple fields can be parsed and extracted.

After converting a string to a number using the **VAL()** function, you can find out the position in the string of the first character beyond the numeric character sequence

evaluated. The **INDEX** function returns this position right after the **VAL()** function is evaluated. If an error occurred because the string contained no *ASCII-represented number,* **INDEX** Will return zero. If all characters in the string argument participated in the numeric sequence, **INDEX** will return the length of the string argument plus one. This capability is useful for walking through a string that contains many numbers separated by spaces or other delimiters and picking out the numeric information. It is also useful for detecting the presence of incorrect characters in strings intended to contain a single number and nothing else.

ASC*(<string expression>)*

Converts the first character of the string specified into its corresponding numeric ASCII code. The function is the converse of the **CHR$()** function described below. Examples: ASC("A") = 65, ASC("9") = 57, ASC("") = –1.

Asc(S$) executes much faster when the string expression consists of only a string variable reference (indexed or unindexed), because the leading character in the variable can be accessed without evaluating an entire string expression.

COLLAT$*(<numeric exprn>)*

Converts a number into a string which is suitable for sorting as a string. In many applications, it is necessary to sort records containing some arbitrary set of fields, which may include both numeric and non-numeric information. The **COLLAT$()** function allows you to combine numbers with strings into a larger string record which, along with many others, is searched or sorted using purely string methods and operations. Although the **STR$()** function also converts numbers to strings, a string comparison between two such numeric strings by no means implies the same comparison result of the two values as numbers. **COLLAT$()** is designed to fulfill this need.

COLLAT$() converts an integer value into a four byte string, and real values into a 5-to-10 byte string, depending on the prevailing floating point precision of the **BASIC** being used. Comparing two *real* strings will produce the same result as comparing the two real numbers from which they came. Comparing two *integer* strings also compares as if they were numbers. However, you cannot compare an *integer* string with a *real* string and get any meaningful result because these two representations are not compatible. Hence you must ensure that the argument data type is what you expect it to be.

MATCH*(<search string>, <match string>[,<start>] [,<count>])*

Returns the character position (a number) in the <*search string*> that matches all the characters in the <*match string*>. Zero is returned if no matching string is found, or if either string contains no characters (i.e., null string), for example:

```
MATCH("abcdefg","de") = 4
MATCH("abcdefg","DE") = 0
```

An optional <*start*> may be specified to cause the search to begin on the character position given (skipping earlier characters of the string), for example:

```
MATCH("abcdefg","de",3) = 4
MATCH("abcdefg","de",5) = 0
```

An optional <*count*> may be given to specify how many occurrences of the search string to locate before returning the position found. If you specify a <*count*> then you must also specify the preceding the <*start*>, for example:

```
MATCH("This is the search string"," ",1,3) = 12
MATCH("This is the search string"," ",7,3) = 19
```

If this repeat count is omitted, a default of one is assumed causing the function to return the position of the first occurrence encountered. If the repeat count is higher than the number of occurrences that exist in the target string or if it is set to zero, then a zero result will be returned and **INDEX** will contain the number of successful matches found before failing. This capability greatly improves the performance of repetitive searches and pattern counting procedures. A typical application is the processing of packed string fields separated by spaces or commas (as above), where direct accessed to the Nth field or word is needed quickly.

Both the search string and pattern may be specified with general string expressions. Sufficient internal workspace (scratchpad memory) is required to temporarily contain both strings specified. If the <*search string*> *is* a simple string variable or an indexed string variable, MegaBasic searches the variable directly (i.e., without copying it to internal storage). If you specify a <*search string*> using any more complex a string expression than this, it always requires scratchpad memory to hold it for searching. The <*match string*> *is* always held in scratchpad memory while the search is in progress.

The **MATCH()** function is very fast and compatible with most other **BASICS** that include a string search function. However, the **FIND()** function, described next, is much more general purpose and, in some cases, it can provide even faster response.

FIND(*[<mode>]<vbl$><relation><string>[,<step>] [,<count>])*

This function provides a general purpose searching capability designed for a wide variety of applications. String variables may be searched forward or backward and the criterion of the search may include any comparison relation (not just equality as in the **MATCH** function described above). One of several types of searches may be selected by specifying an optional <*mode*> keyword. Before jumping into examples of the **FIND()** function, we will first describe each of the **FIND()** parameters in detail.

<mode>

This optional parameter selects the type of search to be done, and may be one of the following reserved words: **MIN, MAX** or **ORD**. **MIN** and **MAX** select a search for the minimum / maximum substring satisfying the search <*r elation*>. **ORD** selects an ordered table search for much faster response. When <*mode*> *is* omitted, the string being searched is assumed to be an unordered table and the search simply **FINDS** the first position that satisfies the comparison criterion. These modes are discussed in complete detail later on.

<vbl$>

This parameter specifies the string variable to be searched. The result returned by **FIND()** is always a character position within this string variable. The search can be restricted to any substring within this string variable by merely indexing the variable to the desired region within it. In such a case, the result position returned is still relative to the beginning of the string variable, rather than to the indexed region. The variable is

always searched in-place (i.e., without moving it anywhere first). **FIND** cannot search the result of a general string expression unless it is first stored into a string variable.

<relation>

This parameter specifies a criterion that constitutes a successful search. It is specified as any one of the six comparison operators: = <> < <= > >=. **FIND** returns the first position in the string variable where the comparison relation holds (i.e., *is true).*

<string exprn>

The <*string exprn*> parameter specifies the string that **FIND** will be searching for, and it may contain up to 255 characters. The length of this string is not related to or limited by the optional <*step*> parameter (described below). **FIND()** always compares this string with a substring of equal length in the string variable being searched.

<step>

Specifies the *step size* to advance through the search string variable for each successive comparison. Omitting the <*step*> results in steps of one byte (as done by the **MATCH** function). Steps larger than one are useful when the contents of the string variable are organized as a sequence of fixed length substrings (i.e., string records). For example a <*step*> of 5 compares substrings in the string variable at positions 1,6,11,16,21,... with the target string expression until the relation is satisfied. To search backwards through the string variable, you can specify a *negative* <*step*>, which searches from the highest possible comparison position in the string variable down to the first.

<count>

As in the *MATCH***()** function, the optional <*count*> specifies how many occurrences of the search string to locate before returning the position found. For example FIND(T$>S$,5,N) returns the position of the Nth 5-byte substring in T$ that is greater than S$.

If this count is omitted, a default of one is assumed, returning the position of the first occurrence encountered. If the count is higher than the number of occurrences that exist in the target string (or set to zero), then zero is returned and **INDEX** Will contain the number of successful matches found before failing. See the **MATCH()** function for other examples.

Results Returned by FIND( )

FIND returns the position in the string variable at which the specified comparison criterion was met. Zero is returned if no such location exists. No matter how the string variable may be indexed, this position is always relative to the beginning of the string variable, rather than to the beginning of the indexed region being searched (as returned by **MATCH**). For example, given that S$ = “ab-cd-ef-gh-ij-kl-mn-op”:

```
Find(S$(6)=“ij”) will return position 13.
Find(S$(1)=“ij”) will also return 13.
```

However, at certain times you may desire a result position which is relative to the indexed region, rather than to the absolute string base. For this reason, MegaBasic also computes this relative position and provides it in the **INDEX** function for your use if needed. For example, after each of the uses of **FIND()** above, **INDEX** returns the positions 8 and 13, respectively. **INDEX** is set in this manner for all but the ordered search case (when <*mode*> *is* **ORD**). In that case, after a successful search **INDEX** contains the same position returned by **FIND()**; after an unsuccessful search **INDEX** contains the position where the search string expression should be placed if inserted into the ordered table.

Using FIND( ) in the Default Mode

In default mode (i.e., no <*mode*> *is* specified), the **FIND()** searches the string according to the specified <*relation*> and <*step*> until it finds the first occurrence of success, even though there may be other locations further on that also satisfy the <*relation*>. Given that we are searching string TBL$ for string K$, the following examples should show how this works:

FIND(TBL$=K$)	Compare the characters in K$ with the same number of characters at **TBL$(1).** If equal then return position 1. Otherwise, repeat the comparison at **TBI$(2)**, then **TBL$(3)**, and so on until a match is found. If none is found then return zero. This particular example is identical with **MATCH(TBIS,K$)**.
FIND(TBL$(J)=K$)	Shows how to search from position J instead of 1. The result returned will be an absolute position in **TBL$** and the position returned in **INDEX** Will be relative to region **TBL$(j**).
FIND(TBL$>K$)	Instead of searching for equality, this search stops when the comparison finds the character position where **TBL$(i)** is greater than K$. You can specify any comparison operators.
FIND(TBL$<K$,8)	Performs the comparison at every 8th character, instead of one every byte.
FIND(TBL$<>K$,-8)	Searches backward through **TBL$**, starting from the end of **TBL$**. Suppose that **TBL$** is 800 bytes long. The search first compares substring **TBL$(793)** with K$, followed by **TBL$(785)**, **TBL$(777)**, and so on. Each comparison remains an ascending byte-by-byte compare but reverses the order comparisons are performed.

You may have noticed the similarity between **FIND** and **MATCH**. In fact, **FIND(T$-S$)** returns the same result as **MATCH(T$,S$)**. However **FIND** has several differences that should be emphasized. **FIND** is designed to find relationships in string tables by searching forward or backward through fixed length substrings. This generality makes **FIND()** slower for certain simple byte-by-byte matching applications. **MATCH** is designed for faster simple pattern matching and compatibility with similar functions in other languages and other dialects of **BASIC**.

Using FIND( ) in MIN or MAX Mode

By preceding the string variable to be searched with the reserved word **MIN** or **MAX**, you search for the minimum or maximum substring that satisfies the given comparison. As in all string comparisons, concepts of *higher, lower, minimum* and *maximum* are based upon the collating sequence set forth by the ASCII code set (shown in Appendix D). Although it is quite possible to use a looping process to find the minimum and maximum

substrings, implementing your search with **MIN** or **MAX** mode will execute as much as 40 times faster. For example, the following two searches are equivalent:

Case 1:	10 1 = Find(max TBL$ < K$, W)
Case 2:	10 1 = 0; J = 1 20 J = Find(TBL$(J) < K$, W) 30 If J=0 then 60; If 1=0 then [I=J; Goto 50] 40 If TBL$(J:W)>TBL$(1:W) then l=J 50 J = J+W; Goto 20 60 Rem -- l is now set to the position desired

Both cases search **TSL$** for the highest substring, W characters in length, which is less-than (<) the string contained in K$. As you can see, **MIN** and **MAX** mode is not only faster but also much easier to understand within your programs. If you wanted to search for the absolute minimum or maximum substring (i.e., not subject to any other comparison criterion), you need only specify a comparison which will always be true, for example:

FIND(MAX TBL$<>"", W)	Returns the maximum substring position.
FIND(MIN TBL$<>"", W)	Returns the minimum substring position.

One clever application for **FIND()** in **MIN** or **MAX** mode is substring sorting. Suppose that we sorted a table of substrings by moving the lowest substring in the table to the front, followed by the lowest of the remaining substrings, and so on through the table. This obvious sorting method that is rarely used because it is usually inefficient to implement. But with the **FIND()** function, this method is the fastest way to sort a string table of up to several hundred substrings (i.e., other methods are faster on longer lists). The example below sorts **TSL$** into ascending order using this method:

```
10 For I = 1 to Len(TBL$)-W by W
20 J = Find(min TBL$(1) <= TBL$(1:W), W)
30 Swap TBL$(1:W), TBL$(J:W)
40 Next I
```

To sort the substrings into a descending order, just change the **MIN** in the **FIND()** to a **MAX**, and change the less-than-or-equal (<=) to a *greater-than-or- equal (>=).* For fun, see if you can eliminate the need for the intermediate variable J in the above sort program to make it a *3-statement* sort.

Using FIND( ) in ORDered Mode

An ordered substring table is a series of substrings such that given any two consecutive substrings in the table, the first one is always less than or equal to the second one. Such a table can be searched very quickly a *binary search.* Instead of searching sequentially until the desired substring is found, a binary search proceeds somewhat differently. It first examines the middle substring to determine which *half* of the table to search next. Then, it examines the middle substring in that half of the table. On each iteration, a binary search *rules out half* of the region the desired substring is known to reside and very quickly converges on the target substring.

To specify that the string variable is an *ordered string table* and that a *binary search* is to be used, you precede the search table string variable with the word **ORD**. The **ORD** mode is well suited to applications where massive table lookups dominate the processing time. For example a program that reads a text file to determine all the unique words and how many times they occur can greatly benefit from using the **FIND()** function in **ORDered** mode.

FIND() in **ORDered** mode has some important differences and restrictions as compared with the other **FIND()** modes. These are described below along with several other fine points:

- The sign of the <*step*> does not determine the direction of the search. Instead, a negative <*step*> specifies that the table is in descending order, a positive <*step*> specifies a table in ascending order. In any case, the <*step*> must be a number in the range from –32767 to +32767.
- The only <*relation*> supported in **ORDered** mode is equals (=). If you specify any other <*relation*> (i.e., < <= > >= <>), an *Out Of Context Error* will result.
- When the search for equality succeeds, **INDEX** is set to the same location as the result position returned. When the search fails, the result returned is zero, but **INDEX** is set to the position in the table where the string would be, had it been found (i.e., insert it here for future lookups). This feature makes table maintenance quite a bit easier. An example follows shortly.
- When the search table contains more than one instance of the string sought, an **ORDered FIND()** will return the position of the one beginning the sequence. In an ordered table, such occurrences will be in consecutive positions. In an ascending table, this is the one toward the beginning of the table; in a descending table, this is the one toward the end of the table.
- MegaBasic does not check that the substrings are perfectly ordered throughout the table, nor does it check that the direction of the ordering is as specified. Hence, searching an unordered table as though it were ordered will yield meaningless results.

It is imperative that you ensure that the substrings in string variables to be searched in **ORDered** mode are really ordered the way you think they are, and without any exceptions. Even one exception to this rule can easily lead to a totally meaningless result.

The example below illustrates a good model of how you go about building and maintaining an ordered string table. It is a program that simply gets an input string from the user, finds it in a table, inserts it if it is a new string, and increments a count if it has been entered before.

```
10 Rem *** Table Insertion into an Ordered Table
20 W = 20; DIM TBL$(1000*W), COUNT(1000), STRING$(W)
30 TBL$ =; Rem -- Set TBL$ to empty
40 Input Enter string to insert -- ,STRING$
50 If STRING$ = then End; Rem -- Done when null string en-
tered
60 1 = Find(ord TBL$ = STRING$, W)
70 If I then [J = (1-1 )/W; COUNT(J) = COUNT(J)+1; Goto 40]
80 Rem -- String not found, so insert at position - INDEX
90 1=index; TBL$(1:0):=(STRING$+ *W)(1:W); Goto70
```

In line 10, the substring length is defined (W=20) and the string and count tables are defined. Line 30 sets the table to empty, which ensures that trailing spaces do not corrupt the order nature of the table. Line 40 inputs a string from the user and line 50 terminates the program if a null string was entered (i.e., only a carriage return was typed). Line 60 searches for the input string in the table using an **ORDered** find**()**. Line 70 increments the count corresponding to the string location found in the table. Line 90 inserts a new string into the table using an insertion assignment statement (:=). Before being inserted, the string is padded with extra spaces (as needed) so that its length is exactly 20 characters long. It may prove beneficial to type this program into MegaBasic and play with it for a while to get comfortable with concepts it uses.

TRAN$(<*tar get string*>, <*original chars*>, <*translated chars*>)

Translates one set of characters into another set of characters throughout the given <*tar get string*>. The translation is based on the characters in corresponding positions of the second and third arguments. When the <*original chars*> and the <*translated chars*> differ in length, the longer is truncated to the length of the shorter. If the <*original chars*> or the <*translated chars*> is null ( ) then the <*tar get string*> is returned unchanged. Neither the second nor third arguments may exceed 256 characters without causing a *Length Error.*

```
TRAN$("ABCDEFG","BDFR","xxxx") = "AxCxExG"

TRAN$("capitalized vowels","aeiou","AElOU")
= "cApltAllzEd vOwEls"
```

One application of **TRAN$()** is to change the collating sequence used by a string sort, by mapping all key characters into another character set, sorting the strings, then re-mapping the keys back to their original character set. Other useful applications include upper and lower case conversions and mapping between ASCII and **EBCDIC** codes or other character sets.

Another application for **TRAN$()** is constructing *classification strings* for strings that you want to break into words, numbers commands, sentences, etc. For example suppose you translate all letters to *L*, all digits to D, all spaces, line-feeds and tabs to S, all punctuation characters to P and all other characters to ?. Since the resulting string has a classification letter for each corresponding character in the original string, words appear as a sequence of LLL's, numbers as DDD's, blank space as SSS's, and so on.

A second, faster form of **TRAN$()** is supported that can save time in applications that make heavy use of this function. It has the following syntax:

```
TRAN$(<target string>: <translation map>)
```

where the <*translation map*> *is* a string variable containing the translation characters in the following form. At each position corresponding to the ASCII code of the original character (i.e., ASCII 0 in position 1, ASCII 1 in position 2, and so on) you store the character you want that code translated to. For example, to translate an "A" to an "a", the character in the map at position 66 would be an "a". The map always begins with ASCII 0 and continues sequentially up to the highest code you want to translate. ASCII codes beyond this limit are left unchanged. The intervening positions corresponding to untranslated characters must be filled with those same characters.

This form is much faster than the former (typically 2 times), especially when the <*tar get string*> is short. This is because the translation table doesn't have to be set up on every invocation. Although it takes extra work to initially set up the table, you can define all your translations once, then refer to them by name as needed.

RESEQ$(<*tar get string*>,<*step ske*>)

Resequences the bytes of the <*tar get string*> into a different order which depends upon the value of <*step size*>. Given a <*step size*> of N, the sequence returned always begins with the first byte of the <*tar get string*>, followed by the Nth byte after that, followed by every Nth byte after that through to the end of the string. This sequencing then *wraps around* to the beginning of the string and continues the cycle until all bytes have been accessed. If a byte is accessed a second time before all bytes have be output, it is skipped and the cycle continues from the next byte instead. A few examples should clarify how this works:

```
RESEQ$("AaBbCcDdEeFfGgn,2) = "ABCDEFGabcdefg"
RESEQ$("ABCDEFGabcdefg",7) = "AaBbCcDdEeFfGg"
RESEQ$("abcdefghijkl mn",3) ="adgjmbehkncfil"
RESEQ$("AAAAbbbbcccc",4) - "AbcAbcAbcAbc"
```

This unusual function has a number of applications which would execute hundreds of times slower in ordinary **BASIC** statements. The most important of these is the ability to restructure a string of substrings in row-column order into a string in column-row order (as shown by the first two examples) and back again. Applications include character re-distribution procedures, text formatting, character and attribute processing for graphics, sector translation tables for operating systems, Monte Carlo simulations, etc. **RESEQ$()** requires a scratchpad area equal to twice the length of the argument string to perform this process.

ROTAT$(<*string exprn*>, <*rotation distance*>)

Rotates a byte string by the number of bit positions specified by the <*rotation distance*>, as given by a numeric expression that evaluates to a number from –524287 to 524287. A negative distance rotates the string to the left (toward the beginning) and a positive distance rotates it to the right (toward the end). No rotation takes place when a distance of zero is specified or the string is a null string ( ). The entire string rotates as a unit and bits that *fall off* the end are moved to the other end (i.e., no information is lost). Execution time is linearly related to the length of the <*string exprn*> and unrelated to the rotation distance specified. Rotating by multiples of 8-bits rotates the string by character positions, because each character is represented by a sequence of 8 bits.

ROTAT$ has numerous applications which would not normally be feasible in **BASIC**. Checksums and string hashing are efficiently implemented using exclusive-OR (**XOR**) in conjunction with **ROTAT$**. Rotating bit-masks into desired positions is another application. Communicating 8- bit binary information through a 7-bit com-line is easily accomplished using procedures which unpack 8-bit data into 7-bit bytes at one end, and reassemble them back into 8-bit bytes at the other end. See the **LIBRARY.pgm** file for examples

BIT(<*string variable*> [<*bit position range*>])

Returns the numeric value represented by the range of bits specified within the string variable. If no bit range is supplied, then the value of the leading bit in the string is returned (0 or 1). Bit ranges are specified by a starting bit address and either a length (number of bits) or an ending bit address (similar to string indexing). In no case are you permitted to specify a bit range of more than 24 bits. The value returned is always a positive integer from 0 to the maximum integer representable by the number of bits accessed. When the string variable is indexed to a smaller region of bytes within it, the

bit range is relative to the beginning of the indexed region, rather than to the beginning of the string variable. The following 5 examples illustrate the various ways you can specify a bit range:

BIT(S$)	Refers to the leading bit of S$.
BIT(S$:N)	Refers to the N leading bits of S$.
BIT(S$,1)	Refers to the Ith bit of S$.
BIT(S$,1:N)	Refers to the N bits beginning at bit I of S$.
BIT(S$,1,J)	Refers to the bits in position I through J of S$.

Bit positions are not the same as character string positions: for each byte position there are 8 bit positions. The actual bit string accessed will be shorter than specified if part of what is specified lies beyond the last byte of the string. An *Out Of Bounds Error* will occur if the actual number of bits accessed is less than 1 or greater than 24. This function may be placed on the left side of an assignment statement in order to set the bit range to some other value. This is described in Chapter 5, Section 2 along with additional information about the BIT**( )** function and bit string processing.

CARD*(<string> [<bit range>])*

Counts and returns the number of l-bits in the <string>, which may be either a string variable or a general string expression. **CARD** (an abbreviation of *cardinality*) may include a bit subrange identical to the **ORD** function (described below), providing an extremely fast and flexible method for counting the number of l-bits within any arbitrary bit string. This capability is particularly useful for bit strings representing sets, as it it tells you how many elements are contained in a set. **CARD( )** is identical with the **ORD( )** function described below except, of course, for the result returned. Refer to the **ORD** function above for details on specifying bit subranges in this function.

ORD*(<string> [<bit range>])*

Returns the bit position of the first bit in the <*string*> set to one. A bit range may be specified to restrict the bit search to any bit subrange within the <*string*>. The <*string*> may be specified by either a string expression or a string variable. When **ORD** searches an indexed or unindexed string variable it uses no internal scratchpad space and executes much faster than for the same string specified by a string expression. Regardless of the bit subrange specified, **ORD** always returns a bit position relative to the beginning of the byte string being searched. The following examples show the various options of the **ORD** function:

ORD(S$)	Returns the position of the first on-bit in S$.
ORD(S$,1)	Returns the position of the first bit set on or after position I.
ORD(S$,1,J)	Returns the first bit set within bit positions from I to J (inclusive).
ORD(S$,1:N)	Returns the first bit set within the N bits starting at bit position I.
ORD(S$:N)	Returns first bit set within the first N bits of S$.

ORD returns –1 if no bit is on in the specified bit-string. Bit positions may range from 0 to the length of the string * 8 minus 1, which can potentially exceed half a million. Only the portion of the specified bit rang which is actually defined in the byte string will be searched. The following example shows how to use the **ORD** function to display all the bit positions within string variable TBL$ that contain ones:

```
10l = –1
20 1 = ord(TBL$, 1+1)
30 If I > –1 then [ Print l; Goto 20 ]
40 End
```

ORD is extremely fast and permits bit strings to control other operations. For example, bit strings can represent arbitrary record selections in large data bases. Such bit strings can be combined and manipulated using logical operators to form complex result bit strings which are then scanned (using **ORD**) to generate a report. The **ORD()** function bears absolutely no relationship to the **ORDered** mode of the **FIND()** function (Chapter 9, Section 3). **ORD** is an abbreviation of **ORDinal** number, a number representing the order of elements in logical sets.

Section 4: File and Device I/O Functions

Most file and I/O functions require you to specify an *open channel number,* which specifies the file or device to be accessed. Open files and actual peripheral I/O devices can be addressed in an identical manner, so that your program can deal with both types as if they were identical in most cases. See Chapter 7, Section 1 for additional information about I/O independence and redirection.

Pos(D)	Gets or sets the column position on channel D.
Line(D)	Gets or sets the line position on channel D.
Inchr$(D,..)	Returns input characters from channel D. Variousoptions provide control over termination, echoing and time-out.
Edit$	Returns the previous line input, or the command *tail* that invoked the program.
Input(D)	Returns the input status of input channel D.
Output(D)	Returns the output status of output channel D.
File(F$)	Returns true (1) if file F$ exists, false (0) if not.
Filepos(D)	Gets and sets the byte position on open file D.
Filesize(D)	Gets and sets the total file size of open file D.
Filedate$(D)	Returns the date of last update for open file D.
Filetimie$(D)	Returns the time of last update for open file D.
Filectrl(D)	Returns the system file handle assigned to open channel D.
Space(D)	Returns the disk space remaining on drive D.
Dir$(F$)	Returns the file name following F$ in the directory.
Subdir$(D$)	Returns the subdirectory name after F$ in the directory.
Open$(F)	Returns the full name of the file **OPENed** under channel F.
IOCTL(D)	Test l/O channel D for *IOCTL* capability.
IOCTL$(D,C$)	Sends control string to channel D and returns responses.
Typ(D)	Test for an endmark at the current file position.

Numeric arguments supplied to these functions may be specified either as integer or real values. Internally, however, only integer data is used and floating point arguments will automatically be converted to integer form before they are used. These functions all return integer results.

POS(<channel number>)

Returns the current column position of the device specified by the numeric argument. This channel number must correspond to an I/O device or open file number (0 to 31). Column positions range from 0 to 255. The column position is automatically maintained by MegaBasic as characters are sent to the device (or open file) via **PRINT** statements, according to the following rules:

- Set to position zero on each carriage return (ASCII 13) and by the initial **OPEN** statement the made the file or device available.
- Set to the next multiple of 8 on each tab character (ASCII 9).
- Incremented by one on all codes from 32 to 255.
- Decremented by one on each backspace character (ASCII 8).
- Unchanged by all other control characters (0 to 31).

MegaBasic cannot, however, account for cursor position changes caused by special escape sequences, function codes or system calls. Therefore **POS()** can be set to any value (0 to 255) using an ordinary assignment statement. Using this feature, your program can correct the current column position for a device whose positions have been altered by non-standard cursor positioning. Because of the limited range of the column position (i.e., 0 to 255), if you print a string longer than 255 characters, the channel position will *wrap around* back through zero, resulting in a meaningless position when subsequently examined.

LINE(<channel number>)

Returns the current line position of the device specified by the numeric argument. Line positions range from 0 to 255 change according to the following rules:

- Set to zero by opening channel D, by a **PRINT** statement containing a plus sign (+) control or by a form-feed character (ASCII 12) sent to the device (except the console #0).
- Incremented by one on every line-feed character sent to device D.
- Can be set to any value from 0 to 255 using an assignment statement. Once the value 255 is reached, however, the **LINE()** position will *wrap-around* back to zero and start over.

This function is useful for controlling page length while your program is generating output. Without it, your program would have to count output lines itself. As with the **POS()** function, MegaBasic cannot account for cursor position changes caused by special escape sequences, function codes or system calls. Therefore **LINE()** can be set to any value (0 to 255) using an ordinary assignment statement. Using this feature, your program can correct the current row position for a device whose positions have been altered by non-standard cursor positioning.

INCHR$(<channel>,<length>,<break set,<echo>,<time out)

Inputs characters from the channel specified. Control characters and other binary data can be input through this functions. Several options control device selection, input termination and echoing using the following arguments:

<channel>	Specifies the open device or file number of the input channel. Except for the console, printer and auxiliary devices (0,1 and 2), a channel must be **OPENed** before it can be used. Unlike those that follow, this argument is not optional.
<length>	Specifies the absolute maximum number of characters to input before returning them to your program. MegaBasic allocates memory for this number of characters before the input process begins, so don't specify unreasonable values here to simulate *indefinite* input (to be terminated by other means).
<break set>	Specifies a set of characters any one of which terminates the input, returning all preceding characters. Afterward, **INDEX** is set to the character *position* in the <*break set*> of the character that terminated the input. A null break set string can be specified to indicate no break set. If the limiting input <*length*> has been reached without encountering any of the characters in the break set, **INCHR$()** returns with all characters input and sets **INDEX** to zero.
<echo>	A non-zero value specifies that each character is echoed back to the channel as it is input. A zero value suppresses echoing, which is the default when this argument is omitted.
<time out>	Specifies the time in (real) seconds to wait for the next input character before *timing out* and returning with all the input characters received up to that point. The time out specifies the period *between* characters. A time *out of zero* terminates input as soon as the next character is *not immediately available without waiting* (e.g., those in the *type ahead* buffer). Fractional seconds are supported to the time resolution level supported by the host operating system.

All **INCHR$()** arguments except the channel number are optional. Arguments can only be omitted from right to left (i.e., you cannot omit arguments *out of the middle).* This results in five possible ways to specify **INCHR$()**. Examples of each form are given below along with what they do:

INCHR$(D)	Waits for 1 input character from channel D, without input echo.
INCHR$(D,L)	Waits for *L* input characters from channel D and does not echo them.
INCHR$(D,L,B$)	Waits for up *to L* input characters from channel D, or until an input character matches one in *B$.* No characters are echoed.
INCHR$(D,L,B$,E)	Waits for up to *L* input characters from channel D, or until an input character matches a character in *B$.* Input characters echoed if *E*<>*O* .
INCHR$(D,L,B$,E,S)	Inputs up to L input characters from channel D, or until an input character matches a character in *B$,* or until the *time between* input characters exceeds S seconds. All input characters are echoed if *E is* non-zero.
INCHR$(D,1,"",0,0)	Inputs 1 character from channel D if it can be input without waiting. The input the character is not echoed

INCHR$() will also input characters from a file if the file is **OPEN** and you specify the open file number as the channel. No end-of-file mark is checked for, although an *Out Of Bounds Error* will occur if you attempt to read past the physical end of the file. Do not use **INCHR$()** in this manner directly in the data list of a **WRITE** statement to the same file, as it will upset the file pointer for the subsequent **WRITE** operation.

EDIT$

Returns the last line input from the console, i.e., the contents of the *old line* buffer. **EDIT$** has no arguments. Whenever you execute a program, either from the MegaBasic command level or the operating system command level, the old line is set to the command *tail*. **EDIT$** can therefore retrieve this command *tail so* that your program can extract any additional arguments it contains. For example if you run a program from the operating system using the command:

```
BASIC Program arg1 arg2 arg3
```

When *Program* begins execution, **EDIT$** will return the string:

```
"Program arg1 arg2 arg3"
```

This *command tail* must be used before your program requests console input via the **INPUT** statement, because the edit buffer is then overwritten and its prior contents are lost. For testing purposes, the RUN command (in the development version of MegaBasic) copies the text of itself into the *old line* buffer in a manner identical to the above scenario.

If the program is either compiled or running under the **PGMLINK** system, the program name is stripped from the front of the command tail string. You can use **PARAM**(16) to decide in your program whether or not the first word in **EDIT$** is the program name. Because of the volatility of the command tail and its dependence upon the execution model of your program, a program that will be using **EDIT$** for the command tail should begin execution with a sequence like the following:

```
DIM TAIL$(128)
TAIL$ = TRIM$(EDIT$)
IF PARAM(16)<2 THEN
    TAIL$ = TAIL$(MATCH(TAIL$+" "," "))
```

This program fragment sets string variable **TAII$** to the argument string typed after the program name without the program name portion (which normally is not needed) and without depending on any particular execution model of MegaBasic. Subsequent use of the command tail can then access string variable **TAIL$** without worrying about any other issues.

INPUT*(<channel number>)*

Returns *true (1), false (O)* or *unknown (–1)* to indicate the input status of the channel specified. If the channel is an **OPEN** file number, then **INPUT()** returns *false (O)* if the current file position is beyond the end of the file or the character about to be **READ** is the endmark code, otherwise **INPUT()** returns *true (1)*. Unknown (–1) is returned only when the input status is unavailable from the host operating system or the system returns an error for the input status request (e.g., the input status of an output-only channel like a printer).

Do not use **INPUT()** to test for the end-of-file on a binary or non-text file (i.e., any file the you are using **READ** and **WRITE** statements on). It will give false indications of end-of-file depending on the file contents. For such files, use the comparison **FILEPOS(N)>=FILESIZE(N)** to make such a test.

When the console keyboard input status is interrogated (i.e., **INPUT(0)**), YOU should remember that the Ctrl-C detection system of MegaBasic swallows incoming keyboard characters when they are typed during statement execution. Hence you should disable Ctrl-C when using **INPUT()** to control programmed input from the keyboard (using Param(1) = 1). **INPUT()** returns the input status without actually reading the character present. **INPUT(0)** may also be spelled as **INPUT** (i.e., without an argument) to mean the same thing.

OUTPUT(*<channel number>*)

Returns *true (1), false (O)* or *unknown (–1)* to indicate the output status of the device specified. If the device is an **OPEN** file number, then **OUTPUT()** returns *true (1)* all the time. Unknown (–1) is returned only when the output status is unavailable from the host operating system or the system returns an error for the output status request (e.g., the output status of an input-only device like a 80-column card reader). **OUTPUT(0)**, which requests console output status, may also be spelled as **OUTPUT** (i.e., without an argument) to mean the same thing.

FILEPOS(*<open file number>*)

Returns the position of the file pointer of the specified open file number. The file position represents the number of bytes between the beginning of the file and the current file location where the next byte would be transferred by a sequential **READ** or **WRITE**. Hence the beginning file position is zero. The number returned by **FILEPOS()** is always in integer mode.

FILEPOS() may appear on the left-side of an assignment statement (e.g., **FILEPOS(5) = N*L)** in order to explicitly change the current file pointer, rather than setting as part of a **READ** or **WRITE** statement. This is discussed more fully in Chapter 7, Section 2.

FILESIZE(*<open file number>*)

Returns the number of file blocks in the open file number specified. Under the **MS-DOS** operating system, **FILESIZE()** returns the file size in units of bytes instead of blocks. This value always points to the first byte position past the end of the data on the file. You may therefore correctly append information to any file simply by beginning sequential file transfers at this file location.

Also under **MS-DOS**, the **FILESIZE()** function may be placed on the left side of an assignment statement in order to set the file size to any byte length. This feature will usually be applied to reduce a file size, since any file will be extended automatically by the **WRITE** statement as needed.

To be compatible with earlier versions of MegaBasic, **FILESIZE()** returns the file size in units of 256 bytes per block under **CP/M, MP/M** and **TURBODOS** operating systems. You can however employ any block size from 1 to 65535 bytes per block by setting Param(14) (Chapter 9, Section 5) to the size of your choice (independent of operating system).

FILEDATE$(<open file number>)

Returns the date that the specified open file was most recently modified. The date is returned as an 8 character string in **MM/DD/YY** form (month / day / year). This function depends on the host operating system for file date maintenance and access, a facility not available under some operating systems (e.g., **CP/M-86**). **FILEDATE$()** returns a null string (i.e., a zero length string) whenever the date information is not available for any reason.

FILETIME$(<open file number>)

Returns the time that the specified open file was most recently modified. The time is returned as an 11 character string in 24-hour **HH:MM:SS.DD** form (hour:minute:sec.decimals). This function depends on the host operating system for file time maintenance and access, a facility not available under some operating systems (e.g., **CP/M-86**). **FILETIME$()** returns a null string (i.e., a zero length string) whenever the time information is not available for any reason.

FILECTRL(<open file number>)

Returns the internal system *file handle* associated with the specified <*open file channel*>. This function is typically not required in most applications, but it is provided to support *direct system calls* involving files opened under MegaBasic for special applications. If the file was **OPENed** using the obsolete *file control block* method (**FCBS** supported under **DOS 2.xx** only), **FILECTRL()** returns the *offset address* of the ***FCB*** instead of the *file handle.*

MegaBasic has no knowledge of any change in file state made using system calls with access to files using file handles, which can interfere with subsequent MegaBasic file operations. For example, *closing* a file using a direct system call followed by MegaBasic file operations involving that *same open file* can have unpredictable results.

FILE(<file name string exprn>)

Looks up the file name specified in the file directory and returns *true* if it is present (1 =r / w file, 2=r / o file), or *false* if it is not present (0). Since a trappable *Improper File Name Error is* reported for bad file names, this function is also useful in testing for correct file name formation.

SPACE(<drive number>)

Returns the amount of available unused disk space remaining on the disk drive number specified. Use 0 to select the default drive, or 1, 2, 3, ... to refer to any available disk sub-system (i.e., for drive *a:, b:, c:, ...).* Like the **FILESIZE()** function described above, the **SPACE()** function returns a result which is scaled into file blocks: 256 bytes / block under **CP/M** and 1 byte / block under **MS-DOS**. You can however control the block size by setting **PARAM**(14) (Chapter 9, Section 5) to any byte count from 1 to 65535 (independent of operating system).

DIR$

Returns the current directory pathname on the default drive. You can also select different directories on the default drive by assigning your desired directory string to **DIR$** as an assignment statement (Chapter 7, Section 2), for example: DIR$ = “ \bin\basic”.

DIR$(<*prior file name string*>)

Returns file names, one at a time, from the file directory. The string argument must specify the last file name returned from this function, or a drive designator string which causes the first file name on that drive to be returned. Drive designator strings consist of a drive letter followed by a colon (e.g., "A:", "B:", "C:", etc.) or a null string, which selects the default drive. A null string is returned if there are no files on the drive, or file name F$ cannot be found on the drive, or file name F$ is the last file in the directory.

The purpose of this function is to provide a sequence of file names from which your program can select and subsequently process. File directories simply become input data for your programs. Because the names are returned in string form, arbitrary pattern selection criteria can be imposed, i.e., you are not limited to wild-card character (? and *) matching that is usually provided by the operating system. The following example illustrates how one might generate a list of all files from drive B:

```
10 LAST$ = Dir$("b:")
20 While LAST$>; Print LAST$;
30 LAST$ = Dir$(LAST$); Next
```

MegaBasic always looks up the file name supplied and after finding it, returns the subsequent file name from the directory. This approach permits unlimited file processing between one file name and the next without affecting the file name sequence returned. With proper programming, you can even sequence through the file names from multiple disks simultaneously.

This function does have one important limitation. If the last name returned from **DIR$()** is renamed or deleted, it will not be found the next time that **DIR$()** is called, causing the name sequence to terminate. This can be avoided if you defer such a delete or rename until the next file name has been extracted, or the sequence is restarted from a prior file name known to reside in the directory.

SUBDIR$(<*prior subdirectory name string*>)

Returns each subdirectory name, one at a time, from the currently selected directory. It is similar to the **DIR$()** function described above that extracts file names from the current directory. **SUBDIR$()** extracts directory names, permitting programs to *walk* through the subdirectories on the disk. See the discussion about the **DIR$()** function above to see how to use the **SUBDIR$()**.

OPEN$(<*file number*>)

Returns the name of the file or device **OPENed** under that number, or a null string if nothing is **OPENed** under the number specified. File names are returned with the drive code, full directory pathname and the file name with its extension. Device names consist of only a name, and any drive, path or extension specified in its original **OPEN** is not given in the name string because such information does not apply.

OPEN$() is useful for testing a channel number for availability (i.e., where it returns a null string), for obtaining file names given only file numbers, for extracting the directory path of an open file (or its drive code), and for determining whether an open channel is a file or an I/O device (only files have a drive code in their names). **OPEN$()** is also useful for generating a *fully-qualified* file name string that correctly identifies the file without any knowledge of the current drive and / or directory.

IOCTL*(<open channel number>)*

Returns true (1) or false (0) to indicate whether the device opened under the specified channel number (i.e., file number) is capable of processing I/O control strings. **IOCTL()** is not supported under all operating systems. The **IOCTL** statement is described in Chapter 7, Section 1.

IOCTL$*(<channel number> [,<output ctrl string>])*

Returns an input control string from the channel opened under the channel number specified (i.e., file number). A null string is returned if no input control string is available or if the channel does not support control strings. If you specify the optional output control string, it will be sent to the channel before the input control string is requested. **IOCTL$** is not supported under all operating systems. See the **IOCTL** statement in Chapter 7, Section 1 for further details.

TYP*(<open file number>)*

Returns the data type at the current file position of file number X. The data types returned are as follows: 0 = **ENDMARK**, 1 = String, 2 = Floating Point Value, 3 = Unknown (most likely integer or other binary data). **TYP()** always returns 0 whenever the file position is higher than the last byte written to the file (i.e., **FILEPOS() >= FILESIZE()**).

TYP() does not and cannot work correctly on files that contain any data other than **BCD** numbers and *normal* strings (i.e., those written with string headers). The potential conflicts include integers, **IEEE** floating point values and any data written in 8-bit (&) or 16-bit (@) mode. These data types may be incorrectly reported by **TYP()** as numbers, strings or endmarks.

TYP() is useful primarily for detecting the end-of-file mark (endmark) at the end of logical records or files, and for compatibility with North Star **BASIC** data files and early MegaBasic releases. Because of this, **TYP()** is only supported for compatibility with programs that already use it. Avoid reliance on **TYP()** in any new programs.

Section 5: Utility and System Interface Functions

This section describes various functions that provide direct access to memory and machine ports and various internal MegaBasic operating parameters and numerous utility functions. See Chapter 7, Section 3 for important details about Intel 80x86 memory addressing. That section also describes other system interface facilities. Unless otherwise specified, all arguments to these functions are best specified as integers, and numeric results are returned as integers.

Time$	Returns the current time in an 11-byte string.
Date$	Returns the current date in an 8-byte string.
Elapse	Returns elapsed time in seconds since previous **ELAPSE** call.
Argument	Returns argument-remaining status in open-end argument list.
Interrupt()	Returns status and control data on logical interrupts.
Struct()	Returns position, length and type data for **STRUCT** fields.
Real(I)	Returns the value of expression I in real mode.
Integer(R)	Returns value of expression R in integer mode.
Dim(V)	Returns current dimension specifications of variable V.
Errtyp	Returns the error code of the previous error.
Errmsg$	Returns the error message string of the previous error.
Errpkg$	Returns the package name where the last error occurred.
Errline	Returns the line number where the previous error occurred.
Errdev	Returns the channel selected before the previous error.
Index	Returns secondary result values provided as side-effects after executing various functions.
Free(T)	Returns one of a variety of free-memory resource statistics.
Inp(P)	Returns 8-bit data directly from **CPU** port I?
Exam(M)	Returns the contents of memory at hardware address M.
Seg(V)	Returns absolute segment address of variable.
[V]	Returns absolute offset address of variable.
Envir$()	Accesses **MS-DOS** environment strings.
Param(T)	Provides access to various internal MegaBasic control parameters.

TIME$

Returns the current time, as provided by the operating system, in an 11-byte string. A null string is returned if the time is not available from your particular system. A 24-hour format is provided that includes the hour, minute, second and 1/100s seconds (e.g., 12:34:56.78). The first character of **TIME$** is a blank character whenever there is no ten-hour digit. Since **TIME$** is a string, you can easily create any other time format with further string operations.

DATE$

Returns the current date, as provided by the operating system, in an 8-byte string. A null string is returned if the date is not available from your particular system. The date is provided in *month/day/year* format with two decimal places for each value. The first character of **DATE$** is a blank character whenever the month can be expressed in one digit. Since **DATE$** is a string, you can easily create any other date format with further string operations.

ELAPSE *[(<mode>)]*

Returns the number of seconds that have elapsed since the last time that **ELAPSE** was called. The first time it is called, **ELAPSE( )** returns the number of seconds since 00:00:00.00 (midnight of the previous night). You can specify a non-zero argument to get the time without resetting it for subsequent calls. Normally, **ELAPSE** is called without specifying any arguments (e.g., X = **ELAPSE**) .

ELAPSE returns a real result, which may include fractional seconds down to the millisecond level depending on the time capabilities provided by the operating system. Under **MS-DOS**, the precision of **ELAPSE( )** resolves down to the 1 millisecond level. However, do not rely on this level of accuracy when running MegaBasic under Microsoft WINDOWS because the timing base is unpredictable. **ELAPSE** returns zero under systems that do not support a usable time-base.

ARGUMENT

Returns *true (1)* if any *open-ended arguments* (Chapter 8, Section 4) remain to be read from the argument list of the current subroutine call, or *false (O)* if none.

INTERRUPT(<selector> [,<interrupt number>])

Returns selected status and other information about logical interrupts. See Chapter 4, Section 4 for a full description of logical interrupts and this function.

STRUCT(<field name> [,<selector>])

Returns selected information about structure field variables (Chapter 5, Section 3).

REAL(<numeric exprn>)

Returns the value specified by the argument expression in real mode, regardless of its original numeric type (integer or real). All integers can be converted into real values without any precision loss, except when you are using 8-digit floating point precision. In that case, integers with absolute values above 100 million cannot always be converted exactly because they cannot be represented exactly in such a limited floating point word size. If this is a problem, you should switch to a version of MegaBasic that uses 10-digit floating point or higher (e.g., IEEE / 8087 MegaBasic).

Although MegaBasic converts between integer and real mode automatically as needed by internal processing requirements, you may occasionally need to force an integer expression into real mode. For example writing the contents of an integer variable onto a file as a floating point number instead of an integer. Another example is the following expression:

```
X + Real(l) * Y
```

where X and Y are real and I is integer. In this example, MegaBasic might convert Y to integer (depending on its value) so that the multiplication can be performed using the much faster integer multiply. Upon completion, however, the integer product would be converted back to real mode to be added to real X. Therefore we are forcing I to real mode to prevent the unnecessary conversions from slowing down the computation.

INTEGER(*<numeric exprn>*)

Returns the value specified by the argument expression in integer mode, regardless of its original numeric type (integer or real). Be aware that this function truncates non-integral real numbers into integer whole numbers. A numeric overflow error will occur if a real number below –2,147,483,648 or above 2,147,483,647 is supplied as an argument to the **INTEGER()** function.

Although MegaBasic converts between integer and real mode automatically as needed by internal processing requirements, you may occasionally need to force a real expression into integer mode. For example writing the contents of a real variable onto a file as an integer instead of a real. Another example is the following expression:

```
Integer(X) + I + J + K
```

where I. J and K are integer and X is real. In this example, if real X were to remain real, each of the integer terms being added together would be converted to real and added in floating point to X. Therefore X is forced into integer mode to prevent the unnecessary conversions from slowing down the computation.

DIM(*<variable name> [, <dimension number>]*)

Returns the number of dimensions currently defined for the variable specified, which can be any variable name, string or numeric, but no subscripts or indexing or parentheses are to be included with the name. Zero is returned if variable V is not dimensioned. If the second argument is supplied, **DIM(V,D)** returns the upper limit defined for dimension D of variable V. The dimensioned maximum size of a string variable array element or simple string variable is accessible using the **DIM**(vhl$,0) function. Zero is returned if the specified variable name given has not been defined. An error results if a numeric variable is specified and dimension O is selected.

The **DIM()** function is provided specifically so that subroutines can determine for themselves the sizes of arrays they receive for processing, making additional parameters for this purpose unneeded. The <*variable name*> can also be specified with pointers. For example if P points to an array, then **DIM**(*P) returns its dimension count. If P points to a scalar variable or an array element, then **DIM**(*P) returns zero (i.e., no dimensions).

ERRTYP

Returns the error code of the previous error encountered and ranges from O to 255. See the **ERRSET** statement in Chapter 6, Section 4 for additional information. Appendix A lists all the MegaBasic error messages and error types.

ERRMSG$

Returns the error message of the most recent error encountered. This is especially useful for informing the user of errors that the program has trapped. See the **ERRSET** statement in Chapter 6, Section 4 for additional information. Appendix A lists all the MegaBasic error messages and error codes.

ERRPKG$

Returns the name of the package (in string form) in which the most recent error occurred. See the **ERRSET** statement in Chapter 6, Section 4 for additional information.

ERRLINE

Returns the line number of the line on which the most recent error occurred. **ERRLINE**(1) returns the relative statement number on the line where the error occurred. See the **ERRSET** statement in Chapter 6, Section 4 for additional information.

ERRDEV

When an error occurs when using some **OPEN** file or device, your program may need to know which device was involved in the error. **ERRDEV** always returns the channel number of the device most recently in use before or when the error occurred. For example if you invoke the file size function: **FILESIZE(l9)** and no file is **OPEN** under file number #19, a *File Not Open Error* will occur. If your program traps this error, **ERRDEV** can be invoked to return the offending file number, in this case a value of 19.

ERRDEV is only of use when you trap errors using the **ERRSET** statement, because your program will immediately terminate if any error occurs and no error traps are in effect. See the **ERRSET** statement in Chapter 6, Section 4 for additional information.

INDEX

Returns secondary information produced as a side-effect after invoking a number of other MegaBasic statements and functions. A brief description of these secondary results is given below, but for further details you should consult the complete documentation of the operations it involves.

VAL( )	String position in the string argument supplied to **VAL( )** just beyond the number extracted by **VAL( )**.
MIN, MAX	Argumentsequence number of the value selected.
MIN$, MAX$	Argumentsequence number of the string selected.
MATCH( )	Numberof successful matches encountered before failing. If match found, **INDEX** returns the same value as the **MATCH( )** functionitself.
FIND( )	Numberof successful matches encountered before failing a repetitive search, or the insert-position after a **FIND ORD** failure, or relative substring position after a successful search.
INCHR$( )	The string position of the terminating character in the input break character set argument.
Vector Operations	Running operation counter. After a vector operation, **INDEX** returns the number of operations performed.

If you intend to use **INDEX** after some operation, use it right away or store it in a variable because of the likelihood that it may be altered by a subsequent operation (i.e., one of the above) before you use it.

INP(<port number>) or INP$(<port number>)

Returns the value of an input data byte from the hardware port number specified, which must be within the range from 0 to 65535. No status is examined and whatever data byte is present is returned immediately. The value returned (0 to 255) is always in integer mode, never in floating point. For the fastest response times, you should specify the port number using an integer expression. **INP$()** returns the input value as a *one-byte string* instead of an integer so that it can be used in a string context.

FREE(<numeric exprn>)

Returns a variety of memory resource statistics, as selected by the numeric argument:

0	Maximum number of bytes available for the next new string variable. The quantity is the lesser of the total memory free and 65520. **FREE(0)**, or equivalently FREE, is useful in computations that determine how big to make scalar string sizes.
1	Total memory bytes currently available for more code and data.
2	Maximum bytes remaining for internal scratchpad use, i.e., expression evaluation, looping and subroutine control local arguments, etc. **FREE(2)** ranges from 0 to approximately 52000. If your program exceeds this internal space, a *ScratchpadFull Error* will result, which may be trapped as a type 13 error via an **ERRSET**.
3	Number of unused bytes remaining in the global symbol table, which manages all the symbols over all packages in a running MegaBasic application. When full, new variables cannot be created and your program stops with a *Too Many Symbols Error*. When **FREE(3)** becomes less than about 1000 bytes, you are in danger of running out of space.
4	Returns the number of unused memory segments, which if zero means that anything that needs another segment will abort your program.

EXAM(<memory address>)

Returns the value of the memory byte at the address specified by the argument expression. Note the distinction between this and the **EXAM** statement (Chapter 7, Section 3). The memory address may be specified as a *segment:offset* pair as described in Chapter 7, Section 3.

SEG(<variable name>)

Returns the actual 80x86 segment address of either the default segment setup by a **SEG** statement (if no argument is specified), or of a variable specified to the **SEG()** function. If supplied, the variable must be given as a variable name, unadorned by indexing or subscript specifications. This is useful for accessing the memory allocated to variables with the **FILL**, **EXAM**, and **CALL** statements.

The addresses of MegaBasic variables may change during program execution as a side-effect of certain operations: **DIM**, **LINK**, **ACCESS**, **INCLUDE**, or even expression evaluation. Therefore *do not* assume static locations and pick up a fresh copy of the address *just before* using it.

[<*variable name*>]

Returns the memory *offset address* of the variable specified, which may be a string or numeric scalar or array element. Offsets of integer or string variables refer to the first byte of the given integer, string or indexed string variable. Integer values are physically stored in memory low-byte first. Offsets of floating point variables refer to the sign-byte of the value, which is the last byte of the number. Variable offsets can be used in **CALL** statements for passing pointers to data to be processed, or in **FILLing** or **EXAMining** their memory contents directly from your program.

[V] returns only the offset portion of the two-component 80x86 cPu memory address. The segment portion is available as a default segment using the **SEG** statement (Chapter 7, Section 3), and from the **SEG()** function described above.

ENVIR$(<*name or sequence number*>)

Returns any of the **MS-DOS** environment strings by specifying the name of the environment string or by specifying the position of the string in the environment string list, i.e., is sequence number (e.g., 1 returns the first string, 2 returns the second, and so on). The environment is a list of strings each of the form: **<*name*>=<*string*>.** *A* null string is returned if no environment string corresponds to the sequence number or name specified, or if the host operating system is not **MS-DOS**. **ENVIR$()** returns only the <*string*> portion of the environment string accessed.

For example, **ENVIR$("PATH")** returns the directory search path string, and **ENVIR$("COMSPEC")** returns the full path name of the command shell file. These names must be fully spelled out in upper case with no spaces, equals signs (=) or other extraneous characters. **ENVIR$()** returns a null string ( ) if the specified name string does not exactly match any existing string <*name*>.

Environment strings contain information about alternate sub-directory search paths and other information communicated to all programs. The **MS-DOS SET** command is used to build this set of environment strings and display them on the console. Your program can then use these strings to select the directories from which files are accessed, and base its decisions on the prevailing parameters provided in the environment. See the **SET** and **PROMPT** commands in your **MS-DOS** operating system manual for further information.

ENVIR$(0) returns the pathname of the main program of a MegaBasic application. This makes it possible for an application to access files and packages from the same directory that the main program was loaded from. You can also use it to alter program behavior based on the actual name of the main program.

There are some key differences between the *interpreted* and *compiled* versions of **ENVIR$(0)** that you may need to know. The interpreters (**BASIC** or **RUN**) always return a fully-qualified pathname (i.e., with the drive, directory path, name and extension) of the actual main program file. If a **LINK** statement is executed, the new main program brought in by the LINK will then be returned by **ENVIR$(0).**

On the other hand, from compiled programs **ENVIR$(0)** returns the pathname of the original *.exe* file executed from the command shell. If you run .go files with the runtime libraries explicitly, **ENVIR$(0)** will return the pathname for the runtime library file instead of the compiled main program. Furthermore the compiled **ENVIR$(0)** result is unaffected by subsequent program **LINKS**, while under the interpreter it is.

PARAM(<parameter number>)

Returns an internal MegaBasic condition selected by parameter number, ranging from 0 to 25. Each Param is explained below. Certain Params may be set with an assignment statement, such as: Param(1)=0. Such Params are shown marked with an asterisk (*). Additional Parameters may be added to this list from time to time.

<table>
<tr><th colspan="3">Internal Control Parameters</th></tr>
<tr><td>0</td><td colspan="2">Returns the MegaBasic version number. This value is useful when developing Mega Basic programs which may be run under different versions of MegaBasic. Such programs can take advantage of special capabilities only when they are available and branch to different routines when not.</td></tr>
<tr><td rowspan="3">1*</td><td colspan="2">Lets you control the current state of program interruptibility: 1 for disallowed or 0 for allowed. The interruptibility state is local to each package, i.e., its current setting affects only the program in the workspace from which it is read or set. Commands and direct statements are always interruptible. Setting PARAM(1) to negative values controls the method used for program interruption: Ctrl-C (–2), which consumes input type-ahead, or Ctrl-Break (–1), which preservestype-ahead.</td></tr>
<tr><td>O - Enables interruptibility</td><td>-1- Selects Ctrl-Break</td></tr>
<tr><td>1- Disables interruptibility</td><td>-2 - Selects Ctrl-C</td></tr>
<tr><td>2*</td><td colspan="2">Returns the current default drive number: 1=A, 2=B, and so on. This is the drive implied by file names that do not include a specific drive reference. You can set PARAM(2) to any valid drive number. A disk reset is performed each time PARAM(2) is assigned.</td></tr>
<tr><td>3*</td><td colspan="2">Returns the current default I / O device (normally 0) used whenever an optional channel number is omitted. Has no effect on the console messages displayed by MegaBasic (Ready, error messages, etc.)</td></tr>
<tr><td>4</td><td colspan="2">Returns the prevailing floating point precision of the MegaBasic version you are unning under. BCD versions return 8,10,12,14,16 or 18 and IEEE/8087 versions return 2 (for double precision) . Any particular version of MegaBasic supports only one floating point precision (i.e., it is not configurable).</td></tr>
<tr><td rowspan="10">5</td><td colspan="2">Returns a code specifying the operating system environment under which MegaBasic is executing. The following codes have been assigned:</td></tr>
<tr><td>0 = North Star DOS</td><td>7 = TurboDOS-86</td></tr>
<tr><td>1 = CP/M80</td><td>8 = TurboDOS-80</td></tr>
<tr><td>2 = APC’s MTOS</td><td>9 = ConcurrentCP/M</td></tr>
<tr><td>3 = CP/M-86</td><td>10 = Convergent TechnologyDOS</td></tr>
<tr><td>4 = North Star HDOS</td><td>11 = Xebux-286</td></tr>
<tr><td>5 = MS-DOS</td><td>12 = GE PCM</td></tr>
<tr><td>6 = MP/M-86</td><td></td></tr>
<tr><td colspan="2">Like Param(0), this value is useful for writing system independent programs.</td></tr>
<tr><td>6</td><td colspan="2">Not currently defined.</td></tr>
<tr><td>7*</td><td colspan="2">Permits access to the ASCII code used to initialize strings and string arrays. At startup, Param(7) = 32, the ASCII code for spaces. You can revise this value to any code from 0 to 255 with an assignment statement: Param(7)=0.</td></tr>
</table>

Internal Control Parameters	
8*	Setting to a non-zero value (e.g., 1) disables all subsequent *epilogue* execution. In this state, *epilogues* invoked by **DISMISSes** as well as by program termination or a **LINKing** to another program are *not performed*, but execution proceeds as if they were performed. Only *epilogues* in *MegaBasic code* are affected; *epilogues* in *assembler packages* are still executed. **PARAM(8)** does not remain non-zero for very long: it is cleared by any MegaBasic error and by beginning program execution using either **RUN** or **LINK**. **PARAM(8)** can also be cleared by setting it to zero directly. This helps in situations where serious failures detected within large MegaBasic applications can be cleaned up and execution either restarted (via **LINK**) or aborted without causing additional problems that could result by allowing *epilogue* execution to proceed after a catastrophic error.
9*	Redefines the end-of-file code and affects the operation of **READ**, **WRITE**, **TYP()**, **PRINT** and **INPUT** file operations. Any value from 0 to 255 may be assigned, but only values 0-1, 26, and 154-255 should be used to avoid conflict with the encoding of strings and floating point values on files (26 is useful for text files).
10*	Returns or changes the number of file buffers available during file operations and may be set to a value from 4 to 127. Open files are not affected by setting Param(10) and need not be closed first. If you attempt to assign more buffers than 128 or than available memory permits, only the maximum possible will be allocated. File buffers are 512 bytes each and their number is controlled solely via Param(10).
11*	Returns or changes the floating point format type that is assumed by **READ** and **WRITE** statements during file transfers. Param(11) is normally set to the *native* floating point format of MegaBasic (see Param(4) above). You can change Param(11) to any format under the IEEE / 8087 version (1=single IEEE, 2=double **IEEE,** or 8 to 18 **BCD**), or to any **BCD** format under **BCD** versions (8 to 18 **BCD**). This feature permits access to files written by versions of any precision. Shorter values are padded with trailing zeros and longer values are rounded to the prevailing precision. All file transfers use the precision specified by Param(11) until subsequently changed.
12*	Returns and sets the maximum length of string variables subsequently created by default, without being defined by an explicit **DIM** statement. Any maximum size from –1 to 4095 is permitted, although remember that that much memory will be held by such variables until re-defined by a later **DIM** statement, if any. Param(12) is initially set to 80. Programs that do not rely on default strings can benefit by setting Param(12) to –1 to disable the automatic string creation feature of MegaBasic. This is useful to eliminate the possibility of variables created as a result of misspelling their names or of forgetting to **DIMension** strings properly. With this in effect, attempting to access an undeclared string will result in an *Undeclared String Or Array Error*, alerting you to their presence and indicating an error in your program.
13*	Returns and sets the upper limiting subscript boundary for default arrays. Normally a reference to an array which has not been previously accessed causes a new (default) array to be created automatically which has one dimension and subscripts ranging from 0 to 10. Param(13) allows you to control the upper boundary of default arrays subsequently created. The value assigned must be an integer from –1 to 1023. Param(13) is initially set to 10. Programs that do not rely on default arrays can benefit by setting Param(13) to –1 to disable the automatic array creation feature of MegaBasic. This is useful to eliminate the possibility of variables created as a result of misspelling their names or of forgetting to **DIMension** arrays properly. With this in effect, attempting to access an undeclared array will result in an *Undeclared String Or Array Error*, alerting you to their presence and indicating an error in your program.

<table>
<tr><th colspan="3">Internal Control Parameters</th></tr>
<tr><td>14*</td><td colspan="2">Returns the current number of bytes per file block, used to scale the result returned by the FILESIZE() function and the SPACE() function. Under CP / M-type operating systems, the standard block size used by MegaBasic is 256 byte / block; under MS-DOS systems the block size is 1 byte / block. If you are designing generic programs intended to be run under any operating system, Param(14) should be set to some consistent value (e.g., 1) so that your program can ignore this system dependency.</td></tr>
<tr><td>15</td><td colspan="2">Returns the maximum number of files and / or devices that may be OPENed under MegaBasic. Normally, you may OPEN up to 32 files (under file numbers 0 to 31). But the CONFIG utility program can alter this limit for particular copies of MegaBasic to any value from 8 to 128 for specific applications. This limit cannot be changed by a running program, but your program can determine the limit by reading Param(15).</td></tr>
<tr><td rowspan="5">16</td><td colspan="2">Returns a code indicating the kind of MegaBasic that is running your program. The following values are returned:</td></tr>
<tr><td>0 - MegaBasic development version</td><td>3 - PGMLLLINK under the run version</td></tr>
<tr><td>1 - Run-only version of MegaBasic</td><td>4 - Stand-alone, compiler program</td></tr>
<tr><td>2 - PGMLINK under the development version</td><td></td></tr>
<tr><td colspan="2">This information is useful for dependencies on whether the execution of the program is for a develop / test / debug session, or under the ultimate production environment (e.g., see EDIT$). Normally, you will develop and checkout your programs under the development version, while finished programs will be run under the run-only environments provided by RUN or PGMLINK.</td></tr>
<tr><td>17</td><td colspan="2">Returns the largest amount of scratchpad space used so far during the current MegaBasic session. This value is useful in determining the resource requirements of a given program so that its viability in different systems can be ascertained. Param(17) may at times be several thousand bytes higher than the true value because it is updated only when more physical memory is allocated to the scratchpad segment.</td></tr>
<tr><td>18*</td><td colspan="2">Lets you force file and directory pathnames under the Xenix 286 operating system to all upper or lower case. Under Xenix, such names spelled with different letter case but otherwise the same will refer to different files. Param(18) set to zero (its initial value) allow names to be passed exactly as-is to Xenix. Setting Param(18) =1 forces all names to lower case and Param(18)=2 forces them to upper case. This is particularly useful in programs written under non-Xenix MegaBasic if they are being ported to Xenix. Param(18) has no effect under any other operating systems currently supported.</td></tr>
<tr><td>19</td><td colspan="2">Returns the error code reported by the most recent operating system call made by MegaBasic that reported an error. After executing a shell command with the DOS statement, Param(19) returns the exit code returned by the shell command. Under MS-DOS, all INT 24h traps and most INT 21h calls that return errors (but not all) can be determined with Param(19) . All errors reported by CP / M type operating systems (including TurboDOS, MP / M86, CCP / M, etc.) are returned by Param(19).</td></tr>
<tr><td>20*</td><td colspan="2">Returns a non-zero value to indicate that hardware floating point support is currently available for speeding up math operations. Returns zero to indicate all math operations are done using only software methods. You can disable use of the floating point hardware by setting Param(20) to zero, and re-enable it by setting Param(20) =1. If no math support is installed on the host machine, then Param(20) remains zero, no matter what you set it to. Only the IEEE version of MegaBasic supports floating point hardware; the BCD version does not.</td></tr>
</table>

Internal Control Parameters	
21*	Returns and sets the high-level error reporting state of the current MegaBasic package. Normal low-level reporting is 0, high-level reporting is 1. This mode affects the reported error location of any error that, untrapped by an **ERRSET** statement, terminates the program. Normally, MegaBasic reports the exact line number and package name where such an error occurs. This is low-level reporting mode, where Param(21) = 0. Errors that occur within a package set to high-level mode (Param(21) = 1) are not reported as errors in that package. Instead, they are reported as if they occurred in the most recent, outer subroutine call reference within a package in low-level mode (i.e., where Param(21)=0). The purpose of this is to further hide the implementation details of a package from the user of a package. Param(21) must be set by the package itself, such as in its **PROLOGUE** routine. Such a package is then free to generate *errors* in response to improper arguments and other conditions, so that they are reflected as errors in the use of the package instead of appearing to be *package bugs*.
22*	Enables or disables file and record locking operations, meaningful only under **MS-DOS** systems that provide network support. Setting **PARAM(22)** to 0 disables file / record locking, 1 enables both network operations and automatic locking, and –1 enables network operations *without* automatic locking. See Chapter 7, Section 2 for details about MegaBasic file locking.
23	Returns the host microprocessor type as a numeric code: O for 8086 / 88, 1 for 80186 / 88, 2 for 80286, 3 for 80386 and 4 for 80486. When you first begin your MegaBasic session, MegaBasic determines the type of microprocessor currently executing MegaBasic and displays it in the sign on message. This information is not currently used for any other purpose, but future versions of MegaBasic may use it to optimize certain operations.
24*	Sets the size of the most-recently-input line list. This buffer defaults to 512 bytes, but you can change its size to any value from 0 to 4096 bytes by setting Param(24) to the desired size at any time. Setting the buffer size to zero disables the previous line list capability altogether (except for the standard *old line* buffer). Setting Param(24) always clears the buffer of all lines, except for the most recently entered line. Defining a larger or smaller buffer size causes the total available memory space to decrease or increase accordingly. For more information, see the line editor discussion in Chapter 1, Section 6 and the **EDIT$** statement in Chapter 9, Section 4.
25*	Set to 1 enables a special **LINK** mode causing subsequent **LINK** statements to leave supporting packages initialized with all their variables, open files and **ACCESSes** intact. The next program still has to **ACCESS** the packages it needs, but *nothing is* automatically **DISMISSed** by the **LINK** statement in this mode. PARAM(25) can be turned *on and off at any time* during execution to affect *all subsequent* **LINK** statements executed

Chapter 10 Multiple Module Programs

Suppose that you could collect all your favorite functions and procedures and somehow make them available as additional *primitives* of the language, extending its capabilities and expressiveness accordingly. And further suppose that there was no limit on how many of these *primitives you* could add to the language, as long as the total memory resources permitted them. Given enough built-in features in any language it is easy to see that any application could be implemented by a *small* program. This is the philosophy behind MegaBasic *packages*. All MegaBasic package concepts and supporting statements are described in this section, as summarized below:

Overlay and Package Statements	Summarizesall MegaBasic statements that support and manage multiple program modules, including program **LINK** (also called **CHAIN**) and MegaBasic package statements.
Package Definition	Describes the four aspects of package definition: everything you need to know about creating a package.
Using Packages	Discusseshow to load, initialize and gain access to packages from your program. Removing from memory packages that are no longer needed is also covered.
Multi-Package Environment	Shows how to take advantage of the multiple workspace development environment provided by MegaBasic to create, test and debug packages for use in large programs.
Assembler Packages	Protocols, structures and procedures for implementing MegaBasic packages in assembler or machine code. This lets you add very high speed extensions to MegaBasic.

Although MegaBasic packages are normally written in *MegaBasic, you* can also develop packages using low-level *assembly language* as well. Assembler is the fastest possible computer language to implement software in, but it is also one of the more difficult and demanding languages to write in. MegaBasic assembler package development is an advanced capability covered in Chapter 10, Section 5. However, using assembler packages is virtually identical to using *normal* MegaBasic packages, so you need to understand the material presented below regardless of the package implementation you choose.

A package is simply a collection of useful variables, functions and procedures contained within a separate MegaBasic program file and accessible to your program as an external library. The subroutines within packages may be defined in terms of subroutines in still other packages, making it possible to implement programs of virtually any size or logical complexity. Packages that are no longer needed during execution can be made to *go away under program control,* releasing their memory resources for other uses.

When you call a general purpose subroutine, it should not be able to produce unexpected side-effects on your program. Imagine the chaos of developing a huge program where all variable, procedure and function names are accessible throughout the entire program. Indeed, such a situation exists in all **BASIC** programs and this is the major reason that **BASIC** has historically been unsuitable for implementing large complex systems.

MegaBasic package mechanisms have been designed to overcome these limitations and to provide a flexible environment for large-scale application development. MegaBasic packages let you create subroutines in such a way that all their implementation details are hidden from the program that uses them. Further, the controlled interface between MegaBasic packages greatly simplifies the development of special-purpose packages that can be easily integrated into specific systems in an independent way.

The use and definition of packages centers around a small set of *primitive operations* which, in conjunction with one another, provide all the facilities to load, access, detach and remove packages and the structures they contain all during execution. We shall begin by discussing just what makes a package.

File Lookup Order

When MegaBasic loads a program file using **LOAD**, **MERGE**, **LINK, INCLUDE** or **ACCESS**, it first looks in the directory implied by the specified file name. If it is not found there, then the standard system search order, as specified by the **PATH** environment variable. MegaBasic searches each of the directories in turn until the file is found (or not). The file is loaded from wherever the file is found first.

Instead of using the system **PATH**, you can also use the **MBPATH** environment variable to specify an alternate file lookup path-set for MegaBasic program files. It works exactly like the system **PATH** command that you already set up in your **AUTOEXEC.BAT**, except that you set it up it with the system **SET** command (see your **MS-DOS** manual for details). If **MBPATH** is defined, it is used instead of the system **PATH**; if is is not defined, then the system **PATH** is used. Separating the MegaBasic lookup order from the system's can speed up the loading of MegaBasic modules while reducing the size *of* the system **PATH.**

Section 1: Overlay and Package Statements

The statements covered in this section are used for combining programs into larger programs which may, during execution, reside totally in memory, or partially in memory and partially on disk files. A single program *module is* limited in MegaBasic to a maximum of about 75,000 bytes, enough space for up to 3 or 4 thousand statements. However, up to 64 separate programs can *co-exist* in memory, limited only by the total memory installed in your machine and made available to MegaBasic. Under program control, such program modules, called *packages,* can be brought into memory and access relationships established among them, permitting each package to *access the subroutines and data of others* in a controlled manner. The statements discussed in this section are summarized as follows:

DEF SHARED...	Defines functions, procedures variables and fields that will be accessible from other packages in memory and declares their types.
ACCESS	Activates MegaBasic packages and establishes the relationships between them.
DISMISS	Deactivates MegaBasic packages, breaks the interconnections with between packages and releases packages no longer in use by the program.
INCLUDE	Loads and activates MegaBasic packages *without forming access* relationships between them.
LINK	Primitive method for terminating the current program and starting another. Also known as **CHAIN** in some Basics, **LINK** is intended for compatibility with older programs that use it (and not encouraged for use in new programs).

Access between packages is limited to named *procedures, functions* and *variables,* all of which must be explicitly declared as **SHARED** in order to be externally accessible. Subroutines and variables that are not declared **SHARED** can only be accessed from within the program in which they are defined, completely hidden from the view of all outside packages.

This multiple program model is simple to use, yet more powerful and general purpose than the **CHAINing** facilities of other **BASICS** (and of MegaBasic). We highly recommended that **CHAINed** programs be upgraded to the package model and to *avoid* **LINK** statements in new programs. In so doing, future enhancements to your programs will be much easier to implement and programming side-effects from major changes can be minimized and controlled.

DEF SHARED *<function or procedure definition>*

Defines a function or procedure as **SHARED**, meaning that outside packages which can **ACCESS** the package containing this definition, may freely use the function or procedure within their own statements as needed. This statement merely represents the inclusion of the **SHARED** indicator in the **DEF** statement described back in Chapter 8, Section 1, which you should thoroughly read and be able to apply in single-package programs before attempting in multi-package programs. Another kind of **DEF** statement for variables (Chapter 5, Section 1) can affect the numeric type of **SHARED** functions, which

you should also understand. See Chapter 10 for additional information about developing large multiple-module programs.

DEFSHARED <*list of variable names*>

Declares that each of the variables listed will be accessible to outside packages having access to the current package, as granted by the **ACCESS** statement described later on. The list of variable names consists of one or more identifiers separated with commas. Names which will be used as arrays *must* be followed by empty parentheses ( ) to indicate this intention. This statement does not allocate any storage to these variables at this time. An example of this statement is shown below:

```
DEF SHARED VECTR( ), X, Y, ARRAY$( ), integer Z( )
```

This statement is really just an extension of the data type declarative **DEF** statement (Chapter 5, Section 1). You should refer to that discussion for important additional options that can be used to declare **SHARED** variables as string, integer or real variables. **DEF** statements are not executable but processed just before the program begins execution, and their order can affect the data type of **SHARED** variables.

INCLUDE <*list of package names*>

Brings a list of program files (containing packages) into memory for later execution. Package file names are specified with string expressions and separated with commas. An error results if any file cannot be found. Packages specified which are already in memory are ignored and remain in memory, otherwise, as each file is **INCLUDed**, the following sequence of actions is performed:

- The program file is loaded off the disk into memory.
- Its program **DEF** statements are all processed and the definitions are made available locally, but shared entities are not bound externally at this point.
- Its *Ctrl-C enable/disable* state is set to the same state as in the program executing the **INCLUDE**. **PARAM(1)** may be used to subsequently control this setting from within the package itself.
- Its *prologue procedure* is invoked if one has been defined. On completion of the prologue it returns to the **INCLUDE** statement which then resumes **INCLUDing** more files. A *prologue is* an ideal place for additional **INCLUDE** and **ACCESS** statements if the package being **INCLUDed** requires additional packages for its operation.

The sequence in which package files are **INCLUDed** is important because the prologues are executed in that order. Further, this order also controls the sequence in which epilogues are performed when the program **ENDS**. The **ACCESS** statement (below) performs an implicit **INCLUDE** operation on every package it deals with, hence **INCLUDE** statements are useful primarily to control the order of subsequent prologue and epilogue execution.

Although in memory, packages cannot access anything defined in other packages until access has been explicitly granted with an **ACCESS** statement (described next). **INCLUDE** is not generally a frequently needed statement because **ACCESS** always implicitly **INCLUDES** packages not already memory resident.

ACCESS<*package list*> [FROM <*package list*>]

Establishes access to the shared entities defined within each package specified in the first list of packages. If the optional **FROM** clause is omitted, then the package executing the **ACCESS** statement is granted access to the list of packages. Package lists consist of one of more package file names, specified as string expressions, separated with commas. If any specified package name is not present in memory when an **ACCESS** statement is invoked, MegaBasic will **INCLUDE** it automatically from its disk file. Once **ACCESSibility** has been established, further identical **ACCESS** requests are ignored, i.e., redundant **ACCESS** statements are not an error and do nothing. As each package is loaded into memory, the following steps are performed internally:

- Its program **DEF** statements are all processed and the definitions are made available locally.
- Sets its *Ctrl-C* enable / disable state to the same state as in the program executing the **ACCESS**. **PARAM(1)** may be used to subsequently control this setting from within the package itself.
- Executes the *prologue* procedure within the package if one has been defined and not yet been executed by an earlier **INCLUDE** or **ACCESS**. A prologue is an ideal place for additional **ACCESS** statements if the package being initialized requires additional packages for its own operation.
- Makes all **SHARED** names of the package available to the **ACCESSor**. This process, known as *binding*, *is* only performed by an **ACCESS** statement and never by an **INCLUDE** statement. Only those **SHARED** names that actually have references are bound. This is the only step performed if the package was already brought into memory by an earlier **ACCESS** or **INCLUDE** statement.

Your program or package may use any **SHARED** entities that belong to other packages that have been **ACCESSed**, just as if they had been defined directly within your program. Your programs can then be written using much higher level constructs than simply the built-in primitives provided in your language. The details of these constructs are hidden from the view of your program, greatly simplifying your programming design and implementation tasks.

DISMISS <*pkg name list*> [FROM <*pkg name list*>]

Severs external access to each of the packages in the first list from each package listed in the second optional **FROM** list. The package executing the **DISMISS** statement is assumed in the absence of a FROM list. When a **DISMISSed** package becomes **inACCESSible** from all packages, the variables created by it along with its program source lines are erased and the memory held by them is made available for subsequent reuse.

Names of *inactive* or *not-present* packages are ignored. The first list of package names is the list of packages to be **DISMISSed**. The optional second list (the **FROM** list) specifies the packages from which the first list is **DISMISSed**. By omitting the **FROM** list, the packages are **DISMISSed** from the package invoking the **DISMISS** statement. If the **FROM** list is specified as an asterisk (*), the first list of packages is **DISMISSed** from *all packages.*

The **DISMISS** statement does the reverse of an **ACCESS** statement: **ACCESSible** packages become **inACCESSible**. When a package is no longer **ACCESSible** from *any package*, it is automatically removed and its memory space is released for reuse elsewhere. One **DISMISS** statement can *break* the **ACCESSibility** of (potentially) many pairs of packages. The following sequence of operations is performed for each (implied) pair of **ACCESSed/ACCESSor** packages:

- All references (access) to the **ACCESSed** from the **ACCESSor** are *bro*ken. Subsequent reference to these in the **ACCESSor** package will *be* treated as new default variables.
- If the package is still **ACCESSed** from other packages no further action is performed, i.e., finishing this **DISMISS**. See the **ACCESS** and **DISMISS** *functions (Chapter 10, Section 3*) for an important exception to this.
- Once the package is **inACCESSible** from all packages, its *epilogue routine* (if present) is executed. This is a good place to close working files and perform any clean-up procedures necessary.
- All its data, **SHARED** or otherwise, is released back into the system for subsequent reuse for other purposes. The package source is *marked free*, rather than actually released. If the occupied memory is suddenly required by some activity, MegaBasic releases *freed* packages. But if the package is later **INCLUDEd** or **ACCESSed** before actually released, it is already resident and no physical program load from the file is done. *Already freed* packages can be *forced out by* using the **FREE** statement (no arguments). Packages that were either manually **LOADed** or contain **unSAVED** changes are *protected from erasure* to simplify the development and testing cycle.

If you fail to explicitly **DISMISS** a package, it will stay in memory with all its *variables* and *arrays* intact until the program terminates.

LINK *<program name exprn> [,<common vbls>]*

Terminates the current program, releases it, loads another program specified by the *<program name exprn>*, then begins execution on the first line of the new program. All files are closed and data stored in variables may or may not be lost, depending on the *<common variables>* portion of the **LINK** statement. **LINK** thus provides a means for MegaBasic programs to automatically **LOAD** and **RUN** program segments of their own choosing.

Variables may be passed between **LINKed** programs by listing their names after the program file name expression in the **LINK** statement. Any type or size of variable may be passed as long as space in the **LINKed** program permits. For example the statement: ***LINK'"PGM'',X,Y,B$,V()*** will **LINK** to ***"PGM"*** and pass **X,Y,B$,V()** to it, where ( ) indicates that V is an array. Syntax errors result from expressing computations in this list.

To preserve all variables, use an at-sign (@) instead of the variable list. For example ***LINK"PGM',@ will*** pass all variables to ***"PGM"***. With this method, any files **OPEN** before **LlNKing** will still be **OPEN** when the **LINKed** program begins. Only the variables that have references in both programs are passed by the at-sign **LINK** statement. Therefore, as the system **LINKS** from program to program, only variables common to all programs are preserved throughout execution.

The **LINK** statement physically *frees all* packages in memory as it chains from the current program to the next program (including the old program just left). These remaining packages are neither active nor **ACCESSed** by anyone, but subsequent **ACCESSes** or **LINKS** to them will execute quickly because they *don't have to be re-loaded* from the disk again, similar to the **DISMISS** statement operation. To *force* a program to be loaded from the disk, you can *flush all inactive packages* from memory by executing *a* **FREE** statement *(no arguments)* just before executing the **LINK**, **ACCESS** or **INCLUDE**.

If the target **LINK** program is already in memory, it too *is* executed without being re-loaded from the disk. If the **LINK** program does have to be loaded from the disk, MegaBasic requires enough free memory to hold the new program without first freeing the program executing the **LINK**.

To further support systems that **LINK** from program to program, each of which **ACCESSing** many of the same supporting packages, a special **LINK** mode can be enabled by *setting* ***PARAM(25)*** *to 1*. This causes subsequent **LINK** statements to leave supporting packages initialized with all their variables, open files and **ACCESSes** intact. The subsequent program still has to **ACCESS** the packages it needs, but *nothing is* automatically **DISMISSed** by the **LINK** statement in this mode. Variables may be passed to the next **LINK** program the usual way (i.e., listed individually or @ for all variables). **PARAM(25)** can be turned *on and off at any time* during execution to affect *all subsequent* **LINK** statements executed.

Section 2: Package Definition

A package is just a program stored in a file that you develop just as you would any other program. In addition, it contains a few declarative statements for making some of its defined objects externally known and accessible. If you already have an ordinary MegaBasic program, you can turn it into a MegaBasic package by understanding and applying the following four logical components of a MegaBasic package:

SHARED Objects	Variables,functions and procedures that are made available for reference by other programs. Line numbers and line labels (and hence **GOSUBS**) are not sharable.
PROLOGUE	An optional initialization routine, called a prologue, which is executed automatically upon loading the package.
EPILOGUE	An optional clean-up routine, called an epilogue, which is executed just prior to removing an existing package from memory, or when the main program terminates.
Programming Details	The main body of MegaBasic statements that implement the intended operations of the various subroutines. These details are *hidden* from *users* of the package.

SHARED Objects

It would greatly reduce the power and effectiveness of packages if everything they contain was always externally accessible. Therefore, subroutines and variables within packages are not externally accessible unless they have been specifically declared as **SHARED** objects. To declare functions and procedures as externally accessible, insert the word **SHARED** in their corresponding **DEF** statement, for example:

```
DEF SHARED FUNC CUBE(X)=X^3
  or
DEF SHARED PROC SORT TBL
all the procedure implementation details
RETURN; PROC END
```

Without defined as **SHARED**, these subroutines would be available for use only from within the package they are defined. To declare **SHARED** functions as having a string, integer or real type, you should place the word **STRING**,

INTEGER or **REAL** preceding the **FUNC** reserved word, as described in Chapter 8, Section 1. To share *variables* with external programs, each variable must be declared in a **DEF SHARED** statement:

```
DEF SHARED <variable list>
```

The <*variable list*> consists of variable names separated with commas. You may have as many of these statements as you need to list all the desired **SHARED** variables and they may be located anywhere in the package program. This declaration is necessary only for the package that *owns* the data. See the discussion in Chapter 5, Section 1 for further information on **DEF** statements. External programs refer to **SHARED** variables just as if they were defined *locally.*

Do not declare the same name in two different packages as **SHARED**, or a *Shared Name Conflict Error* will occur when one package accesses the other or when both are accessed by a third package. This error also occurs when an external name is already defined locally for another purpose. Declare **SHARED** objects just once in the package that *owns* them. Then, any package that requires access to the name can **ACCESS** (Chapter 10, Section 1) the package that *owns* it.

Line-numbers and line-labels and **GOSUBS** *cannot* be **SHARED**. Subroutines of the **FUNC** or PROC variety are the only program objects that may be executed from another program. Specifically, this means that a program cannot jump into another package, except as part of a *call and return* sequence.

An important application for **SHARED** variables is implementing a set of data which is available for reference throughout the entire program (i.e., by all packages). Known in other languages as **GLOBAL** or **COMMON** data, this type of access is frequently required when developing a large program. Truly global data should be kept in one specific package that contains nothing but global objects (variables and / or subroutines), rather than along with other **SHARED** objects which are not *really* global (i.e., needed by only a subset of the system). By collecting all *global* variables and subroutines into one package, they are easier to manage, control and access by all other packages. See the **ACCESS** statement (Chapter 10, Section 1) to see the various ways for making **SHARED** data externally available.

Prologue and Epilogue Routines

These two *optional* subroutines may be specified to perform *automatic* initialization and clean-up operations when packages are *initially loaded* (e.g., on its *first* **ACCESS**) and subsequently *released.* Hence the outside users of a package require no knowledge of these necessary but arbitrary and sometimes messy details. Packages can then be sufficiently autonomous to be independently developed, simplifying large-scale program development activities.

What the *prologue* and *epilogue* do is *completely up* to the programmer. A typical example is a database service package which requires various open working files and initialized data structures before its subroutines will function properly (using a *prologue).* Then, just prior to program termination or its release from memory, it updates and closes its working files (using the *epilogue).* Another important application for *prologue/epilogue* constructs is the loading and subsequent release of additional packages. More on that subject later.

To define a *prologue* or *epilogue* **GOSUB**, simply place the line labels **PROLOGUE** or **EPILOGUE** in front of the first statements of the **GOSUB**. These labels must be spelled exactly as indicated and may not be used for another purpose. Be sure that the source code that defines each **GOSUB** ultimately terminates by executing a **RETURN** statement. The absence or presence of these line-labels in your program determines the absence or presence of a *prologue/epilogue.*

The **PROLOGUE** and **EPILOGUE** mechanism is designed to satisfy the initialization and clean-up processes needed by a package in a convenient and general way. However they are optional to give you the freedom of controlling the manner in which your particular program will operate. It is entirely possible for initialization and clean-up processes to be implemented using **SHARED** procedures which are called explicitly by the **ACCESSing** package at the appropriate times.

The *prologue* of a package is executed only once when the package is initially loaded and never again (unless it is released and later re-loaded). The *epilogue* of a package is

executed only after all accesses to its package have been released using **DISMISS** statements. An *epilogue will* not execute until the *prologue* (if any) has successfully completed and returned, so that if the *prologue* aborts due to an error, the epilogue will *never* execute.

PARAM(21) controls how errors are reported in packages. If subroutines in your package detect errors in the way they were called, it can be desirable to report such problems as errors *in the use of* the package. By *setting* **PARAM(21)** *to 1* (see Chapter 9, Section 5) in your package **PROLOGUE**, *errors will be* reported as errors in the *calling reference* to the subroutine (i.e., in the *calling* program), instead of in the package itself.

Package Programming Details

The implementation details within a package are no different from an ordinary program. It consists of whatever variables, **GOSUBs**, functions, procedures and any other things that are necessary to implement the package. Typically, only a small percentage of all the objects within a package are made *visible* as **SHARED** objects, while all the rest are used for internal implementation. The main point here is that, for the most part, a package is just an ordinary program module.

Section 3: Using Packages

Once packages are defined, they may be used by any program which requires them. To use a package, its program file must reside in memory along with the program that will be using it, and its **SHARED** objects must be known by that program. Since memory is generally a scarce resource, packages that have served their purpose and are no longer needed can (and should) be removed from memory to make room for other packages or data. The various facilities that provide these functions will now be described.

Accessing Packages

In order to make the SHARED contents of a package available to your program, two actions must be taken. First, the file containing the package must be loaded into memory and initialized. Second, all of the **SHARED** names it defines must be made accessible to your program, so that when you refer to one of these names, the external *object* associated with this name is accessed (instead of creating some new, local program variable). This entire process can be done by the following statement:

```
ACCESS <package name>
```

where *<pack age name> is* specified as a string expression that evaluates to the file name of the MegaBasic package. There are other options on the **ACCESS** that we will describe shortly, but first, let's explore what happens when MegaBasic executes the **ACCESS** statement above.

First the package file name is located in the directory and the program is loaded into memory. An error results if the file cannot be found. If the package is already present in memory, MegaBasic saves time by not reloading it from the disk. After loading a package and initializing its **DEF** statements, The **ACCESS** statement then *binds* every name in your program that matches a **SHARED** name in the package to the *object* having that name. Only those **SHARED** names that actually *have references* are bound. If any of the matching names are already associated with some variable or subroutine in your program (i.e., they are already defined), then the **ACCESS** fails and MegaBasic reports a *Shared Name Conflict Error* along with the name and packages involved. Finally, the *prologue procedure* within the package is invoked if one has been defined. A *prologue* is an ideal place for additional **ACCESS** statements if the package being initialized requires additional packages for its own operation.

This finishes the **ACCESS** statement, and execution continues to the next program statement following it. In summary, an **ACCESS** loads, initializes and connects a package to your program or other package. It creates only a one-way connection, so that the **SHARED** objects are accessible in a controlled, limited way, rather than to all packages everywhere.

As mentioned earlier, **ACCESS** statements have some other options to give it more flexibility. In particular, one **ACCESS** statement can access a list of one or more packages. You can also specify another list of packages that accesses the first list. An **ACCESS** statement can take one of the following three forms:

ACCESS *<list>*	Accesses the packages listed from the current program or package.
ACCESS *<list>* ***FROM*** *<list>*	Accesses the packages specified in the first *<list>* from each of the packages specified in the second *<list>*.
ACCESS FROM <list>	Accesses the current program or package from each of the packages listed.

where *<list> is* a sequence of one or more package names separated by commas, each specified by a separate string expression. If any specified package name is not present in memory when an **ACCESS** statement is invoked, MegaBasic will load and initialize it automatically from the disk. Once **ACCESSibility** has been established, further identical **ACCESS** requests are ignored and no error reported.

Your package may use any **SHARED** objects that belong to other **ACCESSible** packages, just as if they were defined directly within your program. Thus your programs may be written using much higher level constructs than simply the built-in primitives provided in the language. The details of these constructs are hidden from the view of your program, greatly simplifying your programming design and implementation tasks.

Including Packages

The load-initialize sequence performed by ACCESS can also be performed *without binding* the SHARED names within the package to any package or program. This is done using the INCLUDE statement, which takes the form:

INCLUDE *<list of package names>*

where *<list of package names>* is a list of file names specified with string expressions, separated with commas. An error results if any file cannot be found. Packages already present in memory will not be reloaded from the disk. Package names in this and all other package statements are specified as file name string expressions. Once a package has been **INCLUDed** into memory it may be accessed by other packages using the **ACCESS** statement.

In general, **INCLUDE** is not a statement frequently needed because **ACCESS** always implicitly **INCLUDES** any packages not already memory resident. However, it can be useful in controlling the order that packages are brought into memory and initialized. This sequence can be important because the *prologues* are executed in that order. Furthermore, this order also controls the sequence in which *epilogues* are performed when the program **ENDS**. Epilogues are done in the reverse order of the load-initialize sequence. If this is not desirable, the **DISMISS** statement may be used to force the *epilogue* invocation, as discussed below.

Dismissing Unneeded Packages

The **DISMISS** statement provides a means for removing current packages to make room for **ACCESSing** others, or for reclaiming data space. This statement has the following forms, similar to the **ACCESS** statement:

DISMISS *<list1>*	Dismissesthe packages listed from the current program or package.
DISMISS<list1 > **FROM**<list2>	Dismissesthe packages specified in <*listl*> from each of the packages specified in <*list2*>.

where <*listl* > *is* a sequence of currently **ACCESSed** package names, separated with commas. The optional <*list2*> specifies the packages from which the first list is **DISMISSed**. Any package names listed that are not currently in memory are ignored. By omitting <*list2*>, all <*listl* > packages are **DISMISSed** from the package executing the **DISMISS** statement.

The **DISMISS** statement does the reverse of the **ACCESS** statement. The effect is that **ACCESSible** packages become **inACCESSible** from the program or package that previously accessed them. When a package is no longer **ACCESSible** from any package, MegaBasic automatically removes it and its variables from memory. This released memory space becomes immediately available for reuse elsewhere. One **DISMISS** statement can break the **ACCESSibility** of (potentially) many pairs of packages.

The **DISMISS** statement performs a specific sequence of operations on each **ACCESS** relationship to be broken. The sequence is performed the same way on each pair, and it consists of the steps described below. Given that program **MAIN** has previously accessed package **LIBRARY**, the following sequence of operations is performed:

- All references to the **LIBRARY** from **MAIN** are broken. Subsequent reference to these in **MAIN** package will be treated as new default variables. **MAIN** no longer **ACCESSes** this package.
- If **LIBRARY** is still accessed from other packages no further action is performed. If **LIBRARY** is not **ACCESSible** from any package, its *epilogue routine* is executed (if present). *Epilogues* are a convenient place to close working files and perform any general clean-up procedures necessary or additional **DISMISS** statements, if the package was **ACCESSing** any private packages of its own.
- All **LIBRARY** data, **SHARED** or otherwise, is dismantled and its allocated memory is released back into the system for subsequent reuse for other purposes.

DISMISS does have some restrictions, all of which stem from the need for **SHARED** functions and procedures to be able to **RETURN**. A package cannot **DISMISS** itself. If it could, then upon finishing the **DISMISS** statement there would be no program to continue with. For the same reason you can't **DISMISS** the original main program and MegaBasic reports an *Illegal Package Operation Error* if such attempts are made.

When **DISMISS** removes a package from memory, its program source image actually remains intact, but the region it occupies is marked as *free memory*. If another **LINK**, **INCLUDE** or **ACCESS** statement requests its presence before it is actually overwritten, MegaBasic will recover and reuse the code already in memory, rather than spend the time to reload the program from its disk file all over again. If your program needs memory for variables or arrays, these *free areas* are used if no other free memory is available.

Orphaned Packages

A package will *not go away* until you *explicitly* **DISMISS** it and nothing else **ACCESSes** it. Although this sounds simple, it is easy to assume that when packages are **DISMISSed**, their ***ACCESSes*** *will* magically get **DISMISSed** as well. But instead, such *orphaned* packages will just sit there in memory, with all their variables intact and their *epilogues* unexecuted, taking up memory space until they are formally **DISMISSed**. If the package is **ACCESSed** again, then it will still be initialized, so its *prologue will not be executed* and its **SHARED** variables will have whatever values they had at the time they were last modified. *Orphaned packages* show up in the **SHOW** command as *Detached.*

This behavior gives you a good deal of control over the *lifetime* of a package *and* its variables. For example, a package of global variables would normally *never be* **DISMISSed** *so* that it would remain active throughout execution regardless of whether it was actually **ACCESSed** by others at any particular instant in time.

In order to accommodate those applications that really do have to **DISMISS** all the currently **unACCESSed** packages, you can still do so by specifying a **DISMISS** statement without any arguments (i.e., ***DISMISS)***. However, avoid using this statement while inside a *prologue*, directly or indirectly, because it can prematurely **DISMISS** packages that are *in the process* of being **ACCESSed** at that moment.

A package can guarantee that it survives a **DISMISS** (including one with no arguments) by simply **ACCESSing** another package that in turn **ACCESSes** it back. Unless an outer package performs a **DISMISS** ***package*** **FROM** *, such *mutual* **ACCESSes** ensure that both packages always remain **ACCESSed** by at least one package, preventing their removal (and their *epilogue execution)* from occurring (except by the mechanism described below).

ACCESS and DISMISS Functions

It is often desirable to treat a set of packages *as if* they were one package, **ACCESSing** and **DISMISSing** one of them and allowing its **PROLOGUE** and **EPILOGUE** to bring in later release the supporting *sub-packages.* As long as there are *no loops* in the **ACCESS** relationships (e.g., two packages accessing *each other),* this is easy and natural to do. However, in more complex situations, loops in the access paths can be unavoidable and desirable. The discussion below describes the pitfalls of doing this and how to avoid them.

Consider the following scenario. Suppose **MAIN** accesses **PKG-B** and then later on dismisses **PKG-B**. Well of course, **PKG-B** *goes away* as you would expect. Now suppose that for whatever reason, we break **PKG-B** into two packages **PKG-B** and **PKG-C** that access *each other. So* we run **MAIN** and it accesses **PKG-B** and later dismisses **PKG-B**, just as before. Since **PKG-B** has itself and **PKG-C** accessing each other in its *prologue, you* would probably also have an *epilogue* that dismisses **PKG-B** and **PKG-C** from each other so that they will both go away when **MAIN** dismisses **PKG-5**.

Now for the surprise. When you run **MAIN** and it finishes accessing and dismissing **PKG-B**, **PKG-B** and **PKG-C** Will *still* be accessing each other with all their data intact and

no epilogue executed! In fact, the epilogue cannot be executed without **MAIN** explicitly dismissing the two packages from each other. This is because epilogues are *normally* executed for packages being dismissed that are accessed *by no one else*. Thus breaking up a package in this manner may actually require *changing all accessing packages*, an undesirable characteristic from a maintenance standpoint, particularly in large systems of packages.

To enable an epilogue to dismiss such local networks of sub-packages, you *can* execute epilogues in packages with active accesses by raising its *access count threshold* with the **DISMISS** *function* (no arguments). For example, setting this threshold to 5 (e.g., **DISMISS**=5) means that as soon as the package is **DISMISSed** and 5 or fewer accesses remain, its epilogue is executed. This threshold can only be set or examined within the current package and defaults to zero if not set. The way you would use this is to set **DISMISS** (in the *prologue)* equal to the number of **ACCESSes** that would normally be active when the package was *no longer in use* by any external packages. In our example above, you would set **DISMISS**=1 so that the epilogue of **PKG-B** would execute as soon as **PKG-C** remains its only accessor.

To assist package management, the **ACCESS** *function* (also no arguments) returns the number of packages that currently access the current package. One use for this is to set **DISMISS** = **ACCESS** in the *prologue* after all its sub-packages have been accessed and initialized. This would cause its cleanup epilogue to execute as soon as an *outer package* dismissed it, even though one or more of its sub-packages still accesses it at the time. For such a mechanism to operate effectively, the epilogue *must* dismiss all its sub-packages so that upon return from the epilogue, all accesses to the package are cleared and the system can then release the package from memory. Packages are *only* freed when *no other* packages are accessing it (i.e., when **ACCESS** = 0), no matter what you set **DISMISS** to.

Access-counts are incremented *on completion* of the package **ACCESS**, i.e., *after* the prologue executes. So calling the **ACCESS** function *in a prologue* returns an access count that *does not* include the **ACCESS** in progress and in fact will *always be zero* upon entry into a prologue. Conversely, access-counts are decremented *at the start* of the package **DISMISS**, i.e., *before* the epilogue executes.

A Model of a Multi-Package System

MegaBasic packages are extremely general and their facilities may be applied in an unlimited variety of configurations. Having so many different ways of solving the same problem can be initially confusing when learning how to use packages. It is therefore useful to have some simple model upon which to base your beginning approaches to system development. Once experience is gained in using packages with a simple model, the subtle nuances can be more fully explored and applied to your implementations.

A natural model of a system of packages is an extension of the subroutine concept. Suppose that your program appeared something like this:

```
ACCESS all immediately needed packages files
Perform all desired tasks
DISMISS all unneeded packages
GOTO beginning if more tasks to be done.
END
```

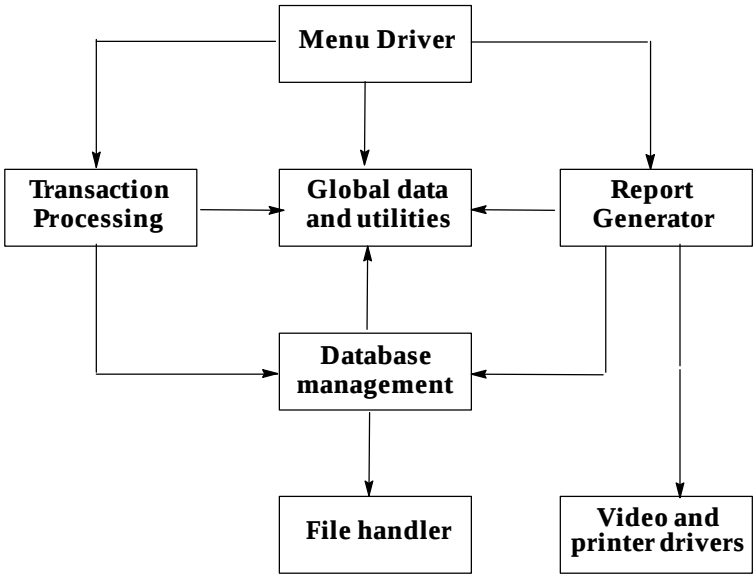
This model is conceptually very simple and yet provides a means to execute programs much larger than the available memory. However, let's further suppose memory was sufficient to contain your entire program, but you had so many packages loaded at the same time that the overall complexity became unmanageable. This important problem can be overcome if each of the packages **ACCESSed** above were to fit the following form:

- The *prologue* routine contains additional **ACCESS** statements to support itself by accessing its own packages.
- The *epilogue* **DISMISSes** all packages **INCLUDed** by the *prologue.*

This use of *prologue/epilogue* routines permits many packages to appear as one package, which has obvious simplifying implications. To do this effectively, each package should contain objects of roughly the same level of abstraction. The original main program would perform only the highest levels of processing, invoking primarily objects defined by other packages which in turn would be defined at lower levels in other packages until, at some point, all details were processed.

Multiple Package System Example

To further clarify how the package concept can be used to construct real software systems, an example transaction processing and reporting system is illustrated below. This is not a real program, but one possible model of a common system which requires 100–200k bytes of program code and 50–400k bytes of data space (variables). Each box shown represents one complete and separate package of related subroutines and data. Arrows have been drawn from each package to all other packages which it requires for its operation, in order to show the **ACCESS** relationships.



Menu Driver

This package is the highest level of the system, although it may be just another package in an even larger system (e.g., as a menu sub-system). It will have an initialization sequence implemented as a **PROLOGUE** which **ACCESSes** all the packages that it needs for its own processing. Since **MENU** provides a user interface to the transaction processing and reporting facilities of this system, its initialization *prologue will* include the statement:

```
ACCESS "TRANSACT", "REPORT", "GLOBAL".
```

Transaction Processing

TRANSACT is the package handling the various transactions that the user is allowed to make. It consists of a set of subroutines which are called from the ***MENU*** package and is shielded from having to *know* about higher-level details. When initialized, **TRANSACT** requires access to the global data area used throughout the system and to the data base manager. Its *prologue* therefore contains the statement:

```
ACCESS "GLOBAL", "DATABASE"
```

Report Generator

REPORT performs all of the display, listing and visual output of this system. Because of this it needs the services of the data base manager (**DATABASE**), as well as to the system global data package (**GLOBAL**) and a special graphics package designed to drive the various video and plotter hardware that may be present on the system. Its initialization *prologue will* therefore contain the statement:

```
ACCESS "DATABASE", "GLOBAL", "GRAPHICS".
```

Data Base Manager

DATABASE provides an interface into the data maintained and accessed from higher levels of the program. It consists of a collection of subroutines that perform searching, deleting, inserting, rearranging, renaming, editing and organizing of application data. **DATABASE** is independent of data representations and individual applications, so it accesses the global data area and to the system file handler. So when **DATABASE** is initialized, its *prologue will* access the required packages with the statement:

```
ACCESS "FILES", "GLOBAL"
```

This *prologue* might also initialize various arrays and other data structures so that it is immediately ready for service requests.

File Handler

This package provides the routines for converting high-level file requests from **DATABASE** into low-level system-specific file transfers. By changing this package, the entire system is able to move to another system with different file conventions (e.g., **CP/M** to **UNIX**). **FILES** does not require other packages for its operation, but its initialization *prologue* might open files and set up data structures so that subsequent service requests can be filled immediately.

Video and Plotter Graphics

GRAPHICS provides the **REPORT** generator with the routines necessary to interface with the display and hard-copy graphics hardware. It converts high level graphics commands (e.g., **DRAW**, **COLOR**, **SCALE**, **BORDER**, **FILL**, etc.) into the low-level hardware-specific commands to drive the graphics hardware. Like the file handler, this package does not require additional packages, but its initialization *prologue* would likely initialize the graphics hardware.

Global Data and Utility

GLOBAL contains all common variables, functions and procedures used by throughout the system. If needed, **GLOBAL** could contain an initialization *prologue* to dimension the **GLOBAL** arrays and strings and fill them with any desired initial information. Most of the packages in this imaginary multi-package system require access to **GLOBAL** and access it in their *prologue* routines.

Additional Comments on this Example

We have assumed this system resides in memory, but if **MENU** is just another package of a larger system, then additional operations are needed to remove it and its sub-packages from memory when no longer needed. Because of its complex data and file structures, various clean-up tasks may be required to maintain the integrity of the data base involved. Therefore **EPILOGUES** should be defined within each package to:

- Clean-up all unfinished business, such as posting recent changes to files, closing various open files, setting global status flags, clearing buffers, etc.
- Release all subordinate packages. This is done by merely listing the same package names that appeared in the **ACCESS** statement (executed by the *prologue)* in a **DISMISS** statement.

An **EPILOGUE** is the perfect place this because it is invoked automatically when a package is **DISMISSed** (or the entire system **ENDS**) and the details of its clean-up operation (or even its existence) need not be known by the package doing the **DISMISSing**. Be sure to order the package names in the **DISMISS** statement if **EPILOGUE** *ordering is* important.

Converting LINKed Systems into Package Systems

The package concept is extremely powerful and general, but large programs written in **BASIC** usually rely on **CHAIN** statements (like the MegaBasic **LINK** statement) to sequence from one program module to the next. Although this permits machines with limited memory to execute large programs, it does nothing to improve the flexibility of the system and usually results in deterioration of program structure. When such program systems are transported to the MegaBasic environment, it is desirable to replace any reliance on **CHAINing** or **LINKing** with an equivalent mechanism based on *packages. To* see how to do this, let's examine the essential properties of **LINKed** systems:

- Each program module in a **LINKed** system runs as a stand-alone program which, on completion, causes another program module to be loaded from a disk file into memory and then started.
- Each program module must decide for itself the next program module to be run, so that each module can sequence to one of any number of other modules depending on the prevailing conditions.

- Some subset of program variables may be communicated from one **LINKed** module to the next. Not all **BASICS** support this capability but programs written to use it (such as earlier MegaBasic programs) must be permitted to continue doing so.

To provide these capabilities using a purely package concept, each of the **LINKed** modules can be a separate package and a very small main program will remain in memory throughout the life of the system to manage the process bringing program modules into memory for execution and removing them when finished. This main program will look something like this:

```
10 Rem 1 *** Main program to control LlNKed system
20 Def shared VBL1, VBL2, ARRAY( ), NEXT_PGM$
30 NEXT_PGM$= module1; Rem--Assign the first module name
40 LAST_PGM$ = NEXT_PGM$; Rem -- Save the module name
50 Include NEXT_PGM$; Rem -- Execute the next module
60 Dismiss LAST_PGM$e; Rem -- Remove it on completion
70 If NEXT_PGM> then 40; Rem -- Sequence to the next module
80 End; Rem -- Done when no more
```

In line 20 we define all the variables which are to be common to all **LINKed** modules. This list can, of course, be extended to define as many variables as required. One such variable, **NEXT_PGM$**, iS a common string variable which will always contain the name of the next package module to bring into memory. Instead of directly **LINKing** to the next module, each package merely sets **NEXT_PGM$** to the name of the module to be run next and then returns so that main program can bring this about.

In line 30 we set **NEXT_PGM$** to the name of the first module to be executed. Line 40 is the beginning of the main execution loop and module name is saved in a private string variable for later reference. Line 50 brings the next module into memory with an **INCLUDE** statement, which causes the **PROLOGUE** of package loaded to be executed. The **PROLOGUE** of each module is responsible for performing all tasks to be done by the module, i.e., it is really the main controlling program within each module. Before each **PROLOGUE** returns, it must set **NEXT_PGM$** to the name of the next module to be run.

When line 60 is reached, the **PROLOGUE** of the **INCLUDEd** package has completed and **NEXT_PGM$** should now contain the name of the next module to be run. But first we remove the module just completed using a **DISMISS** statement. Line 70 tests **NEXT_PGM$** and loops back to repeat the whole process if it contains a module name, or terminates execution (by falling into line 80) if **NEXT_PGM$** contains no module name.

System termination can be done by executing an **END** statement within any of the component module packages, but the method described here centralizes this action and provides more control. For example you could branch to a pre-determined module (e.g., a main menu) whenever **NEXT_PGM$** is returned empty, instead of terminating the system. It should be clear at this point that there is a *structure* to each of the **LINK** module packages. A summary of this structure now follows:

- Each package must contain a **PROLOGUE** which in turn acts as the main program of the module, controlling the sequence of events within that package.
- The **PROLOGUE** must gain access to the set of common variables defined by the managing main program of the system. Its first statement should therefore be: **ACCESS** "**MAIN**", where "**MAIN**" is the name of the main program.
- Any time prior to **RETURNing**, each **PROLOGUE** must set **NEXT_PGM$** to the name of the module to be subsequently run. Instead of **LINKing** to the next module, the **PROLOGUE** merely **RETURNS** with this information and the main program takes over.

The biggest problem you are likely to encounter in converting a **LINKed** system into a package system, is that you must always **RETURN** from the **PROLOGUE** level in order to cause the next module to be executed. **LINK** statements, however, can be executed at any subroutine level to bring in the next module. Low level **LINK** statements will therefore have to be re-implemented in such a way as to meet the **RETURN** level criterion. By restructuring your program module so that all **LINKS** are performed at the top level of the module (i.e., no **LINKS** within procedures, functions and **GOSUBs**), the job of converting to packages will be trivial. Such a change in structure will have the additional side-benefit of forcing the intermodule interface to be centralized at the top of the program, improving the maintainability of the program as a result.

Section 4: The Multi-Package Development Environment

With more than one program source in memory at one time, the workspace environment becomes a collection of workspaces, one for each package. Within each workspace, you may alter and debug the source code it contains without altering the contents of any other workspace. It is only after you switch to another workspace that you may apply program development facilities to another program source.

There are two kinds of workspaces: temporary and permanent. Workspaces created by **LINK**, **ACCESS** and **INCLUDE** statements are *temporary*, because the **RUN** command eliminates them just prior to beginning program execution. The **LOAD** command may be used to create and fill *permanent workspaces*, i.e., those which survive a **RUN**. Permanent workspaces persist until they explicitly eliminated using the **CLEAR** command. If you switch to a temporary workspace and modify its contents, it becomes a *permanent* workspace. When saved to a file, such a workspace reverts back to *temporary* status. This scheme let MegaBasic protect and maintain the source files you are actually working on while automatically eliminating unnecessary source files from your memory space.

When automatically eliminated, workspace contents is merely marked free. If the memory space occupied by freed programs is later needed by other activities, MegaBasic *physically releases* them to obtain the required memory. Such programs will reside in memory indefinitely if their occupied memory is never needed. If such programs are again **ACCESSed** before released, no disk transfers are required because the program is already memory resident. Therefore, by careful design, you can implement multi-package systems which automatically adapt to the amount of memory available in the host computer. Given enough memory, all packages will stay resident; with somewhat less memory, occasional disk activity will occur to bring back a temporarily **DISMISSed** package.

Each workspace carries the name of the package it contains. You can see a summary of all workspaces by issuing a **SHOW** command (Chapter 2, Section 5). Each workspace name is displayed along with the package size and the amount of data currently owned by the package. The currently selected package is always displayed at the top of the list. To see the **ACCESS** relationships involving the current package, use the **SHOW ACCESS** command (Chapter 2, Section 5).

Several operational parameters are locally maintained for each workspace to facilitate their independence. The auto-**SAVE** file name is unique to each workspace since each package comes from a different file. The Ctrl-C state, disabled and enabled by **PARAM(1)**, is also separately defined for each package. Likewise, the **TRACE** mode is set up on an individual basis so that all debugged packages can **TRACE** as if they are built into the language (by not tracing). To more fully clarify the package environment, a complete list of facilities common to all packages is contrasted below with those facilities which are maintained independently:

Global Facilities and Attributes

I/O devices and OPEN files	**Value of INDEX**
File end mark and no mark state	**Line editing buffer**
Scratchpad area	**Initial string code**
File transfer floating point size	**8087/809x87 State**

Local Facilities and Attributes

DATA-READ pointer	**Program source**
Ctrl-C *disable/enable*state	**Package name**
Execution TRACE mode	**Current line ranges**
Conditional TRACE IF expression	**Auto-SAVE file name**
TRACE: execution line	**Default PRINT format**
Structured variable defaults	**Program variables**

Switching from Workspace to Workspace

INCLUDE and **ACCESS** statements load packages into new workspaces, but they do not change your current workspace. With the **USE** command (Chapter 5, Section 5) you can select any of the packages currently available in memory for subsequent operations. Typing **USE** followed by a carriage return lets you to step from workspace to workspace until you reach your desired package.

The **USE** command may also be typed with a *package name* in order to directly select your desired workspace without having to sequence through all the names available. If the supplied package name is found among those present in memory then it will be immediately selected. If it is *not found*, a new empty workspace can be created under the name given.

To minimize the number of unnecessary workspaces in memory, MegaBasic automatically deletes workspaces that contain no program lines. This action is taken only when you *leave* the empty workspace, by entering a different workspace (with **USE**). Hence it is not possible to create several empty workspaces and then go back to fill them in: they must be used immediately.

When a program stops for any reason (i.e., **END**, **STOP**, errors), the currently selected workspace is set to that package which contains the code in which the stop took place. This is most convenient for debugging purposes and eliminates the need for explicitly selecting (**USE**) packages in many instances. MegaBasic displays the current package name whenever it automatically changes the workspace. Whenever you **LOAD** a package into a workspace, it becomes the currently selected workspace.

Workspace Implications to Direct Statements

Any valid program line typed without a line number is always executed immediately by MegaBasic. The meaning of any identifiers in direct statements is always taken from the *current workspace*, which consists of all the names defined by the current package plus all **SHARED** names **ACCESSible** from the current package. Therefore the same direct statement typed under different workspaces may easily produce different results. New variables created by direct statements, by default or explicitly, will *belong* to the package contained by the current workspace. **ACCESS** statements typed directly will affect the access rights of the packages specified. **INCLUDE** statements typed directly will load additional packages into memory. All modifications to the program state (i.e., variable contents, **ACCESSibility**, **INCLUDES**, etc.) will carry over and affect execution accordingly when the program **CONTinues**.

Source Modification Effect on Accessibility

When you interrupt execution with Ctrl-C, it can be **CONTinued**, and MegaBasic permits minor source code alteration without upsetting **CONTinuability**. Revision of certain lines can curtail **CONTinuability** (Chapter 2, Section 4). **ACCESSibility** can also be affected by modification and **CONTinued** execution might not possess the same **ACCESS** configuration, as summarized below:

- Renaming **SHARED** objects usually leads to trouble if subsequent **CONTinuation** is intended.
- Line editing, insertion and deletion does not affect the ability to access **SHARED** names in other packages.
- **SHARED** variables that disappear from the program due to editing out all references to them will continue to exist and will remain accessible from other packages.
- External references to **SHARED** functions and procedures will become undefined if they are *edited out* of the source.
- Additional **SHARED** objects *edited into* a package will be **ACCESSible** from other packages when execution **CONTinues**.

Executing a Multi-Package Program

When more than one program is in memory and **RUN** is invoked, whichever workspace you are in becomes the main program. Prior to beginning program execution, **RUN** performs the following sequence of operations:

- The program residing in the currently selected workspace becomes the main program. All data currently defined by the main program is erased and released to free space. If the current workspace is empty, a *No Program Error* results when **RUN** is attempted.
- All temporary workspaces are marked free, and data they own is released to free additional memory. This consists of all unaltered packages brought into memory with **INCLUDE** or **ACCESS** statements.
- **LOADed** packages are retained in memory, but all **ACCESSibility** to and from them is severed. Data defined by such packages is retained also. Via this mechanism, special purpose packages (e.g., debugging routines or completely independent programs) may remain available indefinitely.
- The main program is set to *permanent status* (regardless of its prior status). The **DEF** statements throughout the program are initialized and program execution begins.

If a program has been interrupted with Ctrl-C or encountered a **STOP** statement during execution, the **CONT** command will continue execution. Regardless of what package workspace you are in when you type **CONT**, MegaBasic always switches to the workspace in which the **STOP** took place, prior to resuming execution.

Unfinished Epilogues

When a multi-package MegaBasic program terminates prematurely, due to a *Ctrl-C* or untrapped error interruption, you can always enter a direct **END** statement from the keyboard to execute all the remaining epilogues. However this step is easy to forget and, under some circumstances, execution of the package epilogues may be vital to subsequent program operation (e.g., epilogues may release important system resources). Therefore, MegaBasic detects situations where *epilogue closure* may be disrupted and executes an implicit **END** statement before continuing on to perform certain commands. This occurs when there is at least one remaining epilogue and you type a **LOAD** (into an existing active workspace), **RUN**, **CLEAR** or **BYE** command.

Loading Programs into Multiple Workspaces

The **LOAD** command is used to load a program file into a workspace. If the filename specified happens to be in memory already, then that workspace becomes overwritten with the loaded program. If it is not present in memory, you can **LOAD** the file into either the current workspace or a new workspace, depending on your response when asked: *Into a new WorkSpace?* An error results if the filename is found neither in memory nor in the file directory. After any successful load operation, the workspace that contains the **LOADed** file is selected for subsequent operations.

Saving Programs and Eliminating WorkSpaces

The **SAVE** command writes the program contained within a workspace onto a file. If the filename specified happens to be the name of a current workspace, that workspace is written onto the file of the same name. Otherwise, the program in the current workspace is written to the specified file, and the workspace is renamed to match the filename given in the **SAVE** command. In any case, after a **SAVE** you will be in the same workspace as before the **SAVE**.

The **CLEAR** command (Chapter 2, Section 5) deletes the program in the current workspace and then eliminates the workspace altogether (unless it is the sole workspace). Afterward, MegaBasic switches to the next workspace in the **LOAD** sequence. All workspaces may be **CLEARed** using this command by confirming an explicit request from MegaBasic when you invoke this command.

Tips on Package Development

It is a good idea to completely debug a package prior to bringing it into a system of other packages. Build a test routine into the package to simulate a main program, then run the package by itself. Such test routines should be left in the packages (where feasible) after being debugged for later debugging and for documentary value.

Avoid declaring anything as **SHARED** unless absolutely necessary. This policy promotes independence among packages and minimizes the possibility for clashes between identical local and global identifiers. Another good practice is to use longer, more descriptive names for **SHARED** objects than you would normally for **unSHARED** objects to make the references to these global objects *stand out* in the program source.

There is about 500 bytes overhead in each executing package, so it generally take less memory to have fewer larger packages than numerous small packages. Executing your application under the RUN version of MegaBasic reduces this overhead further.

Section 5: Assembler Packages

This section describes the structures and protocols required by MegaBasic packages developed in assembler. It assumes that the reader already understands both the concepts involved in using MegaBasic packages and how to write programs in 80x86 assembler. Use of MegaBasic assembler packages is virtually identical to using MegaBasic interpreter packages. Their highlights and differences are summarized below:

- Assembler packages are written in machine code using an assembler or possibly a high-level compiler language, such as C. Because of this, the operations such a package supplies are extremely fast.
- Assembler packages contain procedures and functions invoked from your MegaBasic applications which support a flexible argument structure for communicating with the calling program. **SHARED** variables are not supported by assembler packages.
- Assembler subroutines are invoked with *names* and *argument lists* instead of **CALLS**, **PEEKS** and **POKES**, making them as easy to use as the built-in functions and statements of MegaBasic.
- Assembler packages are **ACCESSed**, **INCLUDEd** and **DISMISSed** from MegaBasic programs just like interpreter packages. They can **also** be included in binary load images created by the **PGMLINK** facility that comes with the MegaBasic **RUN** version.
- Unlike interpreter packages, an assembler package cannot **ACCESS** any other MegaBasic package. It should be regarded as a completely self-contained library of machine-coded subroutines.
- The **SHOW** command will display the names and sizes of all assembler packages (along with all other packages) currently loaded, and they are shown with the type *Binary*. Obviously, such packages cannot be listed or modified and therefore you are prevented from *getting into* these workspaces with the **USE** command.
- Assembler packages have to be written very carefully and all the rules for constructing them must be followed exactly. Programming errors in these packages will, more often than not, crash the system. Furthermore the debugging environment MegaBasic provides to interpreter programs does not apply to assembler packages. In short, you are on your own.
- Additional support for assembler packages includes assembler source code for an example package and several utilities to assist your efforts in creating, checking and listing assembler packages.

Assembler packages are accessed from MegaBasic programs using the **ACCESS** statement or the automatic package mechanism (see the **CONFIG** utility). **ACCESS** performs roughly the same sequence of operations to link both package types to a MegaBasic program. This process can be summarized as follows:

- Look up the specified package name in the set of packages already loaded. If not in memory then look up the specified package name in the file directory. If it is found then load the package into memory, otherwise report a *File not found* error.

- Determine whether the package is a interpreter or assembler package, by examining the identifying characteristics in the package header. If it is an assembler package then continue on, otherwise **ACCESS** the interpreter package. If the package does not contain the proper assembler package *signature* (a 16-bit constant at a pre-defined displacement) then report an *Illegal package operation* error.
- If the package has already been initialized then skip this step. Otherwise validate the structure of each defined procedure and report inconsistencies as a *Package Definition Error*. This is the onetime initialization phase. MegaBasic then searches for a **PROLOGUE** and executes it if found. A corresponding **EPILOGUE** is executed when the package is **DISMISSed** and no longer **ACCESSed** by any other MegaBasic packages. **PROLOGUES** and **EPILOGUES** are optional.
- Link each name defined in the assembler package to references in the **ACCESSing** package. If any of these names are already defined then report a *Package Name Conflict Error*.
- The linkage process is complete. The MegaBasic program can now invoke any of the assembler subroutines as if they were built-in library routines. Once brought into memory, assembler packages will show up in the listing generated by the **SHOW** command as *Binary* packages.

Once **ACCESSed**, all the procedures and functions that have been implemented by the package are immediately available, and can be invoked by name from programs or from direct-statements typed at the keyboard. The rules for invoking procedures and functions are simple and direct:

- Procedures begin with a name, followed by zero or more arguments. The arguments are not surrounded by parentheses and their type must match their type defined in the assembler package.
- Functions must appear in an expression context, and consist of a function name and an optional argument list surrounded by parentheses. Arguments must match their type as defined in the package.

Arguments can be values (i.e., expressions) or variables (i.e., passed by address). Any argument can be defined as optional, so that the calling reference can leave off some or all arguments depending on the application. Optional arguments can be omitted anywhere in the argument list, not just from right-to-left as in interpreter **PROCS** and **FUNCS**. For example the function call ***TEST(FIRST,,THIRD,,FIFTH)*** omits the second and fourth arguments. The extra commas are only required when a specified argument follows an omitted argument (e.g., the **TEST()** function above might have more than five arguments defined).

Defining Assembler Packages

In order for MegaBasic to recognize and correctly process an assembler package, the package must conform to a specific structure or layout. This layout is designed for efficient MegaBasic use and for simplicity from the assembler implementor's point of view. The overall structure is quite simple, as described below:

- At the beginning of the file is a package header, usually 64 bytes. This header identifies the package to MegaBasic and defines certain necessary information for loading and executing it.

- After the package header, are one or more **FAR** subroutine blocks that implement each of the **PUBLIC** entry points provided to the calling MegaBasic program. Each block consists of a subroutine header and the assembler code that implements the subroutine.
- The entire package file must be less than 64k bytes in length. Although this size may seem limited, a tremendous amount of machine code can fit in such a space. MegaBasic itself is not much larger than this.

The package header identifies the package as a MegaBasic assembler file and includes a minimal amount of other global information. The header is defined in the **INCLUDE** file supplied with the MegaBasic developers tool kit **(ASMDEFS)**, which currently defines the following fields:

Field Offset	Package Header Field Contents
0000	Two bytes containing the hex values 04h, 00h to help MegaBasic discriminate between the various types of package files that can be loaded (i.e., interpreter files, ASCII files, assembler packages, etc.).
0002	WORD containing the number of bytes in the header.
0004	WORD containing a *signature* that identifies the file as a valid assembler package suitable for MegaBasic. The value defined by the **INCLUDE** file is **ADEFh** and is generally not of any concern to the programmer
0006	WORD defining the number of memory bytes to provide in the code segment beyond the size of the package file. This allows a small assembler package in a small file to be placed in a code segment of arbitrary size. This value must be specified as the label **ADDMEM EQUated** to the byte count desired and placed just above the **INCLUDE ASMDEFS.asm** statement. If this value implies a total segment size larger than 65536 bytes, the size is set to 65536. If **ADDMEM** is not defined in your assembly, this **WORD** will default to zero.
0008	WORD that specifies the offset of the first subroutine header in the package memory image. Normally, this is the offset that immediately follows the package header (i.e., 0040h).
0010	6 bytes reserved for future use (must be set to zero).
0016	16 byte package name buffer filled in by MegaBasic.
0032	BYTE containing the floating point precision of the running copy of MegaBasic. **BCD** to 18, and **IEEE** real format is approximately 16 digits precisions span from 8 (double precision Intel format for the 80x87). This byte is filled in by MegaBasic when the package is first read into memory.
0033	BYTE indicating the format used by the running copy of MegaBasic to represent floating point numbers. Two types are available: 0 indicates **BCD** format, and 1 indicates **IEEE** double-precision binary format. This byte is filled in by MegaBasic when the package is first read into memory.
0034	DWORD containing ES:BX from an invoking *logicalinkrrupt* call.
0038	BYTE with special control flags. Currently, only bit7 is defined and indicates package usability in protected mode under Extended MegaBasic. All other bits must be zero.
0039	25 bytes reserved for internal use and future expansion and must be set to zero.

The **FAR** procedure blocks that immediately follow the header each consist of a procedure header, the procedure name string, and an assembly code procedure that implements the desired operation. Each of these blocks should begin on an even offset for performance reasons, but this is not mandatory. The procedure header defines the name of the procedure, its type (i.e., procedure or function), and its argument structure. The procedure header specification is defined as follows:

Field Offset	Procedure Header Field Contents
0000	WORD containing the offset of the next procedure block header. Zero indicates that no more procedure blocks follow. Non-zero values are relative to the beginning of the package header.
0002	WORD containing the offset of the procedure code entry point. This is where MegaBasic enters the assembler procedure.
0004	WORD contains the offset of the procedure name. Names are defined later on.
0006	WORD defining the procedure type: procedure or functions. Functions may return integers, reals and strings. The legal values for this field are: **PROCED**, **IFUNC**, **RFUNC**, **SFUNC**, **PROLOGUE** and **EPILOGUE**. These are defined in a special **INCLUDE** file **(ASMDEFS.asm)** provided with the MegaBasic developers tool kit. A **PROLOGUE** is just a procedure that is automatically executed when the package is initially loaded. An **EPILOGUE** is a procedure that is automatically invoked when the package is removed (using the MegaBasic **DISMISS** statement). You should not declare more than one subroutine as a **PROLOGUE** or an **EPILOGUE**.
0008	WORD defining the maximum length of the result returned from a procedure of type **SFUNC**. It should be sel to zero for all other procedure types. This value, a byte count, causes this much memory to be reserved on the logical control stack for the returned string function result.
0010	3 **WORDS** reserved for possible future use.
0016	WORD specifying the maximum number of arguments defined for the procedure. Zero may be specified tc indicate no arguments.
0018	Zero or more **WORDS** defining the argument types from left to right. The number or words defined here must be the same as the count defined in the preceding field. The permissible values for this field are defined in an **INCLUDE** file provided with the MegaBasic developers tool kit. The labels it defines are described below.

The procedure name, referenced by the procedure header, consists of a *length byte* followed by the *name characters* (in upper case) and terminated by a *carriage return* (ODh). The length byte counts *itself,* the name *length* and the carriage return *terminator.* The name portion must be a valid MegaBasic identifier. **PROLOGUES** and **EPILOGUES** do not need names, but you should define a *null* name for them, consisting of the two bytes: 2,0Dh. Avoid names matching MegaBasic reserved words, because they would never be accessible from your MegaBasic program, and no diagnostic is provided.

Defining and Accessing Arguments

Argument types include integer, real and string, which can be specified in two modes: by value, by address. Arrays of any type (i.e., integer, real and string) can be passed by address. Arguments passed by value are read-only values, i.e., the procedures can access and use the values passed, but they should not attempt the change the value. Such attempts would not result in any data being passed back to the calling MegaBasic program. On the other hand, arguments passed by address are designed for access and alteration. Value arguments can be specified in the calling MegaBasic program as general expressions, while address arguments must be simple variables, indexed and non-indexed string variables and array elements only. MegaBasic will report an error if these restrictions are violated or if the number of arguments specified does not match the number defined (allowing for optional arguments, of course).

Arguments are defined in a list at offset 0010h in the subroutine header **(PRLIST)** consisting of the defined argument count followed by a series of 16-bit constants indicating the argument type. These constants are defined in the **ASMDEFS.asm INCLUDE** file. By default, an argument type constant defines mandatory arguments (i.e., non-optional). To define an optional argument constant, you must follow the constant label with **AND OPTIONAL**. For example, the following DWs define a non- optional integer value argument and an optional integer value argument:

```
DW INTVAL ;Mandatory integer-value argument
DW INTVAL AND OPTIONAL ;Optional integer-value argument
```

When MegaBasic evaluates arguments, the actual argument type specified is verified against its defined type and if they do not match, an error is reported. Otherwise, MegaBasic builds a data structure that describes the argument specified and passes a pointer to this structure on the **CPU** hardware stack.

Arguments are passed to assembler package subroutines on the **CPU** stack in the following way. Upon entry to the subroutine, SS:BP will point to a series of 16-bit words that each represent one argument. The words themselves each contain either a zero, indicating an omitted argument, or the stack segment offset of a description of the argument passed. This may be more easily understood from the 4-argument example below:

Addressing	Word Accessed	Argument Description Addresses
SS:BP+6	WORD3	Fourth argument
SS:BP+4	WORD2	Third argument
SS:BP+2	WORD1	Second argument
SS:BP+0	WORD0	First argument

This method is designed for fast indexed access to any argument, for supporting omitted arguments, and to accommodate new argument types. Each argument description depends on the argument type, i.e., an integer value has one description, a string variable has another. The table below summarizes the layouts of each argument description:

	Word0	Word1	Word2	Word3	Word4	Word5
Integer Value	INTVAL	Offset	Segment			
Integer Variable	INTVBL	Offset	Segment			
Real Value	REALVAL	Offset	Segment			
Real Variable	REALVBL	Offset	Segment			
String Value	STRVAL	Offset	Segment	Length		
String Variable	STRVBL	Offset	Segment	Length	Max. length	Len Offset
Any Value	ANYVAL	Offset	Segment	Length		
Any Variable	ANYVBL	Offset	Segment	Length	Max length	Len Offset
Array Variable	ARRVBL	Offset	Segment	Elem. Count	Elem. Length	Dim Offset

In the preceding table, the words **ANYVAL** and **ANYVBL** are replaced by the code for the actual data passed through that argument (i.e., **ANYVAL** is replaced by **INTVAL**, **REALVAL** or **STRVAL**, and **ANYVBL** is replaced by **INTVBL**, **REALVBL**, **STRVBL** or **ARRVBL**). In the following pages, we will discuss each argument type in detail, including both the argument description and how it is specified.

Integer Values

An integer value argument is defined in the subroutine header with the label **INTVAL**. The calling program must supply a numeric expression that evaluates to a number that is an integer or one that can be converted to an integer. This conversion is done automatically by MegaBasic as needed. The internal description of an integer value argument consists of three 16-bit words:

WORD0	Constant **INTVAL** indicating an *integer value* argument.
WORD1	Offset of the integer value.
WORD2	Segment of the integer value.

The Segment / Offset pair is a double-word pointer to the actual 32-bit integer value. Arguments of this type can be accessed by the subroutine but changes to this value are not passed back to the MegaBasic program through this argument. This is generally the fastest method to pass a numeric value to an assembler procedure.

Integer Variables by Address

An integer variable argument is defined in the subroutine header with the label INTVBL. The calling program must specify this argument type with a reference to a scalar integer variable or an integer array element. The internal description of an integer variable argument consists of three 16-bit words:

WORD0	Constant **INTVBL** indicating an *integer variable* argument.
WORD1	Offset of the integer variable.
WORD2	Segment of the integer variable.

The Segment / Offset pair is a double word pointer to a MegaBasic integer variable or array element. The pointer can be used to both access and modify the 32-bit integer, and points to the lowest order byte of the integer variable (at the lowest offset). Changing the memory contents at this location will affect the contents of the MegaBasic integer variable in the calling program.

MegaBasic will report an error if the actual argument passed to the procedure was not an integer variable, e.,g. an expression of any kind, a real variable or a string variable.

Real Values

A real value argument is defined in the subroutine header with the label **REALVAL**. The calling program must supply a numeric expression that evaluates to any number (real or integer). MegaBasic automatically converts integers to real whenever they are specified for this argument type. The internal description of an real value argument consists of three 16-bit words:

WORD0	Constant **REALVAL,** identifying a *real value* argument.
WORD1	Offset of the real value.
WORD2	Segment of the real value.

The Segment / Offset pair is a double-word pointer to the leading byte (i.e., the lowest memory offset) of the actual real value. Arguments of this type can be accessed by the subroutine but changes to this value are not passed back to the MegaBasic program through this argument.

Subroutines that access real variables are expected to know the format of real numbers. MegaBasic supports BCD real formats in 8,10,12,14,16, and 18 digit precisions, and binary real numbers in Intel **IEEE** double-precision format (8 bytes long), suitable for 8087 or 80287 operation. Any individual copy of MegaBasic supports just one of these formats.

Real Variables by Address

A real variable argument is defined in the subroutine header with the label **REALVBL**. The calling program must specify this argument type with a reference to a scalar real variable or a real array element. MegaBasic reports an error if you specify any type of expression or string for this argument type. The internal description of an real variable argument consists of three 16-bit words:

WORD0	Constant **REALVBL**, identifying a *real variable* argument.
WORD1	Offset of the real variable.
WORD2	Segment of the real variable.

The Segment / Offset pair is a double-word pointer to the leading byte (i.e., the lowest memory offset) of the actual real variable passed. Changing the memory contents at this location will affect the contents of the MegaBasic real variable in the calling program. Subroutines that access real variables are expected to know the format of real numbers.

String Values

A string value argument is defined in the subroutine header with the label **STRVAL**. The calling program must supply a string expression for this argument type. The internal description of a string value argument consists of four 16-bit words:

WORD0	Constant **STRVAL** identifying a *string value* argument.
WORD1	Offset of the string value.
WORD2	Segment of the string value.
WORD3	Number of bytes in the string value.

The *Segment/Offset is* a double-word pointer to leading byte of the string value. Arguments of this type can be accessed by the subroutine but changes to this value are not passed back this argument. If you modify this string for some reason, do not alter any bytes passed the end of the string.

String Variables by Address

A string variable argument is defined in the subroutine header with the label **STRVBL**. The calling program specifies this argument type with a reference to a scalar string variable or a string array element, and indexing expressions *are allowed* on these string variable references. The description of a string variable argument consists of six 16-bit words:

WORD0	Constant **STRVBL** identifying a *string variable* argument.
WORD1	Offset of the string variable.
WORD2	Segment of the string variable.
WORD3	Number of bytes in the string variable.
WORD4	Maximum number of bytes the variable can hold.
WORD5	Offset of the length of the string in the variable. The segment is the same as specified by WORD 2. Zero indicates that the length *cannot be altered*.

The Segment / Offset pair is a double word pointer to a MegaBasic string variable or string array element. The pointer can be used to both access and modify the string contained in the variable, and points to the first byte of the string. Changing the memory contents at this location will affect the contents of the MegaBasic string variable in the calling program. Under no circumstances should any bytes beyond the maximum length of the variable be altered.

The length of string variables that are not indexed can be modified to any length from 0 to the maximum permitted length given by WORD 4. Indexed string variables are fixed-length and this is indicated by a zero offset in the length word in WORD 5. The current length (in WORD 3) and the maximum length (in WORD 4) will always be the same for indexed string variables.

MegaBasic will report an error if the actual argument passed to the procedure was not a string variable or string array element, e.g., an expression of any kind, a real variable or an integer variable.

Arrays by Address *(Not currently supported)*

A array variable argument is defined in the subroutine header with the label **ARRAYI** for integer arrays, **ARRAYR** for real arrays or **ARRAYS** for string arrays. The calling program must specify this argument type with a reference to a simple array name, without any subscript or parentheses. MegaBasic reports an error if any expression or non-array variable is specified. The internal description of a string variable argument consists of six 16-bit words:

WORD0	Constant **ARRAYI** for *integer arrays*, **ARRAYR** for *real arrays* or **ARRAYS** for *string arrays*.
WORD1	Offset of the first byte of the first array element.
WORD2	Segment of the array variable.
WORD3	Number of elements in the array.
WORD4	Number of bytes per element.
WORD5	Offset of array dimension list.

The array element size depends on the array type: integers and reals are identical to scalar values, and string elements have a length defined in the array definition (a topic covered below). Large MegaBasic arrays (i.e., those that exceed 64k bytes) cannot be accessed by assembler procedures because of the complex segmented data structures involved. The layout of MegaBasic arrays is described below:

Displacement	Numeric Arrays	String Arrays
0000	1st dimension size	Maximum length
0002	2nd dimension size	1st dimension size
0004	3rd dimension size	2nd dimension size
	and so on...	and so on...

The dimension count is the number of elements in that dimension. For example the array X(99,40) has 100 elements in the first dimension and 41 elements in the second dimension. The list of dimensions is followed by a word of zeros (16-bit) as a terminator. Immediately after the zero terminator follows the sequence of array elements. The elements are ordered such that as you advance sequentially through memory, the right-most subscript varies the most rapidly and the left-most subscript varies the least rapidly.

The elements of a string array contain both the string and its length in the following data structure. The first two bytes of each element form a 16-bit word count of the number of characters in the string. This count is immediately followed by the string (i.e., the number of characters indicated). The maximum length that the string array element can hold is the number of bytes per element minus 2. All elements are the same size. You can modify both the string contents and the length word, but be sure that you do not modify any bytes past its maximum capacity, nor set the length to any value larger than the element size minus 2.

Values of Any Type

By setting the argument type in the subroutine header to the label **ANYVAL**, the argument expression may evaluate to a string, integer or real result. After evaluating the argument, MegaBasic leaves the result in one of the value argument description formats described earlier: **INTVAL**, **REALVAL** or **STRVAL**. The argument type code that appears

in the argument description is set to reflect the actual argument type evaluated (i.e., it *is not* marked **ANYVAL**). For example, if the **ANYVAL** argument expression turned out to be an integer, then an integer value description would be provided and its leading **WORD** would contain the label **INTVAL**.

To use such generic arguments, your assembler code must check the first **WORD** of the argument description to find out which argument type was passed. Generic arguments are useful in subroutines that have to operate on any kind of data or in subroutines that determine omitted arguments by data type context. More steps are required to use such arguments, so you should really need them before you decide to use them in your subroutines.

Scalar Variables of Any Type by Address

By setting the argument type in the subroutine header to the label **ANYVBL**, the argument expression may evaluate to a string, integer or real *variable.* After evaluating it, MegaBasic creates a *variable argument* description using the appropriate format: **INTVBL**, **REALVBL**, or **STRVBL**. The argument type code that appears in the argument description is set to reflect the actual argument type evaluated (i.e., it is not marked **ANYVBL**). For example, if the **ANYVBL** argument was a *string variable,* then a **STRVBL** description would be provided. See the discussion on **ANYVAL** arguments above for further information.

Array Variables of Any Type by Address *(Not currently supported)*

By setting the argument type in the subroutine header to the label **ANYARR**, the argument expression may evaluate to a string, integer or real array. After evaluating such an argument, MegaBasic leaves the result in one of the array argument description formats described earlier: integer array **(ARRAYI)**, real array **(ARRAYR)**, or string array **(ARRAYS)**. As with the other generic value arguments (**ANYVAL** and **ANYVBL**), the argument type code that appears in the argument description is modified to reflect the actual argument type evaluated (i.e., it is not marked **ANYARR**). For example, if the **ANYARR** argument turned out to be an real array, then an real array description would be provided and its leading **WORD** would contain the label **ARRAYR**. See the discussion on **ANYVAL** arguments above and **ARRAYS BY ADDRESS** earlier for further information.

Subroutine Code

The subroutine code accesses the arguments, if any, passed by the calling program, and executes its designated operation in assembler. MegaBasic always enters the subroutine at its defined entry point with the following registers setup:

- DS, ES and CS all point to code segment of the assembler package.
- SS:SP points to the FAR return address on the machine stack.
- CX contains the count of how many arguments were actually specified in the calling MegaBasic program reference. This count includes omitted arguments only if they were preceded or followed by a comma (i.e., CX contains the number of comma-separators plus one). CX is zero if no arguments were specified.
- SS:BP points to the WORD containing the offset of the first (or left-most) argument. The next WORD up (i.e., at SS:BP+2) points to the WORD with the offset of the second argument, and so on. An omitted argument is indicated by a WORD that contains zero.

- SS:BX points to the base location on the MegaBasic logical control stack at which function results are placed. This pointer is meaningless for procedures.

The subroutine is expected to know how to access each of the arguments passed to it from MegaBasic. This is reasonable since the subroutine definition has defined the argument types, SS:BP points to CX words that point to the argument data structures (also in the stack segment), and each data structure begins with the argument type constant for programming convenience.

It is important that the subroutine code does not rely on any fixed code segment addresses. MegaBasic loads the assembler package into memory wherever it is possible, which may not be in the same place at different times. Also, the memory segment it resides in may move from time to time to make room for new data structures required or created by the MegaBasic user program. Hence the code must remain segment relocatable at all times. This also means that your assembler subroutines cannot be **CALLed** from external processes or *hooked* into interrupt vectors, because the assembler package code segment can and will change during interpreter operation (but not while your assembler code is executing).

Data tables and other structures may be included in the assembler package and accessed off of DS, ES or CS (as passed by MegaBasic). NEAR subroutines may also be included in the package for use by any of the formally declared subroutines.

The CPU stack (i.e., SS:SP) has sufficient space for reasonable use by your subroutines. However, if you need more than about 100 bytes of stack space (i.e., 50 **PUSHes**), you should define your own local stack and switch to it at the appropriate time. Be sure that you restore the original stack before your subroutine returns.

Returning From A Subroutine

To return from a subroutine, a **FAR RET** instruction must be executed with the cPu registers and other return structures set up in the following way:

- SS:SP must point to the same stack level that it pointed to upon entry into the subroutine. It points to a **FAR RET** address back to MegaBasic.
- DX must contain the number of bytes returned in the result returned. This only applies to functions; when returning from a procedure register DX is not used.
- The result of a function must be stored at location specified by SS:BX at the time the function was entered (see below).
- The *Carry* flag must be cleared **(CLC)** if no error is being reported.
- If an error is being reported, the *Carry* flag must be set **(STC)** and DS:SI must point to the error code and message to be reported. The format of this message is described below.

An error message that DS:SI points to consists of an error code **(BYTE)** followed by a zero-terminated error message string not exceeding 32 characters. The error code and message are processed exactly like MegaBasic errors and the error will be reported as if it occurred in the MegaBasic statement that invoked the assembler subroutine. A zero error code is not trapped and indicates a *fatal error* that immediately terminates the program. Type 10 errors are assumed to be related to errors in program syntax and are trappable in the RUN version only. Although you can specify any error code from O to 255, you might consider using only codes above 100 so that your programs can always discriminate between MegaBasic errors, which never go that high, and errors originating from your assembler packages.

No other registers need to be restored upon return. If a function result is to be returned to the caller, it must be copied to the location SS:BX provided at entry time and DX must be set to the number of bytes returned in the result. This length must *Never* be longer than the maximum result length declared in the function header. The result area is allocated to the maximum size indicated by the result-length field in the subroutine header just before the subroutine is entered, so you can use that area for temporary storage at any time before the final result is placed there.

Assembling a Package

To simplify the understanding, construction and assembly of MegaBasic assembler packages, there are four files the come with the MegaBasic release disk that will greatly assist you. These files are:

EXAMPLE.asm	A complete MegaBasic assembler package that implements a number of simple but useful procedures and functions and demonstrates most of the concepts involved in forming assembler packages.
ASMDEFS.asm	An assembler file that should be **INCLUDED** at the beginning of every assembler package the you create (it is **INCLUDED** by **EXAMPLE.asm**). It defines all the subroutine types, all the argument types and other useful entities including the package header itself that must appear in from of every package.
ASMPKG.bat	An **MS-DOS** *batch* file that assembles, links and converts MegaBasic assembler packages from the source file to the finished package. To use, simply type: ***ASMPKG asrmfile1 asmfile2 asmfile3 ...*** Each file is processed independently, one at a time. The source file names are assumed to have the **.ASM** extension. Do not type the **.ASM** extension on the names in the command. This process requires the **MASM** assembler, **LINK** linker and the **EXE2BIN** conversion utilities.
ASMCHK.pgm	Utility to check and display the internal structure of a completed assembler package. It shows the name, type and argument structure of each subroutine. To use it just **RUN** the program and supply the package name in the command line (default extension is **.BIN**).

The assembler files provided were prepared using the **MASM 5.x** assembler from *MicroSOFT Corporation*. **ASMPKG** produces binary files with the file extension of **.BIN**, which you must specify in file names supplied to **INCLUDE**, **ACCESS** and **DISMISS** statements. You could rename such files with a **.PGM** extension which MegaBasic adds by default, but this would probably create confusion with the MegaBasic interpreter files in the same directory.

You should study the **EXAMPLE.asm** source file to help understand the material presented here. Although the volume of information we have discussed may seem complex, once you see how it is done and begin to write some of your own assembler packages, you will find it to be about as simple as programming in assembler could ever be. We highly recommend that, at least for your first few packages, you build packages by making a copy of the **EXAMPLE.asm** source file and then modify that copy to suit your needs (deleting those portions that you do not require). It is always easier to modify an existing, correct, running program than to build one from scratch.

Appendix A Error Messages

This appendix describes all error types and messages reported by MegaBasic. Errors are reported with a descriptive *message,* a non-zero *error code* and the *location* in the program where the error occurred. Errors with codes less than 255 can be trapped by the program and handled by user-prescribed actions. Errors with a 255 error code cannot be trapped and constitute *fatal* errors. It is not possible or feasible to recover from errors of this type and error traps have no effect if set. They are usually revealed during the debugging phase of program development and do not occur in well tested final versions of programs.

The **ERRSEI** statement (Chapter 6, Section 4) is used to set traps for errors that later occur. When trapped, MegaBasic branches to a user-specified program location and provides information about the error in the following functions:

ERRLINE	Line number in which the error occurred.
ERRPKG$	Name of the package or workspace where the error occurred.
ERRTYP	Error type code of the error (see below).
ERRMSG$	Error message string that would have been displayed, had no **ERRSET** trap been in effect. Only the descriptive part of the message is returned.
ERRDEV	Device or file number selected at the time the error occurred. The error may or may not be related to I/O, but when it is, knowing the device can be useful.

When an untrapped error occurs (fatal or otherwise), MegaBasic reports the error message and its program location on the console screen and terminates the program. The text of the program line itself is placed in the edit buffer, so that you can immediately use editing control characters to examine and modify the offending line after the error is reported. MegaBasic also puts you into the workspace containing the package where the error occurred so that you can immediately examine the problem.

To assist the program development and debugging process, MegaBasic does not trap type 10 errors when programs are **RUN** from the MegaBasic command level. Type 10 errors are those involving errors in program formation, syntax, loop construction, etc. Such errors need to be exposed during program testing and not hidden by the error processing mechanisms, as they would be if they were *trappable* errors. Such errors are always trapped when the program is run from the operating system command level (under either *run* or *development* versions).

Certain trappable error types have the potential for recovery by retrying the same operation after waiting some amount of time and/or physically adjusting computer system components. A good example of this is getting a *Not Ready Error* when the printer

paper runs out. Retries can be controlled using the **RETRY** statement (Chapter 6, Section 4). This mechanism only applies to those trappable errors below which are marked with an asterisk (*). Read the discussion on the **RETRY** statement for further details.

Argument List Error (10)

The actual argument list of a user-defined function or procedure does not correspond to its formal definition or is otherwise improperly formed.

Array Subscript Error (1)

An array was specified with the wrong number of subscripts or a subscript position specified was outside the range defined for that array dimension.

Attempt to Read Endmark Error (21)

A file endmark code was encountered during the **READing** or **INPUTTing** of (non-binary) data from a file. Endmarks may be employed as end of record marks and hence do not necessarily imply **READing** past the final end of the file. MegaBasic does not normally generate endmarks, you can control them using the **NOMARK** statement.

Buffer Update Error (255)

Disk error encountered when attempting to update file buffers at program termination. No line number is associated with this error. The offending file is **CLOSEd** without updating its buffer, losing all new data it contains (512 bytes maximum). If other buffers contain information destined for this file, their contents is also lost. This problem is most likely to occur when information is appended to files that reside on a disk without any free space. Other error messages may immediately precede this one.

Command Argument Error (255)

A missing or improper argument or operand was supplied to a MegaBasic command. It can result from a program line number argument that was followed by some character other than a comma (,) or a dash (–).

Continue Error (255)

Attempted to **CONTinue** execution of a program without being in the state of temporary suspension left after a Ctrl-C or programmed **STOP**. Non-trivial major modifications to a **CONTinuable** program source can result in loss of the ability to **CONTinue**. This error only occurs in the command level, not during program execution.

Ctrl-C Stop (15)

A Ctrl-C was typed to abort program execution or a programmed **STOP** statement was executed. This is not an error, but Ctrl-C can be trapped as an *error* type 15 (**STOP** statements are not trappable). **PARAM(1)** can be set to enable or inhibit the detection of a console Ctrl-C. The program can be **CONTinued** after stopping from either cause.

Data Type Error (4)

Data specified for an operation was of the wrong type: usually a string (or number) was given where a number (or string) was expected.

Denied Access Error (36)

The operating system would not let your program **OPEN** or **RENAME** an existing file or device. This can result from attempting to open a read-only file for writing, attempting to **RENAME** a device (i.e., devices cannot be renamed) or making some other request prevented by protection mechanisms active on that file.

Device I/O Error (35)

An error was reported by a system device driver while your program was using it, or a device was **OPENed** under one of the built-in device numbers (i.e., O, 1 or2).

Directory Not Found Error (34)

A directory pathname could not be found on the drive as specified. See Appendix B and your operating system users manual for information about path names.

Disk Full Error (8)

All disk space was exhausted before finishing the requested operation or an access to an area beyond the bounds of the physical disk region available was attempted.

Disk Unavailable Error(33*)

An attempt was made to access a disk unit which was non-existent, not ready for access, or locked by another process temporarily. This error is an operating system dependent error that may or may not be supported.

Divide by Zero Error (9)

An attempt was made to divide a number by zero.

Double Definition Error (255)

The same name was used to identify more than one procedure, function, line label or **SHARED** variable. All these objects are bound to their assigned names at startup time, before the execution of the first program statement. Any violations of this rule are reported at this time.

Exit Error (10)

An **EXIT** statement was encountered without any **FOR**, **WHILE** or **REPEAT** loop currently active.

Expression-Depth Error (10)

Too many levels of parentheses during the evaluation of a string or numeric expression. Around 20 levels of parentheses are supported.

File Already Exists Error (6)

A file name specified for a new file **(CREATE)** or for renaming an existing file **(RENAME)** was actually present in the file directory.

File Busy Error (26*)

An attempt was made to **OPEN** a file which was already **OPEN** by another process for its own exclusive use. This can only occur under a multi-user or multi-tasking operating system or under local area networks **(LANS)**.

File Creation Error (18)

The operating system will not honor a request to create a new file directory entry. This can be due to a disk whose directory is already filled to capacity, or the file already exists, or the disk or directory marked as read-only, or an attempt to **RENAME** a file to a different drive, or some other system problem.

File Not Found Error (7)

A file name was specified for a existing file which was not found in the file directory. Misspelling a file name or omitting the source drive code or path from the name will cause this error, or the **DOS** command shell (i.e., **COMMAND.com**) could not be located so that a **DOS** command could be executed.

File Not Open Error (20)

An attempt was made to access an **OPEN** file or device using a file number not assigned by a previous **OPEN** statement.

File Number In Use Error (19)

An attempt was made to **OPEN** a file using a file number already in use. This error can be trapped to test file numbers for availability, or the file number can be tested with the **OPEN$()** function, which returns a null string if that number is available.

File System Error (30)

An inconsistency was discovered while performing file operations. This error should never occur and represents a problem in MegaBasic itself. Please report it immediately to your MegaBasic representative along with a description of how to re-create the error.

Floating Point Operand Error (37)

The floating point processing hardware (e.g., 8087, 80287 or 80387) reported that an invalid operation was requested by the **CPU**. This error should never occur and usually represents a hardware malfunction, a software error within MegaBasic itself, or, most likely, a result of providing improper data to an 8087 / 287 / 387 math coprocessor, such as denormalized numbers, infinity values, not-a-number representations, and others. Such values can result from reading IEEE values from wrong file locations or from incorrectly written files, **EXAMining** incorrect real values from memory, or from real fields in structured variables that have been erroneously set by non-floating point assignment statements. MegaBasic takes steps to prevent improper floating point values from arising out of numerical calculations, hence your source data is virtually the only culprit.

Format Specification Error (5)

An unknown or impossible numeric format was specified, such as a width that is too narrow for the number of decimals requested, or illegal characters encountered in the format string.

Illegal Operation Error (38)

An illegal operation was specified in defining logical interrupts (Chapter 7, Section 4), given the way that they were previously set up.

Illegal Package Operation Error (24)

A package attempted to **DISMISS** either itself or the main program, or you attempted to **LOAD** a scrambled (hidden) or assembler package into memory or to **SAVE** a scrambled or assembler package already in memory (left over from running a program).

Improper Filename Error (17)

A file name has been specified which in some way violates the rules for forming file names as defined by the host operating system.

Improper Vector Error (39)

A vector variable was specified using or containing an undefined name (i.e., a default variable), or the name of a string, procedure, function, or line label.

Incomplete Definition Error (255)

Some essential portion of a DEF statement or the construct it defines is missing.

Insufficient Memory Error (255)

The total amount of memory available to your program and its data has been consumed before the completion of the current operation. Unless the actual memory size available in your machine is severely limited, you should be able to scale down some of your large array and/or string variables to provide more free space and prevent this error from occurring.

Internal Stack Error (255)

The scratchpad memory stack that maintains loop structures, returns locations for active subroutines and intermediate calculations has been left in an unusable state. This error should not normally occur and usually means that MegaBasic tried to recover from an error in your program, but the transient memory stack could not be properly aligned to the recovery routine specified by a earlier **ERRSET**. You should examine the region around the reported program location to determine the true cause of the error.

Internal System Error (255)

Some erroneous condition has been detected within MegaBasic itself which prevents further program execution and from which recovery is impossible. This error should never occur during normal operations and may indicate a serious problem in MegaBasic, or a corruption of internal data structures due to an incorrect **FILL** statement or hardware malfunction. **Do not** attempt to continue work with the present execution copy of MegaBasic. If you can rule out **FILL** statements and hardware problems, please report in writing the circumstances which lead to this error to your MegaBasic representative or dealer in as much detail as possible so that a correction can be made.

Interrupt Service Error (255)

In context with a **SERVICE** routine, one of the following conditions produces this error:

- More than 16 distinct interrupt numbers have been assigned.
- Invoking a **SERVICE** routine that is no longer defined in the program or invoking it from the same copy of MegaBasic that assigned it.
- Invoking a **SERVICE** routine in a MegaBasic program that is currently processing a **SERVICE** routine.

Length Error (16)

The length of a string is too long or too short for the intended operation, or the length of a result or specification is not defined for the operation. For example, strings arguments passed by value to user-defined functions or procedures must be within the maximum length of the formal string parameter defined. This error also occurs if you attempt to load an ASCII text program file longer than 65535 characters.

Line Number Error (10)

A line-number was used in the program which referred to a line number not present in the program. Such errors occur when encountered during execution because MegaBasic binds line number references to their target lines when they are first encountered.

Local Declaration Error (10)

A **LOCAL** statement was encountered in a loop (**FOR**, **WHILE** or **REPEAT**) or outside of any active function, procedure, **GOSUB**, *prologue* or *epilogue.*

Loop Index Error (10)

The index variable provided by a **NEXT** statement didn't match the index variable specified in the opening **FOR** statement, or the index variable in a **FOR** was not a valid variable type (e.g., an array element or string variable), or an index variable was supplied on the closing **NEXT** of a **WHILE** or **REPEAT** loop.

Loop/Case Overlap Error (10)

A CASE statement block partially overlapped with a **FOR**, **WHILE** or **REPEAT** loop. Block and loop structures may be nested but not overlapped.

Missing Argument Error (10)

An essential operand or argument expression is missing from a statement, function, procedure or from either side of an operator.

Missing Bracket Error (10)

A closing right or left bracket bracket was not found as expected, usually due to a compound **THEN** or **ELSE** clause that was never opened or closed properly.

Missing CASE END Error (10)

A **CASE BEGIN** statement appeared that was not followed later on by its matching **CASE END** statement.

Missing DATA Statement Error (11)

A program data **READ** statement attempted to access a **DATA** statement after the last one had already been read, or in a program containing none. Use the **STAT** command to find out where the **DATA READ** pointer is during program execution.

Missing NEXT Error (10)

A **FOR**, **WHILE** or **REPEAT** loop was not followed later by its matching **NEXT** statement.

Missing Parenthesis Error (10)

A string or numeric expression ended without closing all the parenthesis levels it began.

Missing Return Error (10)

The physical end of a user-defined procedure or function was reached without encountering a **RETURN** statement.

No Program Error (255)

With no program in memory, a command was issued which required the presence of a program in memory.

Non-recoverable Disk Error (27*)

The system was unable to complete a physical disk read or write operation, due to a hardware failure or other condition beyond its control.

Not Ready Error (25*)

An attempt was made to access some peripheral device on the system which is not on-line or not otherwise available.

Numeric Overflow Error (14)

A computation resulted in a value too large to represent in MegaBasic floating point format. Numbers too small to represent are automatically converted to zero, causing no error for that case.

This error is also reported whenever you supply a floating point value to an integer context (e.g., integer assignment statement) and it was too large to be converted to integer representation. Such values must always lie in the range from –2,147,483,648 to 2,147,483,647.

Operating System Error (29*)

The host operating system ran out of some critical resource prior to completing the requested operation. This can occur when the number of **OPEN** files, file locks, or other limited resource has been used up and more are requested by your program. Under **MS-DOS**, this error is almost always the result of failing to ask for enough files in your **CONFIG.SYS** setup.

Out of Bounds Error (3)

A numeric value was specified that was either too large or too small for the intended operation.

Out of Context Error (10)

Some program object was used in an improper context: using a function as a procedure or operator, a procedure name as a line-label a line label or statement as a function, etc. This is a form of *Syntax Error.*

Out of Memory Segments Error (255)

The internal MegaBasic memory manager ran out of memory segments while trying to create space for a program or some other data object (e.g., a string or array). This error can only happen with an extremely large multiple package software system, an unanticipated operating system limitation or the corruption of the internal data structures of MegaBasic resulting from a hardware or software malfunction.

Pointer Variable Error (41)

Attempted to access some program *object* through an invalid *pointer* (Chapter 5, Section 4*)*. This can be caused by specifying non-integer variables or undefined variables as pointers, or by using numerical values that could not possibly refer to any program *object*. MegaBasic can usually validate pointer values *on the fly*, but not always (neither can C or other languages).

Program Compaction Error (255)

An error occurred during the automatic removal of spaces and **REMarks** performed by the **RUN** version on a newly loaded program about to be executed. This should never occur during normal processing and indicates an inconsistency in program internal encoding structure.

Program Too Big Error (255)

Indicates an attempt to extend the program source contained in the current workspace beyond 65535 bytes (64k). Reducing the length of long names will not correct the problem. You must reduce the number of lines or reduce the number of bytes in lines (e.g., remove extra spaces). Total program size may actually exceed 64k because a program is composed of two regions: one that contains the program lines (referred to by this error), and one that contains the program identifiers, numeric constants and other operational symbolic support data. The best solution to this problem is to break the program into two or more packages, providing plenty of room to expand.

Re-Dimension Error (2)

Attempted to **DIMension** a string or array in the process of being assigned a value at a higher level. This is reported in the assignment statement affected by the erroneous re-dimension, but not all instances can be detected.

Read Past End of File Error (22)

The physical end of file was encountered while **READing** or **INPUTTing** data from the file.

Read-Only Violation Error (28*)

An attempt was made to write on or modify a file, directory or disk that has been set to read-only or write-protected mode.

ScratchPad Full Error (13)

The internal scratchpad area used for evaluating string and numeric expressions and maintaining loop, function and procedure control structures has run out of room. Its currently available size can be obtained at any time using the **FREE(2)** function.

Shared Name Conflict Error (23)

A package being **ACCESSed** for subsequent use defines a **SHARED** name (of a variable, function or procedure) which is already defined and in use by the package requiring **ACCESS** to it. MegaBasic reports the name and the packages involved.

Structured Variable Error (40)

A structured variable field (Chapter 5, Section 3) was improperly defined or improperly referenced or used in a context that does no permit structured variables. This can be due to attempting to define fields at positions beyond 65535 or below zero, or fields too narrow to contain the data type specified, or attempting to define a procedure, function, line label or **SHARED** name of any kind as a structured variable component.

Suspended File Access Error (32*)

An attempt was made to read or write to a physical region of an **OPEN SHARED** file which was temporarily locked by another process. Retrying the operation will eventually succeed when the locking process releases the locked file region. This error is possible only on **OPEN SHARED** files under a multi-tasking operating system or local area network.

Syntax Error (10)

An improperly constructed MegaBasic statement, command or expression was encountered for execution. All other *type 10* errors are either syntax errors or other violations of program form and construction.

Too Many File Locks Error (31)

The last attempt to temporarily lock some region of an **OPEN SHARED** file exceeded the locking capacity of either MegaBasic or the operating system. MegaBasic can lock up to 64 regions among all the **OPEN SHARED** files, while the operating system may support more or less than that number. The error can only occur under operating systems that support record lock-out (e.g., Local Area Networks, Xenix, TurboDos-86, CurrentCP/M, etc.).

Too Many Symbols Error (255)

Usually caused by a new variable or subroutine definition being added to the symbol table after it fills up. Collectively, there is a limit of about 7000 symbols over all packages in memory. Use the **FREE(3)** function to determine the remaining room in this table. This error can also result from having too many different constants or user-assigned names in a package. No one module can have more than 2560 unique program constants, or more than 6656 user-assigned names (for variables, line-labels, functions or procedures).

Undefined Name or Procedure Error (10)

A procedure name or line-label was found that was not defined in the program using the spelling encountered.

Unexpected Argument Error (10)

An extra argument was encountered during the processing of a statement, function, procedure or expression.

Undeclared Array or String Error (42)

An attempt was made to access an array or string that was never **DIMensioned**. Normally, **unDIMensioned** arrays and strings are created automatically by default when your program accesses them the first time. However, if **PARAM(12)** or **PARAM(13)** is set to less than one, this *default* creation feature is disabled and attempts to access undeclared arrays and strings are reported as this error (Chapter 9, Section 5).

Unexpected Bracket Error (10)

Encountered a closing bracket for a compound **THEN** or **ELSE** clause when no such clause was active.

Unexpected CASE Error (10)

Encountered a **CASE END**, **CASE EXIT** or a **CASE** selection branch without previously opening a **CASE** block with a **CASE BEGIN** statement.

Unexpected NEXT Error (10)

A **NEXT** statement was encountered with no active **FOR**, **WHILE** or **REPEAT** loop present.

Unexpected Parentheses Error (10)

A closing parentheses was encountered before any opening parenthesis in an expression.

Unexpected Return Error (10)

A **RETURN** statement was encountered without any procedure, function or **GOSUB** actively underway.

Unexpected THEN/ELSE Clause (10)

A **THEN** or **ELSE** clause was encountered without first evaluating an **IF** condition. This usually results from a multi-line **IF** statement that is improperly formed or from a **GOTO** whose target line begins with the word **THEN** or **ELSE**.

Unintelligible Program Error (255)

An attempt was made to **LOAD** (or **ACCESS**, **MERGE**, **INCLUDE**, **LINK**, etc.) a program file which contained no recognizable binary or ASCII program. The **RUN** version reports this error if anything but a binary program is accessed.

Unknown Command Error (255)

A command was issued which was either misspelled or unavailable under the version of MegaBasic being used. Misspelling a user-defined procedure name at the start of a direct statement or running a command name and its leading argument together without any *white space* in between will also lead to this error.

Unsupported Feature Error (10)

A MegaBasic feature that is supported under some operating system environments was used under an environment that does not support it.

User Trap Error (255)

An **ERRSET#** statement was executed with no **ERRSET** trap in effect. If the **ERRSET#** specifies a custom error message then that message will replace the *User Trap* portion of the message. This error represents the reporting of a user-specified specified error that was not trapped.

Value Conversion Error (12)

An ASCII string intended to be a representation of a numeric constant could not be converted to a number, due to improper number formation. This can occur from a **VAL()** function or from **INPUTTing** ASCII numeric values from a text file.

Write- Only Volition Error (28)

An attempt was made to read or input from a file that was opened in output-only or write-only mode.

Error Messages by Error Code			
255	Buffer update error	**10**	Loop/Case overlap error
255	Command argument error	**10**	Missing argument error
255	Continue error	**10**	Missing bracket error
255	Double definition error	**10**	Missing CASE END error
255	Incomplete definition error	**10**	Missing NEXT error
255	Insufficient memory error	**10**	Missing parenthesis error
255	Internal stack error	**10**	Missing RETURN error
255	Internal system error	**10**	Out of context error
255	Interrupt service error	**10**	Syntax error
255	No Program error	**10**	Undefined symbol error
255	Out of memory segments error	**10**	Unexpected argument error
255	Program compaction error	**10**	Unexpected bracket error
255	Program too big error	**10**	Unexpected CASE error
255	Too many symbols error	**10**	Unexpected NEXT error
255	Unintelligible program error	**10**	Unexpected parenthesis error
255	Unknown command error	**10**	Unexpected RETURN error
255	User trap error	**10**	Unexpected THEN/ELSE clause error
1	Array subscript error	**10**	Unsupported feature error
2	Re-Dimension error	**11**	Missing DATA statement error
3	Out of bounds error	**12**	Value conversion error
4	Data type error	**13**	ScratchPad full error
5	Format specification error	**14**	Numeric overflow error
6	File already exists error	**15**	Ctrl-C Stop
7	File not found error	**16**	Length error
8	Disk full error	**17**	Improper filename error
9	Divide by zero error	**18**	File creation error
10	Argument list error	**19**	File number in use error
10	Exit error	**20**	File not open error
10	Expression-depth error	**21**	Attempt to read endmark error
10	Line number error	**22**	Read past end of file error
10	Local declaration error	**23**	Shared name conflict error
10	Loop index error	**24**	Illegal package operation error
25*	Not ready error	**34**	Directory not found error
26*	File busy error	**35**	Device I/O error
27*	Non-recoverable disk error	**36**	Denied access error
28*	Read-only violation error	**37**	Floating point operand error
28	Write-only violation error	**38**	Illegal operation error
29*	Operating system error	**39**	Improper vector error
30	File system error	**40**	Structured variable error
31	Too many file locks error	**41**	Pointer variable error
32*	Suspended file access error	**42**	Undeclared array or string error
33*	Disk unavailable error		

Appendix B Other Operating Systems

MegaBasic runs on a variety of different operating systems. Each operating system has its own peculiarities which are difficult, if not, impossible to reconcile with one another. Hence there are some slight differences in certain capabilities and operational rules between versions of MegaBasic that run under different operating systems. All such differences are related to operating system services and do not affect the underlying language constructs.

The *host* operating system type can be determined from within your program by interrogating the the **PARAM(5)** function (Chapter 6, Section 2), which returns an integer code corresponding to each operating system supported by MegaBasic. Operating system dependent routines in your program can use this code at any time to select the implementation appropriate to the operating system at hand. In this way, you can develop generic versions of your programs that will execute properly regardless of the system you run them under.

As the **MS-DOS** operating system is the most prevalent system in current use, the MegaBasic Reference Manual describes MegaBasic from the **MS-DOS** point of view. Hence Appendix B describes how MegaBasic under other operating systems differs from the **MS-DOS** implementation. Appendix B is organized in the following way:

Section	Description
Appendix B Section 1	Xenix 386/486 System V
Appendix B Section 2	CP/M-86 on 8088/8086 machines
Appendix B Section 3	Concurrent CP/M-86 and MP/M-86
Appendix B Section 4	TurboDos-86

Section 1: Xenix 386 System V

MegaBasic under Xenix 386 System V is very close to MegaBasic under MS-DOS, as MS-DOS itself was designed with Xenix compatibility in mind. We will describe in this section how MegaBasic under Xenix differs from MegaBasic under MS-DOS. Because MegaBasic is a living, growing language, some of the differences described below may disappear in future releases of MegaBasic.

Features That Differ from MS-DOS

- File and directory pathnames under Xenix use a forward slash (/) to separate the pathname components, as compared to a backward slash (\ ) required under MS-DOS. Also, drive letters (e.g., B:FILE) are ignored in file names used in Xenix MegaBasic programs.
- PARAM(18) was added to provide upper / lower case conversion to filenames before they are passed to the operating system. This is useful when running MS-DOS MegaBasic programs under Xenix 386, because MS-DOS does not care about case (case-insensitive) but Xenix does (case-sensitive). Setting PARAM(18)=1 forces all names to lower case, PARAM(18) =2 forces them to upper case, and PARAM(18)=0 passes all filenames on to Xenix without any case conversion. Xenix MegaBasic begins execution with PARAM(18)=0. PARAM(18) has no effect upon the directory pathname portion of file names.
- The DIR statement is supported by invoking the Xenix Ic command. If you specify a string argument to DIR, it will be used as the command tail to the lc command. If you specify a device number in DIR (e.g., DIR #1), the output of Ic will be redirected to that device.
- PARAM(5)returns 11 to indicate the Xenix 286 version of MegaBasic.
- After executing a DOS command, the error status of the command executed can be accessed through the ERRTYP function. The value returned is the Xenix exit status of the SYSTEM( ) call, not a MegaBasic error code.

Features Unique to Xenix MegaBasic

- Up to 8 megabytes of memory are available to user programs and their data.
- When you PRINT to device #1 or #2, MegaBasic sends all characters to a file instead of an actual device. This file is named / tmp / prnxxx, where xxx is the process identification number of the MegaBasic process. Concurrent invocations of MegaBasic will therefore use different prnxxx files. PRINTed output will accumulate in the prnxxx file until MegaBasic terminates or a PRINT END statement is executed.
- At this time, MegaBasic passes the prnxxx file to the Xenix spooler process using the system command: *Ipr -c prnxxx*. Following this action, the prnxxx file is deleted. If a PRINT END is issued, MegaBasic goes on to create a new prnxxx file and again associates it with devices #1 and #2.

- By modifying the system command that passes the prnxxx file to the spooler, you may be able to redirect the output to devices other than the standard printer. The PRINT END <*string*> statement can be used to specify such modified printer commands. The <*string*> is a string expression that evaluates to the desired command prefix, to which MegaBasic appends the file name currently in effect. For example the string *lpr -c* specifies the default command as described above; the string *more* causes the file to be displayed on the console screen. Note that PRINT END terminates the current printer session and begins a new one, and that a command specified by the optional <*string*> applies to the new printer session rather than the terminated one.

Features not Supported

- The IOCTL statement is not supported and the IOCTL( ) functions always indicate that any I/O device does not support IOCTL strings.
- I/O redirection from the MegaBasic command line is not currently supported completely, but data transfers are passed through.
- DIR$() and SUBDIR$( ) functions are not supported. However, directories can be opened (read-only) and their contents accessed to determine what files and subdirectories are present and their names extracted and used as needed.
- RETRY procedures may be defined, but they are currently never invoked by the error processing system.
- The BASIC command from MegaBasic is not supported. However you can easily invoke BASIC using the *DOS "basic"* command at any tlme.
- The CONFIG, PGMLINK and CRUNCH utilities are not currently supported under the Xenix 286 operating system (although they may be supported in future versions).
- The SERVICE statement is not supported and the CALL and CALL#, although available, will cause protection violations and immediately terminate MegaBasic. You should use the request.c interface described elsewhere to access system services and other software not directly provided by MegaBasic.
- The IBM keyboard function keys and cursor control keys do not generate the standard two-byte sequences under Xenix 386. Therefore these keys are not recognized by MegaBasic for editing purposes. You must use the various CTRL keys for editing.
- The logical interrupt system is not currently supported because the CPU hardware interrupt vector region is protected from modification by user programs by the 80386 virtual mode protection mechanism. In future versions of MegaBasic, there may be an analogous capability that can be implemented for use under Xenix.
- Any attempt to directly access hardware I/O ports (through INP( ) and OUT) or memory locations outside MegaBasic data structures (using FILL and EXAM) will be trapped by the 80386 protection mechanism and terminate the MegaBasic process immediately. Such operations cannot be supported in this environment.

Section 2: CP/M-86 On 8086/88 Machines

DOS Statements

CP/Mhas no command shell processor, hence you cannot execute operating system commands from your MegaBasic program. Furthermore, the background processing supported under PC-DOS is unavailable under CP/M.

DIR = *<user number expression>*

This statement selects the CP/M user number for all subsequent file operations. The user number is specified by a numeric expression that evaluates to an integer from 0 to 15. The user number of any file must be selected prior to accessing the file.

FILEDATE$( ) and FILETIME$( ) functions

Neither of these functions is supported under CP/M. If called, they both return a null string.

SPACE( ) and FILESIZE( ) Functions

The two functions return disk and file size in units of 256 byte blocks, instead of bytes as under MS-DOS. Because of this, you have to be careful appending information to a CP/M file because it is difficult to know exactly where the end of the file really is.

FILESIZE( ) cannot appear on the left side of an assignment statement because CP/M does not support changes in file size except for extending it by writing past the end of the file.

DOS Exit Codes

Termination codes are not supported by CP/M-86, but they are supported under MP/M and ConcurrentCP/M.

Section 3: Concurrent DOS and MP/M-86

Concurrent CP / M is a multi-tasking single-user operating system version CP / M-86. MP / M-86 is a multi-user, multi-processing version of CP / M-86. All three of these operating systems are supported by the CP / M version of MegaBasic (i.e.. one BASIC runs under all three). This appendix will describe only those few differences that exist when running MegaBasic under Concurrent CP / M or MP / M-86, as compared with CP / M-86. All these differences stem from the enhanced features of Concurrent and MP / M-86 over CP / M-86 and the requirements imposed by the support of shared resources (e.g., files, queues, printers, consoles, CPU and memory).

At startup, MegaBasic determines which of these operating systems is running so that subsequent operations take the appropriate action. MP / M-86 and CCP / M are virtually identical from the users view from within MegaBasic. However CCP / M users should assign the **SHARED file attribute** to their MegaBasic so that one resident copy of MegaBasic is shared among multiple *contexts*. See your operating system manual for instructions on how to do this. When MegaBasic is waiting for keyboard input under both MP / M-86 and CCP / M, it lets the system do the waiting so that time-slices are not unnecessarily burned up.

Very few differences exist at the present printing of this manual and future releases of MegaBasic may include more enhancements than those described below. Programs which run under CP / M-86 will run without change under MP / M-86. With more than one user competing for the same resources however, the following mechanisms are provided to control access:

System Printer (device # 1)

As soon as you use the printer, by sending at least one character to device #1, the printer will be attached to your program. If however the printer is in use by another user, a Not Ready Error will be generated by MegaBasic. This error can be trapped via an ERRSET statement as a type 25 error. Once you have successfully sent at least one character to the device #1, your program will *own* the printer until your program ends and MegaBasic is exited.

In programs that execute for an extended period of time (or even continuously), it may not be desirable for the printer to be tied up by the program indefinitely. Therefore the additional statement PRINT END may be issued to release the printer so that other users may access it while your program no longer needs it. At any subsequent time however, your program may again grab the printer by merely printing to device #1 as described above.

Because it is important to not monopolize the printer unnecessarily, your programs should very clearly define all printing requirements as a three step process:

- Make sure the printer is available. Use ERRSETs to continuously retry the initial output to device #1 until it succeeds, or maybe inform the user and request what action to take.
- Perform all the required printing transfers.
- Release the printer for others to use with a PRINT END statement.

File and Record Locking Facilities

MP / M-86versions of MegaBasic support the system of private and shared file and the automatic record lockout capabilities (Chapter 7, Section 2). This is a consistent set of facilities that MegaBasic supports under various multi-user and network operating systems. You should read that material for further information.

Section 4: TurboDos-86

TurboDos-86 is almost identical to MP / M-86, and this appendix explains only the differences between those two operating systems. See Appendix B, Section 4 for all features identical in MegaBasic under both systems.

TurboDos routes all printer characters through the FIFO file assigned to the current user or the printer. This makes the printer always (logically) available, obviating the need for printer locking mechanisms. This does however mean that there may be a delay (possibly significant) between the time that printed material is sent and the start of its physical printout. PRINT END does nothing if it appears in any MegaBasic programs.

DIR = *<user number exprn>*

Under TurboDos-86, user numbers may range from 0 to 31, rather than 0 to 15 as in MP / M-86andCP / M-86.

RETRY Procedures

TurboDos does not allow user programs to control all the errors that can occur from certain file operations. In such cases, TurboDos takes control away from the user program rather than informing the user program so that it can take appropriate action. The only retriable errors supported by TurboDos are the locked access errors that can occur when more than one process is attempting to access the same region of a file OPENed in SHARED mode or OPEN files which are already OPEN in exclusive (non-SHARED) mode by another process. Several important errors, however, are not recoverable. These are listed below:

- Drive select errors cause an abort back to the operating system level like under CP / M-86, rather than being trapped by MegaBasic as in MP / M-86.
- Bad sector errors can be retried using the TurboDos error handler, rather than the MegaBasic RETRY capability.
- Read-Only errors cause an abort back to the operating system level.

Appendix C — Utilities and Other Software

This section concerns itself with a number of programs external to MegaBasic that perform functions useful to the development process.

Section	Description
Appendix C Section 1	**PGMLINK**: utility for combining a MegaBasic program and any supporting packages with a **RUN** version into a stand-alone execute-only single file program.
Appendix C Section 2	**CRUNCH**: a program image compactor that reduces program size by 30% and more through removal of blanks and **REMarks**.
Appendix C Section 3	**CONFIG**: a configuration program to set various options in your versions of MegaBasic.
Appendix C Section 4	**FLIP**: utility providing MegaBasic with a separate program development and debugging screen from the application output screen.
Appendix Section 5	Utilities for generating real-time events on standard PCs and processing them using the logical interrupt facilities of MegaBasic.
Appendix C Section 6	MegaBasic packages that come with MegaBasic and **ACCESSed** from your program to provide a variety of special functions and procedures.
Appendix C Section 7	Summary of the currently available MegaBasic and related products.

Section 1: Stand-Alone Programs with PGMLINK

PGMLINK binds the currently running image of MegaBasic with a series of MegaBasic packages into a single, stand-alone, execute-only file. Such a program may then be run without having any copies of MegaBasic on the disk, by merely typing its name at the command level of the operating system, like the other utility programs that came with your machine. In fact, such programs are indistinguishable from those produced by compilers and assemblers, as far as their surface appearance is concerned.

To run this utility, you must be prepared to enter the file names of the main program and the packages which it accesses. You must also be running **PGMLINK** under the same version of MegaBasic that you want executing your final program. **PGMLINK** will display the floating point precision and operating system type of the version of MegaBasic you are running under before you enter the names of your program files, so that you can abort the process if necessary.

PGMLINK first requests the name of the file on which your complete program system will be built. If it already exists, then its contents will be automatically erased before proceeding. **PGMLINK** then requests the file name of the main program and the names of the subsequent support packages. When all names have been entered, type a carriage return to the file name request to terminate the program. Your stand-alone program will then be ready to run.

Your program will run the same way as it would as if you executed it under the **RUN** version. It can still **ACCESS** or **INCLUDE** other program files that are on the disk if needed. All packages that you bound together in the executable file are brought into memory at the same time, and hence there must be sufficient memory available to load all of them and execute the entire program. If any packages contained in this file are **DISMISSed** and removed from memory, they must reside on another separate disk file if they are ever needed again by a subsequent **INCLUDE** or **ACCESS** statement. No program component packages can be individually extracted from the execute-only file once they have been bound together.

There is one minor difference in the way execute-only programs behave that you should be aware of. The command tail that the operating system communicated to any transient command (like **BASIC**, a text editor, **RUN**, etc.) consists of all characters typed immediately after the command name. If you use command tails, you know that the first word in every command tail is the name of your program, because it is typed immediately after the command **BASIC** or **RUN**. However, your execute-only program file name will be the command and hence it will not appear as the first word in the command tail, or anywhere else. There will always be one less word in command tails received by execute-only programs as compared with the same programs run under **BASIC** or **RUN** explicitly. This incompatibility is important if you pass arguments to your program in the command tail.

Under **MS-DOS**, console input may come from a text file instead of from the keyboard. This capability is quite useful when you are creating stand-alone programs with the **PGMLINK** facility. By preparing a text file using your favorite editor, you can store the names of the program files required for any **PGMLINK** process, so that you no longer have to type them in or even remember what they are. This is especially useful when your stand- alone program consists of many packages. Once your text file is prepared, type the following command at the operating system level:

```
RUN PGMLINK < pgmnames.txt
```

where *pgmnames.txt is* any file containing the necessary program file names. The left arrow (<) causes MegaBasic to take its standard console input from the specified file instead of the actual console keyboard.

The text file containing the list of program file names must be constructed in an identical manner to the way you would type the names from the keyboard in response to **PGMLINK** file name requests. This format is summarized as

Line #1	Name of the file to which the final resulting stand-alone program will be written.
Line #2	Name of the *main* program file.
Line #3	Name of the first supporting package file.
Line #4	Name of the second supporting package. You may continue to add additional lines as needed to list all of the supporting packages required.
Last Line	The last line must be a blank line to signal **PGMLINK** that no more file names are listed. For this reason, you must not have any other blank lines in this text file.

Another point about **PGMLINKed** systems should also be mentioned. When your stand-alone system begins execution, all the supporting packages will be in an uninitialized state: equivalent to *free* workspaces (see the **SHOW** command). If you perform any processing which would normally release *free* workspaces, your packages will be removed from memory before you may have had a chance to **INCLUDE** or **ACCESS** them.

DOS commands, in particular, will always get rid of all *free* packages in order to provide the greatest possible working memory to the subsequent **DOS** command. This can be avoided by making sure all of your supporting packages are **INCLUDEd** or **ACCESSed** before giving the **DOS** command.

Section 2: Program Compaction with CRUNCH

A surprising amount of memory space is taken up by blanks inserted into the code and **REMarks** that have nothing to do with program execution. Well commented, structured **BASIC** programs typically have 30% or more of their memory area invested in blanks and remarks. The **CRUNCH** program conveniently optimizes a **BASIC** program by creating a new **BASIC** program without spaces and remarks, leaving the execution properties unchanged. This utility is useful only when more memory or source code protection is desired.

MegaBasic performs this **CRUNCHing** process automatically when programs are executed with the **RUN** version. This utility program is included so that convenient storage of a **CRUNCHed** program on a disk file is possible for program secrecy or for extremely large programs that cannot even be **LOADed** without size reduction. **CRUNCH** is able to reduce programs about 10% further than the automatic method used by MegaBasic **RUN**. The features of **CRUNCH** are summarized below:

- Helps keep a program secret by removing all traces of internal program commenting and readable formatting. Additional code security provided by the scrambling option, which irreversibly creates a **RUN**-only version of your finished program that is impossible to **LIST** under any version of MegaBasic.
- Reduces program memory and file requirements by 20%-60% in only seconds. Such programs load faster and execute slightly faster. The memory saved increases the capacity for program variables and working storage.
- File-to-file conversion allows preservation of the original version.
- Deletes *all* **REMarks** from your program. Line number references to deleted **REM** lines (such as **GOTOS**, **GOSUBS**, or **ERRSETS**) are adjusted to the nearest non-**REMark** following the deleted lines.
- Spaces and line-feeds within quotes (“”) are preserved; all others are deleted.
- **CRUNCH** aborts if the source program has any unresolved line number references and displays the number of such occurrences.
- Program lines can optionally be *joined together* to eliminate most of the line numbers in a program to gain further size reductions.

How to Use CRUNCH

CRUNCH is a separate utility program intended to be run from the operating system, rather than from within MegaBasic. You simply type the name **CRUNCH** followed by the original file name then the result file name, as follows:

CRUNCH ***<program name><result file name>***

Both files may be typed with or without the **.PGM** program file name extension, and **.PGM** is assumed when omitted. **CRUNCH** aborts if the original program file cannot be found in the directory or if the original program and result files are the same. **CRUNCH** tells you if the result file exists or not, and requests confirmation before proceeding. **CRUNCH** creates a new file automatically as needed for the resulting **CRUNCHed** program.

You can specify the same file names for the original and the result files only when they are on different drives. **CRUNCH** will automatically derive such a result file name from the original if you specify the result file as a drive specifier only. For example, the two **CRUNCH** commands below both mean the same thing:

```
CRUNCH A:MYPROG B:

CRUNCH A:MYPROG B:MYPROG
```

Code Security by Scrambling

CRUNCH will request whether or not you wish to scramble your program. This option produces a **RUN**-only version of your program that can never be **LISTed**, **EDITed**, or otherwise accessed by anyone, and is provided for those users who require additional program source security. The technique used is intended to provide at least the same level of code protection for MegaBasic programs that complier languages implicitly provide for compiled programs. This scrambler requires no password, and is irreversible, so be sure that you select this option intentionally. **CRUNCH** always gives you a second chance to get out of this option.

The development version of MegaBasic (i.e., **BASIC**) permits **ACCESSes** of scrambled packages, so that large systems that use such packages can be tested and debugged without requiring the presence of unscrambled versions of the packages. Such packages are displayed by the **SHOW** command as *hidden* and you cannot *get into* these programs to look at the source. Scrambled packages cannot be **LOADed**, **SAVEd**, **LISTed**, or any other operation that could bypass the protection afforded by scrambling. **INCLUDE** and **ACCESS** are the only statements that can load scrambled packages into memory.

Section 3: MegaBasic Configuration with CONFIG

Your version of MegaBasic can be personalized in a number of respects to accommodate its operating environment and your needs. These options are implemented by making changes to certain memory locations within MegaBasic. The **CONFIG** utility program provided in the MegaBasic software package lets you selectively alter the available options. When you are satisfied with your choices, **CONFIG** will install your modifications onto all MegaBasics contained on one disk, or allow you to install onto some while bypassing the rest. You can re-configure MegaBasic whenever necessary. From the system command line, you can run **CONFIG** in batch mode using the following forms:

RUN CONFIG	Menu-driven,interactive mode.
RUN CONFIG *<directory path>*	Selects path and asks for options.
RUN CONFIG *<file path>*	Displayssettings for one MegaBasic file.
RUN CONFIG *<file path> <options>*	Sets options for one MegaBasic file.
RUN CONFIG *<directory path> <options>*	Sets options for all MegaBasics in the directory.

The *<options>* are of the form: *letter=value* (with no spaces). Multiple options are separated with spaces. Omit the *<options>* to display settings. The option letters are defined as follows:

	Option	ValueRange
c	AllowCtrl-Cabort	On or Off
b	UseCtrl-Break for Abort	On or Off
k	Consolemode	0 to 15
o	Maximum OpenFiles	8 to 127
l	File lockingsupport	On, Off, Auto
n	Number of FileBuffers	0 to 127
e	End-of-file mark code	0 to 255
f	Floating point file format	1 to 18
a	Defaultarray size	–1 to 255
s	Defaultstringsize	–1 to 255
u	Force Upper-CaseIdentifiers	On or Off
m	Maximum low-memoryKbytes	96 to 1023
z	Minimumhigh-memoryKbytes	O to 16384
h	Maximumhigh-memoryKbytes	O to 16384
x	Default pkg file extension	String
p	AutomaticPackage Access	String

When you are satisfied with your choices, you can then have all MegaBasics immediately configured at once, or you can have them configured one by one, individually subject to your confirmation.

Be sure to only modify *copies* of your original software while protecting your originals by keeping them safely away from your computer system. Each of the currently available options is described below.

Allow Ctrl-C Abort

Normally MegaBasic will always detect a Ctrl-C typed from the console and abort program execution. Using **PARAM(1)** you can disable or re-enable this mechanism. However for *turn-key* MegaBasic applications, Ctrl-C can be disabled at startup (prior to executing the 1st program statement). **PARAM(1)** may be later executed within the program to modify this state as needed. When disabled, Ctrl-C may be input as an ordinary control character (through **INCHR$**).

Use Ctrl-Break for Abort

When the Ctrl-C abort mechanism is enabled, all keys you type during program execution are examined for Ctrl-C and discarded, making them unavailable as type-ahead for subsequent inputs. Ctrl-Break is a different mechanism that does not have this limitation. Turning this option on makes Ctrl-Break the execution abort key and Ctrl-C a potentially available input character. **PARAM(1)** may be used to control recognition of a Ctrl-Break abort.

Console Mode Byte

This option controls several console features. To each feature, corresponds a separate value which are added together to form the console mode value. These values and their meanings are summarized below:

1	Slow Console Output. This is required by some so-called **IBM-PC** compatible machines which have a *bug* in their console driver. If this option is not selected, such machines will display nothing or display continuous garbage when MegaBasic is started. Therefore, this is the standard setting of MegaBasic but you can turn it off if it is unnecessary (resulting in a much faster console response on *block* output).
2	**IBM ROM BIOS** Supported (not necessarily on an **IBM** machine). This extra set of system calls is used on **IBM** Personal Computers and compatibles by the MegaBasic line editor. It is not required when using generic terminals, but you should use it if it is present. If you select the **ROM BIOS** and it is not present, the machine will *crash* as soon as Mega-Basic attempts to use it.
4	Automatic detection of the **IBM ROM BIOS** presence. If you select this option, option 2 above is automatically determined by MegaBasic (and its configured value is ignored). This is the standard factory setting of MegaBasic.
8	Indicates that **ANSI** sequences are supported for the console and causes the line backup operation to use **ANSI** sequences. This takes precedence over code 2 / 4 when both are specified. **ANSI** sequences are particularly important on non-**DOS** machines (e.g., **TurboDOS**, Xenix, CP / M96, etc.) using stand-alone terminals that do not support backspace wrap-around (the method used when neither **ANSI** or **ROM BIOS** is selected).

When you set the video mode byte, specify the sum of all options you wish to select. For example, the standard setting is 7 (1+2+4). This byte affects only the **MS-DOS** versions of MegaBasic.

Maximum Number of OPEN Files

MegaBasic is normally configured to support up to 32 open files and / or devices, under file numbers ranging 0 to 31. However some application may require more than this number and so you are allowed to change the open file limit to any number from 4 to 127. Be aware that the more open file capacity you have, the more memory is required and dedicated to this purpose (about 80 bytes per file, opened or not). You may also have to configure your operating system to support a large number of open files.

Number of File Buffers

MegaBasic normally begins execution with a number of file buffers based on the amount of memory available to the user. If, for specific applications, this is an inconvenient setting, your program can control this number via **PARAM(10)**, or the particular copy of MegaBasic can be configured for a specific number of file buffers at startup. You may specify 0 buffers for the default case (automatic sizing), or 4 to 127 fixed number of buffers. See **PARAM(10)** for further information about file buffers.

File Locking Support

This setting provides global control over file locking operations. Setting to **OFF** disables multi-user file locking services to files supported. Setting to **AUTO** enables shared / exclusive open files and enables MegaBasic automatic file locks on all file transfers. Setting to ***ON*** enablesshared / exclusive open files and disables MegaBasic automatic file locking mechanisms. Locking files that are opened **SHARED** much be done explicitly by the application using **LOCK** and **UNLOCK** statements.

End-of-File Mark Code

The 8-bit code normally used to indicate the termination of data on a file is a decimal 26. This code may be altered during program execution using **PARAM(9)** or by configuring its default value from the start. To avoid conflict with string or floating point values stored on the file, the endmark code should be one of the following values: 0,1, 26, or 154 to 255.

Floating Point File Format

You can set the initial value of **PARAM(11)** (Chapter 9, Section 5) that takes effect when MegaBasic begins execution. This value specifies the floating point format to use when transferring real numbers between data files and your program. You can specify **BCD** formats from 8 to 18 digits and **IEEE** binary formats 1 for single and 2 for double precision (**IEEE** versions only).

Undimensioned Array Size

When array variables are accessed without being previously dimensioned, their size is automatically set to 11 elements numbered 0 to 10. Most **BASICS** also provide a default upper limit of 10, but there are various reasons why other sizes may be useful. You can control this default upper subscript by setting **PARAM(13)** (Chapter 9, Section 5) and you can control the initial startup value of **PARAM(13)** by configuring it to any length from 0 to 1023. A value of –1 disables the creation of arrays without a **DIMension** statement.

Undimensioned String Length

When string variables are accessed without being previously dimensioned, their size is automatically set to 80 characters maximum. You can control this default size by setting **PARAM(12)** (Chapter 9, Section 5) and you can control the initial startup value of **PARAM(12)** by configuring it to any length from 0 to 4095. A setting of –1 disables the creation of string variables without **DIMensioning** them.

Lower-Case Symbols

You can configure MegaBasic to leave the letter case in names the way you type them instead of forcing them to upper case. In this mode, MegaBasic will show the *case spelling you* typed for the first reference to the name, no matter how you spell it in subsequent references. This lets you type a name in any case you like, but it will always **LIST** the same way throughout the program. You can change the case spelling of existing names using the NAME command (which affects all references). Shared names in other packages do not have to match the letter case of references to them, i.e., shared name linkage is case-insensitive.

This feature is for purely esthetic and programming style purposes. It will not affect program size or performance, but it can enable you to make your software *look* a little more the way you would like it to. MegaBasic versions earlier than 5.72 use case-sensitive name linkage, so avoid using this feature for programs to be executed under such earlier versions.

Maximum Low-Memory KiloBytes

MegaBasic normally uses all of the available memory in the machine as supplied by the operating system when you first start MegaBasic. Sometimes, this can be undesirable when you would like to reserve some memory for another purpose. This option allows you to specify how much memory to allocate to MegaBasic, which you enter in units of lk (1024 bytes). Since MegaBasic requires a minimum of about 96k just to *come up, you* must specify at least this minimum or **CONFIG** will not accept it.

Minimum High-Memory KiloBytes

Specifies the minimum amount of extended memory that must be present to allow execution to proceed (Extended MegaBasic only). Execution aborts if less than the specified number of Kilobytes of extended memory is available. Non-zero values less than 128 are treated as 128, but you can specify any value higher than that. This minimum is useful to guarantee that sufficient memory exists prior to running any particular program. A default of 128k is the minimum setting for the version on the release diskette.

By setting the minimum to zero, Extended MegaBasic will run on machines that have no extended memory. However, this mode of operation is intended only for testing extended MegaBasic on such machines, rather than for general program use. This is because, under machines without extended memory, standard MegaBasic has fewer limitations and provides more available memory than extended MegaBasic, making it a better choice.

Maximum High-Memory KiloBytes

This Extended MegaBasic option specifies the overall limit on how much extended memory MegaBasic will ever use, even if more exists. This option is specified as the number of Kilobytes (units of 1024 bytes) and currently defaults to 4096K (i.e., 4 megabytes) on the release diskette.

Default Program File Extension Name

The default extension name to MegBasic program file names is **.PGM** whenever a program file name without any extension is specified. You can change this default with this **CONFIG** option to any upper case extension of 1, 2 or 3 characters. Do not specify the dot (.) in front of the new extension name.

Automatic Package Access

MegaBasic can be **CONFIGured** so that one or two MegaBasic packages are automatically **ACCESSed** by each and every program and package executed under that copy of MegaBasic. Either package can be an interpreter or assembler package. If an error occurs when MegaBasic attempts to internally **ACCESS** an automatic package, no error will be reported, the automatic package will not be **ACCESSed** and no further attempts will be made. Specify one package name or two package names separated by a comma and no spaces.

Section 4: Screen Flipping for Debugging

Debugging screen-intensive MegaBasic programs is often complicated by the fact that the output from the application shared the same screen as output from the testing and debugging dialogue. Application screens would invariably become corrupted by any debugging dialogue that took place, causing the screen output to become so contaminated that the very bugs being sought would be even more difficult to locate.

To remedy this difficulty, MegaBasic supports *screen-flipping*, a technique that separates application output from the MegaBasic command-level dialogue. This technique relies on the multiple video pages in **IBM**-compatible video systems running in text mode. When an application is running, the standard video page 0 is selected, which is the page normally selected for most **DOS** applications. When MegaBasic enters its command-level (i.e., at the *Ready* prompt), it selects video page 1 so that all output generated while *in* the command level will be displayed on video page 1 (without *contaminating* the application output still stored in video page 0). As soon as the application begins or continues execution (e.g., via **RUN** or **CONT**), the application output is restored as if no command dialogue had taken place.

Regardless of which video screen you are viewing, you can always see the other screen by pressing *both shift* keys simultaneously. This screen remains selected until any key is pressed, as which point the screen reverts back to the video page shown before. If the application was running when you pressed *shift-shift*, it is temporarily suspended (i.e., paused) for the period that the other screen is being viewed and resumes execution as soon as the screen reverts back. In order to accommodate systems that do not support the *shift-shift* combination (due to inadequate **ROM BIOS** compatibility), you can also use *alt-alt* or *ctrl-ctrl* on keyboards that provide such *dual-shift* keys.

All development activities use video page 0, including **TRACE** operations and output generated by direct statements. You do have to be careful with direct statements that invoke program procedures and function involved in changing the display, because their output will go to **video page 1** (which may not be what you had intended).

Screen-flipping is an optional feature that you enable by running the **FLIP** utility. **FLIP** *iS* a terminate-and-stay-resident (**TSR**) utility that provides the screen-flipping mechanisms through INT 2Fh, code BAh. All you have to do is include **FLIP** in your **AUTOEXEC.BAT** file so that it is present *before* you run MegaBasic. You *cannot install TSRs from MegaBasic:* they *must be installed before* MegaBasic is entered and **AUTOEXEC.BAT** is the best place to install them from. When **FLIP** is executed, it displays a message that says *MegaBasic screen flipping is ON*. If you run **FLIP** again, it toggles the screen-flipping and again displays a message informing you of its current state (i.e., **ON** or **OFF**). When **OFF**, MegaBasic uses **video page 0** for all operations, and behaves just like prior versions of MegaBasic. Running **FLIP** therefore lets you turn screen-flipping ON or **OFF** at any time. The **RUN** version of MegaBasic does not use screen-flipping regardless of whether or not screen-flipping is enabled.

Interrupt Trapping

Once **FLIP** is installed, it monitors a number of interrupt vectors that are never called, except when the system crashes due to a programming error. For example, INT 6 is called when the **CPU** attempts to execute an invalid instruction. Normally, an INT 6 will *crash* the machine because neither **DOS** nor the **BIOS** ever bothered to trap this important interrupt vector and do something intelligent with it. **FLIP** on the other hand, traps INT 6, displays the **CPU** registers and stack contents and then terminates the offending program. The interrupts **FLIP** traps in this manner are: 0 (divide error), 1 (single-step), 3 (breakpoint), 4 (numeric overflow), 6 (invalid instruction), and 7 (coprocessor unavailable) and 13 (protection violation). Programs that have valid uses for these interrupt vectors would normally insert their own interrupt vectors, overriding the ones set by **FLIP**. The protection **FLIP** provides is present for all programs run, not just MegaBasic, and is included with **FLIP** only because of a design omission in **DOS** itself. **FLIP** consumes only a couple kilobytes of memory and is therefore economical to have installed all the time.

Exiting Crashed Applications

FLIP also provide a special hot-key you can use to abort any application stuck in an infinite loop that would otherwise require re-booting the machine to regain control. This key, Ctrl-alt-ESC, aborts the currently executing process and displays the location and register contents in effect when you aborted execution.

Note

Aborted application will not have had a change to perform any necessary termination clean-up, which can leave the system in an unstable state.

Section 5: Real-Time Event Processing Utilities

EVENTS is a removable **TSR** that lets MegaBasic applications respond to a wide variety of simultaneous external **EVENTS** on **MS-DOS** systems without using wasteful status polling mechanisms. This is accomplished by using the logical interrupt facilities of MegaBasic to *catch the events* as they happen and queue them for processing by the MegaBasic application. The events currently supported are:

- Mouse events (buttons up / down / click, moved / stopped, etc.)
- Up to 10 independent timers
- Up to *40 hot keys*
- Any Shift / Ctrl / Alt / Caps / Insert key combinations
- Ctrl-Break

Serial port events are not provided by **EVENTS**, because the **MegaCOMM.sys** driver (available separately) is already designed to provide logical interrupt support for all aspects of interrupt driven serial communication (e.g., buffer getting full, buffer empty, special character received, error conditions, line control, baud rates, etc.).

The basic operation of **EVENTS** is very simple. After **EVENTS.COM** is installed as a **TSR**, you (i.e., the application) tell **EVENTS** to signal the running MegaBasic application when some specific event occurs and to identify this event with an *event id* (a 16-bit integer of your choice). Then the application continues about its business without any more attention on the pending event. When it occurs, the application automatically suspends its whatever it was doing, processes the event in the manner prescribed for its *event id,* and resumes the application where it left off (and without any additional polling or other programming at the application level).

Installing EVENTS.COM

EVENTS must be installed by running it *prior to executing* MegaBasic and any applications that use it. Once installed, **EVENTS** provides a set of calls you can make through **INTerrupt OBFh** that make requests for events to be triggered. Its various functions are selected by setting AH to a function number along with any other register values appropriate to that function. All functions are described later in this document.

You can remove the **EVENTSTSR** from memory by running it again with the /U command line option. **EVENTS** will not be able to remove itself from memory if another **TSR** has hooked the interrupt vectors used by **EVENTS,** and will display a message in such a case. If the *uninstall is* successful, the original interrupt vectors that were hooked by **EVENTS** will be restored and all memory used by **EVENTS** will be released back to **DOS** (only about 2300 bytes). **EVENTS** currently hooks following software interrupt vectors:

- Interrupt 09H for all keyboard shift key and hot key events
- Interrupt lBh for Ctrl-Break events
- Interrupt lCH for the timer events
- Interrupt 17h for printer events

Once installed, EVENTS provides a variety of functions to enable and disable the various events supported. These functions are called through **INTerrupt** OBFh, with a function number in register AH and additional parameters in other registers as needed. Since the event facilities are extended from time to time, the function details are documented in **EVENTS.DOC**, along with the error codes, recent changes and other information.

The Task Manager

A demonstration of **EVENTS** named **TASKMGR.PGM** is included. This is a skeletal task manager suitable as a starting point for building your own event-driven applications. It displays the activity of 10 independent timers, mouse events, keyboard shift key changes and function hot-key activity.

TASKMGR is completely driven by the external events provided by the **EVENTSTR** utility, i.e., it does not perform any device polling. This is accomplished by using the MegaBasic logical interrupt system to trap events and queue them for processing by the application in a controlled and organized manner.

Timer Facilities

EVENTS supports 10 timers driven by a single logical interrupt. Times are measured in milliseconds (resolved to **IBM-PC** timer-ticks of about 54 msecs). You can specify a 32 bit unsigned value for the number of milliseconds on a timer resolved up to the next tick (up to a maximum of about 3 weeks). When the timer times out, a MegaBasic logical interrupt is invoked, and the timer stops counting. Several functions are provided to: set timer values, set timer logical interrupts, pause / resume a timer, get timer status.

Keyboard Shift and Hot Key Events

You can request that **EVENTS** invoke a specified logical interrupt whenever a specified combination of keyboard shift keys is pressed down (Ctrl, Alt, Left Shift, Right Shift, Caps, Insert, etc.).

Another set of functions in **EVENTS** allows you to specify up to 40 hot keys. A *hot key* definition is comprised of 2 arguments: The scan code and **the shift** key status. When a user presses a key combination that has been defined as a *hot key combination*, a specified MegaBasic keyboard intercept procedure is invoked using a logical interrupt.

When a keyboard intercept occurs, the keyboard intercept mechanism is disabled until re-enabled by your program. Keys that are intercepted go no further than your program (i.e., to the **BIOS**). The keyboard intercept procedure can determine which Hot Key was invoked by examining the value in **INTERRUPT(2)** which contains the *event id you* assigned for that hot-key when you requested it from **EVENTS** (function 11).

Mouse Events

The *mouse is* a rich source of external events which can really consume a lot of processing power using a polled mechanism, but can be very efficiently handled using the event-driven approach provided by logical interrupts. **EVENTS** assumes a standard mouse driver is already installed supporting an **INT 33h** entry point into the mouse functions. One of the **EVENTS** functions enables and disables the mouse events. Enabling the mouse events tells the physical mouse driver to inform the **EVENTSTSR** about any change occurring on the mouse. You should disable the mouse events after your application has finished with using the mouse to clear the connection between the **EVENTS** and the Mouse driver.

Left and right mouse-button events are independently set for each button and can be any combination of: pressed, released, changed, clicked and double-clicked. A change event is the press or release of a button and is posted immediately; a *press* or *release event* is only posted if not part of a click; a *click event* is only reported when not part of a *double click*. Therefore, except for a *change event*, mouse button events are not reported until the button has remained unchanged for about 90 milliseconds.

The *mouse-position* events currently supported are *mouse-moved* and *mouse-stopped;* movement is posted immediately; stoppage is reported only after movement has ceased for at least 90 milliseconds. You can also define a region on the screen where the mouse must be in order for the event to be triggered.

All mouse events return a pointer (i.e., ES:BX returned to the logical interrupt handler) to a data structure containing 4 words:

Word 0	Event ID assigned by user (at the time event is armed)
Word 1	Current button status at the time of the event with bits positionally defined as (0,1) 1 (left,right)
Word 2	Mouse column position
Word 3	Mouse row position

Once an event occurs, you must re-arm the trap to activate the event again. Click-events take precedence over pressed / released events. You can call the various mouse driver functions directly (i.e., through Int 33h) for additional information about the mouse or to control it in other ways.

Section 6: Other Supplemental Packages

The program files described below are provided with the MegaBasic software system. They can be **ACCESSed** from your application and freely used and/or given away in any manner you wish.

LIBRARY.pgm

The **LIBRARY.PGM** contains a variety of procedures and functions that you may find useful in the development of your own programs. These modules also demonstrate the use of many features and serve as additional examples to supplement the manual. At the present time, all documentation for these routines is limited to the remarks in the **LIBRARYPGM** program file.

PCBASLIB.pgm

The **PCBASLIB.PGM** contains a variety of procedures and functions that you may find useful in the development of programs intended to run on the IBM-PC family of microcomputers. Many subroutines have been added to make it easy to access the various capabilities of the screen and keyboard, in particular. At the present time, all documentation for these routines is limited to the remarks in the **PCBASLIB.PGM** program file.

Section 7: MegaBasic Products

This section covers the MegaBasic products currently available. For additional information, pricing and ordering, and additional support, please contact your MegaBasic representative, the MegaBasic Bulletin Board System (415-459-0896) or MegaBasic Language Products at PO. Box 723, Fairfax CA 94978.

Development vs. Run-time Versions

The MegaBasic interpreter comes in two separate configurations: the full, **development** version (named **BASIC**), and its **run-time** subset (named **RUN**). They may be purchased together or separately. Each includes 14-digit **BCD** and **IEEE** binary floating point support.

MegaBasic Development Versions

The **development** version is the primary development tool that supports all phases of program development including program entry, editing, saving to files, debugging, testing, etc. All MegaBasic software development is performed in this configuration. When you run **BASIC**, YOU can either run an existing program directly or enter the program development environment.

The licensing for **BASIC** permits use on only one machine at a time. Distributing **BASIC** with MegaBasic applications is not permitted. However, you can economically extend your own license to additional machines by purchasing an additional MegaBasic Reference Manual for each machine.

BASIC includes a 600 page manual, various utilities, numerous programming examples and useful subroutines for your applications. **BASIC** requires at least 128k bytes.

MegaBasic Run-time Systems

The **run-time** MegaBasic subset, named **RUN**, executes MegaBasic applications, **without** program development capabilities. This allows special optimizations for running programs in the smallest space and shortest time. **RUN** is about 30% smaller than **BASIC** (saving about 24k bytes) and up to 50% faster. **RUN** transparently (and in negligible time) compresses each application code module as it loads into memory, reducing its load-image size by up to 60%.

RUN licenses you to bundle it with your MegaBasic applications for third-parties without additional fees or other restrictions. **RUN** includes a utility to link it together with your application modules to build a stand-alone program file, and another utility to cipher executable code into a form that cannot be converted back into readable source code. **RUN** requires at least 96k

MegaBasic Compiler System

This product boosts the execution speed of MegaBasic applications by a factor of 2 *to* 12 *times,* depending on the MegaBasic interpreter you are comparing it to and the kind of processing being done. Typical speed improvement is 5 *times faster* than the interpretive **run-time** system, with memory requirements for compiled applications roughly equal.

The MegaBasic compiler is also useful for **completely** analyzing and verifying the syntactic correctness of existing MegaBasic programs, often finding errors in code you may have thought correct.

Compilations are command-line oriented, with parameters specified in the command line, in response files, or both. It compiles at a rate *exceeding 1000 statements per second,* even on s-l-o-w computers. It reports all source code errors and supports all MegaBasic constructs.

The compiler supports various compile-time options to control optimization, line number inclusion, symbolic map file generation, error messages, package dependencies, output file format, destination path for result files, etc. Complex multiple package applications may be compiled through a response file, so that you only have to specify its compilation details once. The components of this product include:

- A compiler with libraries for 8 and 14 digit **BCD** and 16 digit 80x87 **IEEE** binary floating point. Other libraries supporting protected mode operation giving access to extended memory come with the **Extended MegaBasic** product**.**
- A utility automating the compilation of large applications.
- A utility combining multiple packages into a single **.EXE** file.
- A utility generating extensive cross-reference listings for documenting large, multiple package applications.
- A compiler manual supplementing the MegaBasic Language Reference and Programmer's Guide.

The MegaBasic compiler is used in conjunction with the MegaBasic Development System running under **MS-DOS** or compatible operating systems (e.g., Concurrent **DOS**).

Extended MegaBasic

Extended MegaBasic runs completely in *protected-mode* in order to provide full access to as much as 16 megabytes of *extended memory,* otherwise it is indistinguishable from standard MegaBasic.

A **DOS**-extender is not needed with this special version of MegaBasic. Extended MegaBasic is a **DOS**-extender, dedicated to developing and running **very large** MegaBasic applications with **lots** of data. It is the first, and possibly the only, **BASIC** that runs in protected-mode on **MS-DOS**. It provides the following features:

- It executes MegaBasic programs under **MS-DOS** with access to up to 16 megabytes of memory, providing an ideal environment to develop extremely large applications in **BASIC** without paying workstation or mainframe prices.
- String and numeric arrays may be up to 16 megabytes. The total number of arrays and strings is limited only by the amount of available memory.
- 100k *more memory is* provided in the 640k base memory region for **DOS** shell commands executed under extended MegaBasic than under standard MegaBasic. This is because all but 8k of code and 16-128k of its data resides in extended memory.
- Nearly all existing MegaBasic programs execute perfectly without any modifications. Some programs that involve themselves in direct machine access (using **FILL**, **EXAM** and **CALL** statements) may require some very minor changes.

- Special installation procedures, **DOS**-extenders and memory drivers are **not required**, you simply run your program as a normal MegaBasic program.
- Extended MegaBasic only uses the amount of memory left unused by other processes that may also be using extended memory (e.g., **VDISK** drivers and operating system caches). You can set Extended MegaBasic to limit the amount of extended memory available even when more is present.
- It operates in the **protected mode** of the 80x86, but allows user programs to call operating system functions residing in real mode (e.g., **DOS**, **ROM BIOS**, etc.).
- All MegaBasic programming constructs are supported, including **FILLS**, **EXAMS**, **CALLS**, logical interrupts and assembler packages.
- The execution speed in protected mode is roughly the same as in real mode, with only about 3 to 10 percent lost, due to inherent differences in protected mode operation.

Extended vs. Standard MegaBasic

Apart from the vast memory resources available, virtually all MegaBasic constructs in Extended MegaBasic operate the same way as they do under standard MegaBasic. The few features that differ are all a result of the fact that extended MegaBasic executes in the *protected-mode* of the 80286 / 386 / 486 microprocessor, while standard MegaBasic operates in *real-mode*. These features are summarized below:

- Direct access to machine memory using **EXAM** and **FILL** statements is different because the segment portion of the address is no longer a *paragraph* address, but rather a segment *selector*. Extended MegaBasic provides a method to convert paragraph addresses into selectors for use in **EXAMS** and **FILLS**.
- Machine language interrupt calls using **CALL#** statements are supported in a virtually 100% compatible manner. This allows you to make interrupt calls to *real-mode* interrupt routines, such as **DOS** calls, **ROM BIOS** interrupts, etc.
- Machine language *subroutine far-calls* using **CALL** statements are supported by *extended* MegaBasic, but only to call protected mode routines using known code selectors.
- **CALLing** machine code that has been loaded into a MegaBasic string variable is not supported.
- Background processing using **INTERRUPT** statements is not supported, but logical interrupt driven real-time event processing is supported.
- **BASIC** Terminate-and-Stay-Resident processes (**TSRS**) implemented with SERVICE statements are not supported.

System Requirements

Extended MegaBasic requires a certain minimum hardware configuration to operate. This minimum is generally satisfied by PC-AT or PS-2 class machines using 80286, 80386 or 80486 microprocessors. Specifically, the requirements are:

- An 80286, 80386 or 80486 microprocessor. Systems that routinely process over 1000 interrupts per second may require an 80386 / 80486, due to the limitations of the 80286 microprocessor.
- At least 128 *kbytes* of available memory in both the base megabyte and the extended memory regions. *Expanded memory is* neither used nor affected.

- **DOS** version 3.0 to 5.x running in *real-mode*, or in the Virtual-8086 mode of the 80386/486 if *VCPI/DPMI is* supported (as provided by most 386 memory managers such as *QEMM, 386MAX* and *Microsoft Windows).*
- A standard **IBM**-compatible **ROM BIOS** that includes support for extended memory. Any system that supports extended memory **VDISK** drivers will support protected mode Extended MegaBasic.
- Assembler packages are supported under Extended MegaBasic, but there some "rules of the road" that must be obeyed by the assembler code when running in protected mode.

The ***Extended MegaBasic System*** includes both the development and runtime systems, support for both decimal **BCD** and **IEEE** 80x87 binary floating point, extended memory versions of the compiler run-time libraries (requires MegaBasic compiler), plus all utilities, full documentation and no-nonsense unlimited license to bundle the run-time system with your applications distributed to third-parties.

VSCREEN Window Manager

The **VSCREEN** package provides sophisticated windowing, field management and menu support for MegaBasic applications. For compatibility and very low-overhead, the **VSCREEN** software operates only in text-mode and includes the following capabilities:

- Windows are view-ports through which you view all or part of an arbitrarily large **virtual screen** behind it. The virtual screen portion seen through the window can be instantly repositioned or resized.
- Window style and layout includes border type, shadow and color, foreground and background colors for output and for input fields, window size, placement and visibility mode, virtual screen size, and cursor size and visibility.
- The number of active windows is unlimited and you can control how windows overlap. Deleting a window releases all its memory for use elsewhere in your program.
- Windows can appear and disappear instantly or gradually (explode, implode, digital fade, etc.). They can be moved about the screen instantly, or slid across the screen under program or mouse control.
- Ordinary MegaBasic **PRINT** statements are used to send output to **VSCREEN** windows. Field-input editing and menu selection routines are used for keyboard input and mouse control.
- **ANSI** escape sequences to windows are supported. Cursor locations relate to the virtual screen. Character output and other operations are supported on hidden, visible or partially overlapped windows.
- **VSCREEN** includes other packages that provide high-level functions, such as screen-forms and field management, menu-bars with drop-down menus, file selection, directory browsing, multiple text-file browsing, and more.

VSCREEN supports monochrome, Hercules, **CGA**, **EGA** and **VGA** under **MS-DOS** on any **IBM**-PC compatible machine. It includes 6 major packages, extensive demos illustrating all features and capabilities, full source code, 80 page manual and license to include execute-only components in your MegaBasic applications supplied to third-parties.

VTRIEVE Record Manager

VTRIEVE removes the burden of developing data management facilities, one of the more tedious and complex chores in applications programming. This lets you concentrate on the design and development of your application instead.

VTRIEVE is a powerful record management engine for MegaBasic programmers, containing all the necessary functions and procedures to efficiently manage an application's data files. Programmers **familiar with Novell's BTRIEVE** record manager will find that **VTRIEVE** supports most of its functions plus a few more, including shared access to **VTRIEVE** files over networks, with better performance.

Access to data records in **VTRIEVE** is extremely fast, no matter how many records are in the file. This is because it is based on **B-tree file structures,** known for their speed and reliability. **VTRIEVE** can store data records **and** their indexes within a **single file**, reducing the number of files that you must physically open for large applications. This lets you have over 100 different databases open through **VTRIEVE simultaneously.**

It is easy to create and destroy indexes on the fly (up to 40 per file), which are then maintained automatically as records are added, updated and deleted. There is virtually no ceiling on the number of records **VTRIEVE** can maintain within each file. **VTRIEVE** supports fixed length and variable length data records (up to 65520 bytes) and duplicate record keys are permitted. Each **VTRIEVE** file can include arbitrary **application-specific** information, useful for field descriptions, record counters, configuration data, display control, etc.

Contents and Requirements

The **VTRIEVE** product includes the **VTRIEVE** module, a manual explaining all its functions and procedures, a description of the internal layouts of **VTRIEVE** files, and a small demo application. **VTRIEVE** is written entirely in MegaBasic and includes full source code with liberal commentary.

VTRIEVE requires MegaBasic interpreter 5.60 or later running under **MS-DOS** 3.0 or later (optionally MegaBasic Compiler 1.62 or later). Execute-only **VTRIEVE** components may be included with MegaBasic applications distributed to end-users.

MegaCOMM Serial Device Driver

MegaCOMM is an attachable **MS-DOS** serial device driver that supports up to 8 serial devices, any or all of which can be interrupt driven. It provides special event-driven support for use with MegaBasic applications.

A single **MegaCOMM** driver supports any set of 8 serial devices, **COM1** through **COM8**, while keeping driver size to a minimum (i.e., under 6k plus buffers). Interrupt driven operation, on input, output or both, is supported independently on each device. **MegaCOMM** lets you assign multiple interrupting devices to the same interrupt vector as needed (or to multiple interrupt vectors).

A utility is provided that allows static configuration of serial device operation, by modifying the device driver file itself. An extensive **IOCTL** language lets you dynamically control device operating characteristics and request status information, including:

- Device baud rate, stop bits, byte width and parity.
- **RTS/CTS** and **DTR/DSR** line control and forcing breaks.

- **XON/XOFF** flow control usage and character data masks.
- Device status: input available, output ready to send, carrier or **CTS**, output device still sending, **DTR** and **CTS**, break detect, framing error, parity and overrun.
- Operational status: input buffer count and availability, output data remaining and available, and detailed device configuration parameter and status set.
- A request for **input event traps** to generate a MegaBasic logical interrupt when an input character count is reached, a specified character is input or a time-out expires.
- A request for **output event traps** to generate a MegaBasic logical interrupt when a specified amount of output buffer space becomes available or a time-out expires.

The **MegaCOMM** product includes: the device driver, documentation, configuration utility, **IOCTL** demo / utilities, and a MegaBasic demo / utility package providing modem connection facilities for MegaBasic applications, and license to distribute execute-only components, when bundled with applications.

AnsiPLUS Enhanced Console Device Driver

AnsiPLUS is the ideal console driver for power-users. The standard device drivers supplied with **MS-DOS**, the critical interface between you and your personal computer, do not take full advantage of the capabilities of the major video controllers currently in use: **EGA** and **VGA**. In addition, the standard personal computer **BIOS** has severe limitations.

AnsiPLUS integrates major console elements missing from **MS-DOS** and **PC-BIOS** into a single compact device driver that you can easily control and personalize. With **AnsiPLUS** installed, your interaction with **MS-DOS** and non-Windows applications will be significantly improved.

There are distinct advantages to using **AnsiPLUS** with **MS-DOS** programs in Windows 3.0 Enhanced Mode. For example in this mode, separate copies of **AnsiPLUS** are automatically included in each **MS-DOS** process.

Extended EGA and VGA Features

The **AnsiPLUS** enhanced **MS-DOS** console device driver supports the following **EGA/VGA** display features, not provided by other screen drivers:

- Automatic recognition of all text and graphics screen modes, including those with large numbers of lines and columns.
- Much faster operation in text and 16-color graphics modes, than either the original **MS-DOS** console driver or the standard **ANSI** driver.
- Key-enabled smooth scrolling so you can effortlessly read the screen while it is scrolling (only with faster **CPUS**).
- Full control over colors, supporting 16 foreground / background colors in color text modes and color-mixing in 16-color graphics modes. Control over palette registers and **VGA DAC** registers let you tap the full color possibilities of your video board.
- Selection of **EGA/VGA** character sets, to control the character height used for any video mode. Simple **ANSI** control sequences to switch between 25 / 43 / 50 line text modes.
- An extended **ANSI** Set Mode control sequence to specify modes based on expanded **CPU** registers.

Replacements for Common TSRs

The **AnsiPLUS** driver effectively integrates several console functions eliminating the need for separate **TSR** programs or device drivers. These functions include:

- An intelligent screen saver.
- The ability to freeze scrolling and then scroll measured amounts of text in various useful ways.
- Extending keyboard input type-ahead up to a maximum of 192 keystrokes.
- Preventing your type-ahead buffer from filling with unintended keystrokes by disabling **repeat key** type-ahead, while allowing repeat keys that are immediately consumed by programs.
- An interrupt-controlled Ctrl-G tone generator that prevents multiple beeps from "stacking up" and stalling your programs, and a special **ANSI** control sequence to set the frequency(s) and duration(s) of the Ctrl-G beep tone.

OtherAnsiPLUS Extensions

- Ability to let you *scroll back* many screen-fulls of lines that have recently scrolled off the top of the screen out of view.
- **AnsiPLUS** highlights keys you type as they are displayed on the screen, visually separating typed entries from computer output.
- Unlike standard drivers, **AnsiPLUS** displays a cursor for typed entries while in graphics mode.
- **AnsiPLUS** provides functions to insert and delete lines and characters.
- **AnsiPLUS** has a "transparent background mode" that writes each output character in the current foreground color without changing the background color.
- You can apply bold, underline, black shadow, and slant enhancements, in any combination to output characters in 16-color graphics modes.
- **AnsiPLUS** allows blanks between parameters, commas and semicolons as separators, single or double quotes around character parameters, and signed parameters up to 32767.

MegaBasic Support Features in AnsiPLUS

AnsiPLUS comes with an extensive MegaBasic application support package that provides a programming front-end to all features of **AnsiPLUS**. It allows you to:

- Set video and graphics modes, select character set height text treatment in graphics modes, select video page, set or modify cursor position, save / restore complete video state.
- Define available colors, set **EGA/VGA** palette registers, select current color attributes for output characters, enable / disable transparent background and blinking colors.
- Clear the current screen page or current line, insert and delete lines and characters on the screen.
- Fill rectangular areas on the screen, with or without borders, shadows and titles. Scroll a rectangular area in any direction, save and restore a rectangular area between the screen and a MegaBasic string variable.
- Query **AnsiPLUS** status for cursor position, colors, etc.
- Enable and disable various **AnsiPLUS** driver features, define Ctrl-G beep tones and key translations.

AnsiPLUS comes complete with the **AnsiPLUS** console device driver, detailed user documentation, and utilities for controlling colors and other **AnsiPLUS** features. It also includes create and edit escape sequence programs, the MegaBasic **AnsiPLUS** application support package, and license to distribute necessary execute-only components of **AnsiPLUS** with MegaBasic applications.

Appendix D Miscellaneous Information

This appendix covers various subjects that are less likely to be needed on a regular basis, but are nonetheless of vital importance at certain times. The following areas are covered:

Appendix D Section.1	**MegaBasic Enhancements**	Brief summary of all changes to MegaBasic made since the previous MegaBasic manual (April 1989)
Appendix D Section 2	**MegaBasic Reserved Words and Characters**	Alphabetical listing of all words and special characters reserved by MegaBasic for use in commands, statements, operators functions and other operations.
Appendix D Section 3	**Code Conversion Tables**	Reference table for conversion between decimal, hexa-decimal, binary and ASCII codes.
Appendix D Section 4	**Converting Non-integer Programs to Use Integers**	Step-by-step instructions on how to take advantage of integers in programs originally designed for floating point.
Appendix D Section 5	**Loading Earlier Programs**	Explanation of the built-in automatic translator that converts programs written prior to MegaBasic Version 4.0, including the loading and running of North Star Basic programs.

Section 1: Recent MegaBasic Enhancements

This appendix summarizes the major changes to MegaBasic that have occurred since MegaBasic Reference Manual, Revision C (April 1989). It provides only very brief references to each item are given and you should seek the appropriate section in the manual for complete information. Users familiar with MegaBasic versions earlier than 5.65 can study this appendix for a quick review of recent extensions that may impact current software development efforts. With a couple minor exceptions, MegaBasic has been extended in an upward compatible manner to preserve the operating characteristics and executability of all programs developed under earlier versions of MegaBasic. Descriptions of minor improvements are excluded.

Changes Since Revision E (November 1991)

- The **IN** comparison operator has been extended to integer bitstring comparisons, in addition to strings.
- MegaBasic under **MS-DOS** now requires **DOS** release 3.0 or later.
- In a multi-user or network environment, SAVE without arguments gives you a warning if someone else has modified the same MegaBasic program file since you last LOADed it.
- **LOAD** commands can now load more than one program file at a time.
- The **CONFIG** utility is now easier use, has more options and includes command line operation (i.e., non-interactive batch mode).
- In **DOS** versions, **WAIT** releases current virtual time-slice for multitasking systems. **ELAPSE()** and **WAIT** are now accurate to within 1 millisecond.
- Arrays with 2 or more dimensions can now have more than 65535 elements.
- USE sequences to the next package name beginning with a character typed and no longer sequences through *free packages* to avoid clutter.
- The new **MBPATH=** environment variable lets you override the **PATH=** program directory search order.
- **ENVIR$(0)** now returns the complete pathname of the main program file.
- **END** now executes any remaining **EPILOGUES** in first-to-last order
- **PRINT** formats can now alter the characters used for ,.$ in formatted numbers.
- **OPENing** files in *exclusive read/write* mode are now allowed to be opened *read-only by* others in a network environment.
- **OPENC INPUT** statements will now create input files if they do not already exist.
- The **FILE()** function returns 0 for *no file,* 1 for *read/write* files and 2 for *read-only* files.
- Setting **PARAM**(22)=-1 enables network use *without* enabling the automatic file locking features of MegaBasic.
- **PARAM**(23) now returns 4 for 80486 CPUs and 3 for 80386 CPUs.
- **FLIP** now locks the temporary screen context until the next keypress, instead of requiring you to hold down both shift keys.

- **SHOW SIZE** now displays the size of field structures.
- Total symbol capacity over all packages has been increased to 8190, up from about 6000 to 7000.

Changes Since Revision D (March 1991)

- Extended MegaBasic is now compatible with **VCPI**-compliant 80386 memory managers, such as **QEMM** and **386MAX**. Symbol capacity has been increased by about 15%. String arrays may now be up to 16 megabytes in size. Memory segment capacity has increased to 400 segments.
- During any MegaBasic command that allows you to pause screen output with the space bar, you can now immediately type in the next command while you are in pause mode without having to first abort the command. The **TAB** key can now be typed during the display pause to display the next dozen lines.
- **OPEN READ** statements now open the file in a mode that allows other external processes to also have the file open for either reading or writing. No locking of any kind is imposed and the file operations are buffered for fast accessed.
- **SHOW** , **SHOW ACCESS** and **USE** commands display the package names in alphabetically sorted order. The **SHOW** command display is enhanced with additional information. **SHOW ACCESS** commands now show only the **ACCESS** relationships of the current package and to show the relationships between all packages—you have to append an asterisk (*) to the end of the **SHOW ACCESS** command.
- **USE** commands now supports the Up, Down, Left, Right, Home, End and Tab keys to walk around in the package list.
- **FREE(3)** returns the number of unused bytes remaining in the global symbol table, which manages all the symbols over all packages in a running MegaBasic program. **FREE(4)** returns the number of unused memory segments.
- Logical interrupts are now serviced during **INPUT**, **INCHRS** and **WAIT**. Individual logical interrupts can now handle multiple interrupts without necessarily causing an overrun condition. This is done by buffering the external interrupt post requests up to a maximum count, specified by an optional 4th parameter on the **INTERRUPT** statement.
- **XREF** lets you display the locations of references to lines and names *by procedure or function name* as well as by line number. In other words, instead of showing a list of line numbers that contain references to each name, **XREF** can show you a list of *subroutine names* that contain references to each name.
- Displays showing program locations by line number now include the name of the subroutine that contains that line. This is particularly useful in the **TRACE RET** command, as it makes the current calling path plainly visible.
- The line range of a subroutine (i.e., **FUNC or PROC**) can now be specified by name in any command that expects a line range. The name must be preceded by a dot (.) to indicate this is a line range, rather than some other command argument.
- The starting line and step-size arguments now default to the previous values left over from the previous **ENTER** command (in that workspace), or to 10 on the first **ENTER** command.

- Ctrl-**HOME** deletes all characters to the left of the cursor.
- PgUp / PgDnkeys which access the recent input entries now searches for entries that match the characters appearing to the left of the cursor.
- You can now enter extended ASCII codes (codes 128 to 255) into an input line using the ALT-000 mechanism provided by more recent keyboard drivers.
- The child-process *exit code* left after a **DOS** statement executes a shell command is now available as a system error code from **PARAM**(l9).
- **ACCESS** and **DISMISS** functions added to provide greater control over systems of packages.
- **FIND**(**VEC** *vector exprn)* added to let you search vectors.
- The error code for all non-trappable errors has been changed from O to 255 and any user-defined error with that code will be non-trappable.
- **INPUTS** into string fields or indexed (i.e., fixed-width) string variables now perform right-fill as done by string **==** assignments. **INPUT2** now displays prompts instead of suppressing them.

Changes Since Revision C (April 1989)

- **MERGE** and **DEL** *statements* are no longer supported in MegaBasic (the **MERGE** and **DEL** *commands* are, of course, still supported).
- The **MERGE** *command* (Chapter 2, Section 3) now supports *multiple line ranges,* allowing you to restrict the merge to only those source program lines. These ranges can be specified either by line *number* or by subroutine *name.*
- Seven new operators provide support for *bit-wise* logical combinations, shifting and rotating in both directions of 32-bit integers.
- **STRUCT** statement extended in various ways, the most important being multi-line **STRUCT** and **DEFSTRUCT** statements for static **STRUCT** definitions. The **STRUCT** reserved word may be used in **XREF** and NAMES commands to select field variables in the command report.
- **DEF** DATA statements now longer need the DATA reserved word when defining variables data types or defining SHARED variables.
- **DEF** statements can be entered as direct statements to affect subsequent type assumptions (until the next **RUN** command).
- **DEF** statements to define *pointer types* have been added to assist the MegaBasic compiler's treatment of pointers. Such statements are *ignored* by version 5.600 and later of the MegaBasic interpreter, but reported as *Syntax Errors* in earlier versions.
- The $ *format-modifier* appends the radix letter (i.e., *b, o* or h) to numbers printed in binary, octal and hexadecimal formats.
- The **DESTROY** statement now *ignores* file names that already do not exist, instead of reporting a *File not found error.*
- The new **ENTER** statement lets you temporarily *redirect console input* from a file or other device.
- MegaBasic now gives you the opportunity to finish any remaining *epilogues* prior to processing any **LOAD**, **RUN**, **CLEAR** or **BYE** commands you enter while an initialized program is active.

- When **LOADing** a text-file program, MegaBasic no longer renames the loaded program with a *.pgm* file extension as it used to, so be careful to rename it yourself if you later **SAVE** the program.
- File locking is now supported and enabled in all **MS-DOS** versions of MegaBasic. **PARAM**(22) can be set non-zero or zero to enable or disable file locking.
- You can no longer **SAVE**, *by name*, the contents of a workspace other than the current work space. However, **SAVE** now lets you save multiple workspaces with one **SAVE** command.
- **PARAM**(8) can be set non-zero to disable all further *epilogue* execution, which can be useful in fatal error termination and cleanup processing.
- The new **FLIP** utility (a **TSR** you run before running **BASIC**) improves the MegaBasic program development and debugging environment by separating the program *application output screen* from the program *debugging and editing screen* so that they do not interfere with one another.
- **INTERRUPT**(2) returns additional information to logical interrupt driven background processes. Other background process enhancements have also been made.
- **DISMISS** no longer removes **unACCESSed** packages unless you specify their names in the **DISMISS** statement, i.e., you have to explicitly **DISMISS** packages in order to get their *epilogues* executed.
- The new **CREATE()** *function* lets you create new variables of any data type. Such variables, called *pseudo variables*, have no name and are accessed via the *pointer* returned by **CREATE**. The new FREE *statement* lets you release variables created by the **CREATE** function.
- **DIR$=***pathname* now changes the default drive if a different drive code is specified in the *pathname*.
- If memory permits, **LINK** statements no longer obliterate packages left in memory so that subsequent **ACCESSes** or **LINKS** can pick them up from memory instead of unnecessarily re-loading them from the disk.
- **PARAM**(25) can be set to 1 to cause subsequent **LINK** statements to preserve the execution state of the current packages (i.e., their program variables, open files and **ACCESSes**) through to the next program.
- The **SHOW** , **SHOW ACCESS** and **SHOW SIZE** commands now provide more information to assist in the testing of large applications with *many packages*.
- **ERRLINE(1)** returns the *relative statement number* on the line in which the most recent error occurred. **ERRLINE** and **ERRLINE(0)** both return the line number of the most recent error.
- *Ctrl-Break* now optionally replaces *Ctrl-C* for program interruption using an interrupt driven approach that preserves the t*ype- ahead* input buffer. See **PARAM**(1) for details.

Section 2: MegaBasic Reserved Words and Characters

The table below lists all the reserved words used by MegaBasic. None of these words are available for use as names of structure fields, variables functions, procedures or line-labels.

ABS	cont	errline	include	max$	rem	stop
access	copy	errmsg$	index	merge	ren	str$
acos	cos	errpkg$	inp	min	rename	string
and	create	errset	inp$	min$	repeat	struct
append	**DATA**	errtyp	input	mod	request	subdir$
argument	date$	exam	input$	move	reseq$	sum
asc	def	exit	input1	**NAME**	restore	swap
asin	del	exp	input2	names	ret	swapdef
atn	delete	**FILE**	int	next	retry	**TAB**
BASIC	destroy	filectrl	integer	nomark	return	tan
begin	dim	filedate$	interrupt	not	rev$	then
bit	dir	filepos	ioctl	**ON**	rnd	time$
by	dir$	filesize	ioctl$	open	rotat$	to
bye	dismiss	filetime$	**LEN**	open$	round	tr
CALL	div	fill	let	openc	run	trace
card	dos	find	li	or	**SAVE**	tran$
case	dupl	for	line	ord	scr	trim$
ceil	**ED**	frac	link	out	seg	trunc
ch	edit	free	list	output	service	typ
chain	edit$	from	ln	**PARAM**	sgn	**UNLOCK**
change	elapse	func	load	pi	shared	use
check	else	funcend	loadpkg	poly	show	**VAL**
chr$	end	**GOSUB**	local	pos	sin	vec
chrseq$	ent	goto	lock	print	size	**WAIT**
clear	enter	**IF**	log	proc	space	while
close	envir$	imp	loge	procend	sqrt	write
collat$	eqv	in	**MATCH**	**READ**	stat	**XOR**
const	errdev	inchr$	max	real	step	xref

The two names **PROLOGUE** and **EPILOGUE** are not really reserved words, but they do have special meaning in conjunction with packages and hence you should avoid using them for other purposes as well. From time to time, new built-in statements and functions may be added to the repertoire of MegaBasic. User-assigned names in older programs that match new reserved words must be renamed in order to use newer versions of MegaBasic. To this end, MegaBasic automatically renames such occurrences when a program is first loaded. This action, which takes place on a **LOAD**, **INCLUDE**, **ACCESS**, **MERGE** or **LINK** (in the

development version only, not in **RUN**), merely doubles the leading character of the name until it no longer matches any reserved word or user defined name. MegaBasic reports each name reassignment it makes to you on the console.

Special Characters

In addition to the various reserved words, MegaBasic also uses most of the symbols in the standard ASCII character set for specific purposes within program statements and commands. For example, letters (A-Z, a-z), digits (0-9) and underscores (_) are used to form identifiers; numeric constants are formed from the digits, letters A through F, plus (+), minus (–) and period (.). A complete summary of all special characters follows below:

Special Characters and Delimiters	
( )	*Parentheses:* surround function arguments, subexpression groupings, array subscripts and indexing expressions.
[ ]	*Brackets:* surround multi-statement **THEN/ELSE** clauses in IF statements, memory offsets of program variables, concatenated vector variables, structured variable definitions.
“ ”	*Quotes:* surround string constants.
’ ’	*Apostrophes:* surround string constants.
+	*Plus sign:* addition and string concatenation operator.
–	*Minus sign:* subtraction, negation and string difference operator, line range separator in commands, letter range separator in data type declaration **DEF** statements, search string option disable.
*****	*Asterisk:* multiplication operator, pointer variable resolution operator, vector wild-card selector, pointer argument lead-in in **PROC/FUNC** definitions, wild-card string match symbol in **LIST** , **EDIT** and **CHANGE** searches).
/	*Slash:* real division operator, separator in directory pathnames, newline generation in **PRINT** statements.
****	*Backslash:* statement separator, separator in directory pathnames.
^	*Caret:* exponentiation / poweroperator, pointer extraction operator, bit-wise *xor-operator (^^).*
_	*Underscore:* visual separator for use in long identifiers.
.	*Period:* decimal point in numeric constants, file name extension separator, structured variable pathname component separator, open-ended arguments in **FUNC/PROC** definitions, the current program line number or range in development commands, lead-in to symbolic line ranges.
,	*Comma:* separator between arguments, array subscripts and indexing expressions.
:	*Colon:* line-label separator, part of the string replacement assignment operator :=, separator indicating a width expression in indexing of strings, vectors and bit ranges.
;	*Semicolon:* separator between program statements.

Special Characters and Delimiters	
<	*Less-than:* string and numeric comparison operator, arithmetic shift operator (<<), lead-in for numeric scaling value in format specifications.
>	*Greater-than:* string and numeric comparison operator, arithmetic shift operator (>>), lead-in for numeric scaling value in format specifications.
=	*Equal sign:* assignment operator, comparison operator, default argument value lead-in in **FUNC/PROC** definitions lead-in for single-line **FUNCtion** expression definition.
#	*Number sign:* device or open file number lead-in, software **INTerrupt** number lead-in in **CALL** statements.
!	*Exclamation point:* real type indicator in identifiers, shorthand for **PRINT**.
$	*Dollar sign:* string type indicator in identifiers, symbol for line number of last program line.
%	*Percent sign:* integer type indicator in identifiers, file position in **READ/WRITE**, copy argument type lead-in in **FUNC/PROC** definitions.
&	*Ampersand:* binary mode file transfer in **READ/WRITE** statements, wild-card string match symbol in **LIST**, **EDIT** and **CHANGE** searches, bit-wise *and-operator* on integers.
|	*Vertical bar* bit-wise *OR-operator* on integers.
~	*Tilde:* bit-wise *ones-complement* unary operator on integers.
@	*At-sign:* 16-bit integer transfers in **READ/WRITE** statements, lead-in for by-reference or by-address arguments in **FUNC/PROC** definitions, lead-in for **STRUCT** position changes.
?	*Question mark:* wild-card character used in program editing searches.

Section 3: ASCII Character Codes and Special Keys

The table below lists all 256 byte values in decimal, hexadecimal, binary, and in ASCII. *IBM compatible keyboards* provide a number of *extended keys,* which produce a two-character sequence when typed. The first is always a null character (ASCII zero) that indicates a second *extended key-code* follows. The second code identifies the special key itself and these codes are shown in the table for your convenience. You can safely assume that if a null character is input from the console, then an *extended code* will follow for which your program should wait. The following IF statement illustrates a simple way to input a single *normal* or *extended* key.

```
IF (let C$ = inchr$(0)) = chr$(0) then C$ = C$+inchr$(0)
```

Be sure to disable the MegaBasic Ctrl-C mechanism (by setting PARAM(l) to 1) while performing *one-at-a-time* character input, so that none of the input characters are lost to the Ctrl-C detection mechanism.

Some of the extended codes are typed with the *Ctrl* or *Alt* keys held down and are indicated in the table by a ^{ctl} or ^{alt} superscript. Others are keys that have special names, such as *Ins* or *Del,* which are shown by name. Still others are called *function keys* and are indicated as *F1, F2, ..., F10.* While some computers may have additional extended keys beyond those shown, we have included only those that are in general use and available on the vast majority of *IBM compatible keyboards.*

The ASCII codes from 128 to 255 are called the *extended ASCII* codes (unrelated to the *extended keys* described above). Although they cannot normally be typed at the keyboard, more recent keyboard drivers often let you type any arbitrary ASCII code by typing its code in decimal digits *while holding the* **ALT***-key down*. For example, **ALT-234** generates an ASCII 234 code as soon as you release the **ALT** key. Be careful that you *do not* type extended ASCII characters into a program, except for characters inside quotes (i.e., within string literals), because they will conflict with the internal binary representation of the program line.

Dec	Hex	Binary	Key	Ext	Dec	Hex	Binary	Key	Ext
0	00h	0000000	@ ctl		32	20h	00100000	sp	
1	01h	00000001	A ctl		33	21h	00100001	!	
2	02h	00000010	B ctl		34	22h	00100010	*	
3	03h	00000011	C ctl		35	23h	00100011	#	
4	04h	00000100	D ctl		36	24h	00100100	$	
5	05h	00000101	E ctl		37	25h	00100101	%	
6	06h	00000110	F ctl		38	26h	00100110	&	
7	07h	00000111	G ctl		39	27h	00100111	’	
8	08h	00001000	bksp		40	28h	00101000	(	
9	09h	00001001	tab		41	29h	00101001	0	
10	0Ah	00001010	1feed		42	2Ah	00101010	*	
11	0Bh	00001011	K ctl		43	2Bh	00101011	+	
12	0Ch	00001100	L ctl		44	2Ch	00101100	,	
13	0Dh	00001101	ret		45	2Dh	00101101	–	
14	0Eh	00001110	N ctl		46	2Eh	00101110	.	
15	0Fh	00001111	O ctl		47	2Fh	00101111	/	
16	l0h	00010000	P ctl		48	30h	00110000	0	
17	11h	00010001	Q ctl		49	31h	00110001	1	
18	12h	00010010	R ctl		50	32h	00110010	2	
19	13h	00010011	S ctI		51	33h	00110011	3	
20	14h	00010100	T ctl		52	34h	00110100	4	
21	15h	00010101	U ctl		53	35h	00110101	5	
22	16h	00010110	V ctl		54	36h	00110110	6	
23	17h	00010111	W ctl		55	37h	00110111	7	
24	18h	00011000	X ctl		56	38h	00111000	8	
25	l9h	00011001	Y ctl		57	39h	00111001	9	
26	lAh	00011010	Z ctl		58	3Ah	00111010	:	
27	1BH	00011011	[ctl		59	3Bh	00111011	;	F1
28	1Ch	00011100	\ ctl		60	3Ch	00111100	<	F2
29	1Dh	00011101	] ctl		61	3Dh	00111101	=	F3
30	1Eh	00011110	^ ctl		62	3Eh	00111110	>	F4
31	1Fh	——11111	_ ctl		63	3Fh	00111111	?	F5

Dec	Hex	Binary	Key	Ext	Dec	Hex	Binary	Key	Ext
64	40h	01000000	@	*F6*	96	60h	01100000	’	*F3*^{ctl}
65	41h	01000001	A	*F7*	97	61h	01100001	a	*F4*^{ctl}
66	42h	01000010	B	*F8*	98	62h	01100010	b	*F5*^{ctl}
67	43h	01000011	C	*F9*	99	63h	01100011	c	*F6*^{ctl}
68	44h	01000100	D	*F10*	100	64h	01100100	d	*F7*^{ctl}
69	45h	01000101	E		101	65h	01100101	e	*F8*^{ctl}
70	46h	01000110	F		102	66h	01100110	f	*F9*^{ctl}
71	47h	01000111	G	*home*	103	67h	01100111	g	*F10*^{ctl}
72	48h	01001000	H	↑	104	68h	01101000	h	*Fl*^{alt}
73	49h	01001001	I	*PgUp*	105	69h	01101001	i	*F2*^{atl}
74	4Ah	01001010	J		106	6Ah	01101010	j	*F3*^{atl}
75	4Bh	01001011	K	←	107	6Bh	01101011	k	*F4*^{atl}
76	4Ch	01001100	L		108	6Ch	01101100	l	*F5*^{atl}
77	4Dh	01001101	M	→	109	6Dh	01101101	m	*F6*^{atl}
78	4Eh	01001110	N		110	6Eh	01101110	n	*F7*^{atl}
79	4Fh	01001111	O	*end*	111	6Fh	01101111	o	*F8*^{atl}
80	50h	01010000	P	↓	112	70h	01110000	p	*F9*^{alt}
81	51h	01010001	Q	*PgDn*	113	71h	01110001	q	*F10*^{atl}
82	52h	01010010	R	*insert*	114	72h	01110010	r	
83	53h	01010011	S	*del*	115	73h	01110011	s	←^{ctl}
84	54h	01010100	T		116	74h	01110100	t	→^{ctl}
85	55h	01010101	U		117	75h	01110101	u	*end*^{ctl}
86	56h	01010110	V		118	76h	01110110	v	*pgdn*^{ctl}
87	57h	01010111	W		119	77h	01110111	w	*home*^{ctl}
88	58h	01011000	X		120	78h	01111000	x	*I*^{alt}
89	59h	01011001	Y		121	79h	01111001	y	*2*^{alt}
90	5Ah	01011010	Z		122	7Ah	01111010	z	*3*^{alt}
91	5Bh	01011011	[		123	7Bh	01111011	(	*4*^{alt}
92	5Ch	01011100	\		124	7Ch	01111100	|	*5*^{alt}
93	5Dh	01011101	]		125	7Dh	01111101	]	*6*^{alt}
94	5Eh	01011110	^	*F1*^{ctl}	126	7Eh	01111110	~	*7*^{alt}
95	5Fh	01011111	_	*F2*^{ctl}	127	7Fh	01111111	“	*8*^{alt}

Dec	Hex	Binary	Key	Ext	Dec	Hex	Binary	Key	Ext
128	80h	10000000		*9*^{alt}	160	A0h	10100000		
129	81h	10000001		*0*^{alt}	161	A1 h	10100001		
130	82h	10000010			162	A2h	10100010		
131	83h	10000011			163	A3h	10100011		
132	84h	10000100		*pgup*^{alt}	164	A4h	10100100		
133	85h	10000101			165	A5h	10100101		
134	86h	10000110			166	A6h	10100110		
135	87h	10000111			167	A7h	10100111		
136	88h	10001000			168	A8h	10101000		
137	89h	10001001			169	A9h	10101001		
138	8Ah	10001010			170	AAh	10101010		
139	8Bh	10001011			171	ABh	10101011		
140	8Ch	10001100			172	ACh	10101100		
141	8Dh	10001101		↑^{ctl}	173	ADh	10101101		
142	8Eh	10001110			174	AEh	10101110		
143	8Fh	10001111			175	AFh	10101111		
144	90h	10010000			176	B0h	10110000		
145	91h	10010001		↓^{ctl}	177	B1h	10110001		
146	92h	10010010		*ins*^{ctl}	178	B2h	10110010		
147	93h	10010011		*del*^{ctl}	179	B3h	10110011		
148	94h	10010100			180	B4h	10110100		
149	95h	10010101			181	B5h	10110101		
150	96h	10010110			182	B6h	10110110		
151	97h	10010111			183	B7h	10110111		
152	98h	10011000			184	B8h	10111000		
153	99h	10011001			185	B9h	10111001		
154	9Ah	10011010			186	BAh	10111010		
155	9Bh	10011011			187	BBh	10111011		
156	9Ch	10011100			188	BCh	10111100		
157	9Dh	10011101			189	BDh	10111101		
158	9Eh	10011110			190	BEh	10111110		
159	9Fh	10011111			191	BFh	10111111		

Dec	Hex	Binary	Char/Key		Dec	Hex	Binary	Char/Key	
192	C0h	11000000			224	E0h	11100000		
193	C1h	11000001			225	E1h	11100001		
194	C2h	11000010			226	E2h	11100010		
195	C3h	11000011			227	E3h	11100011		
196	C4h	11000100			228	E4h	11100100		
197	C5h	11000101			229	E5h	11100101		
198	. C6h	11000110			230	E6h	11100110		
199	C7h	11000111			231	E7h	11100111		
200	C8h	11001000			232	E8h	11101000		
201	C9h	11001001			233	E9h	11101001		
202	CAh	11001010			224	EAh	11101010		
203	CBh	11001011			235	EBh	11101011		
204	CCh	11001100			236	ECh	11101100		
205	CDh	11001101			237	EDh	11101101		
206	CEh	11001110			238	EEh	11101110		
207	CFh	11001111			239	EFh	11101111		
208	D0h	11010000			240	F0h	11110000		
209	D1h	11010001			241	F1h	11110001		
210	D2h	11010010			242	F2h	11110010		
211	D3h	11010011			243	F3h	11110011		
212	D4h	11010100			244	F4h	11110100		
213	D5h	11010101			245	F5h	11110101		
214	D6h	11010110			246	F6h	11110110		
215	D7h	11010111			247	F7h	11110111		
216	D8h	11011000			248	F8h	11111000		
217	D9h	11011001			249	F9h	11111001		
218	DAh	11011010			250	FAh	11111010		
219	DBh	11011011			251	FBh	11111011		
220	DCh	11011100			252	FCh	11111100		
221	DDh	11011101			253	FDh	11111101		
222	DEh	11011110			254	FEh	11111110		
223	DFh	11011111			255	FFh	11111111		

Section 4: Converting Floating Point Programs to Integer

The integer version of MegaBasic is capable of running all programs that ran under earlier versions of MegaBasic which did not support integers. It is 100% compatible with such earlier programs and you do not have to make any changes. Since integers can be read from and written to files, the **READ** and **WRITE** statements of earlier programs are automatically altered when initially loaded to ensure that only floating point values are written to the file. This feature is more fully discussed later in these application notes.

Integers provide substantial gains when they replace floating point arithmetic, array subscripts and string indexing. To achieve 100% compatibility, MegaBasic assumes all numeric variables are to be floating point unless you specifically state otherwise. Hence your program cannot take advantage of the performance improvements provided by MegaBasic integers until you change your program. This is actually a very easy process for most programs and is accomplished by following the steps described below:

- □ Thoroughly read and understand all information described for this process.
- □ After *bringing up* MegaBasic, load the program to be converted to integer operation into your workspace. MegaBasic will immediately scan this program (if it was created under a non-integer MegaBasic) and automatically insert the reserved word **REAL** after each **READ** and **WRITE** throughout your program. This is done to ensure that floating point numeric file transfers do not suddenly become integer transfers, with erroneous results.
- □ In virtually all programs, the overwhelming majority of numeric variables and functions are used to store and return integer values. With this in mind, place the following **DEF** statement at the top of your program to set all (numeric) variables and functions to an integer type:

  ```
  DEF INTEGER "A-Z"
  ```

- □ Carefully examine your program to determine if it uses any variables or functions in a floating point capacity. If it does use floating point, make a list of the names of all such variables and functions throughout your program. With this done, insert the following statement after **DEF** statement described above:

  ```
  DEF REAL name1, name2, name3, name4, ...
  ```

  where the list of names is the list of real variables found in your program. Be sure to specify empty parentheses ( ) on names of arrays. Additional **DEF** statements may be used if your list is longer than one program line. Do not include any floating point function names in these **DEF** statements. Instead, insert the word **REAL** into each formal **FUNCtion** definition line immediately before the **FUNC** reserved word:

  ```
  DEF REAL FUNC RTOTAL(LIST_VBL).
  ```

- □ Go through your program looking for divide operations (/). For each one found, examine how it is used and determine if an integer divide (**DIV**) could be used in its place. A regular divide is an exclusively floating point operation which is slow in floating point and slower dividing an integer by an integer (because of the extra conversions). The **DIV** operation with integers is from 5 to 8 times faster.
- □ Remove the reserved word **REAL** from all **READ** and **WRITE** statements which do not involve floating point numeric data transfers.

- Thoroughly test your program and ensure that the results it obtains are identical with those obtained under the earlier non-integer version of MegaBasic under which it was originally developed.

This procedure will be effective and straightforward with most programs. As you become accustomed to integers in your daily programming work with MegaBasic, you will find your own ways of using them suited to your style of programming. This conversion procedure is intended only as a bridge to get you started.

Section 5: Loading Programs from Earlier Z80 Versions

Programs developed under prior versions of MegaBasic will likely be named with a *.zba* suffix (secondary or type name). Files with other than the *.pgm* default suffix must be spelled out in full in order to **LOAD** them. MegaBasic programs starting with the Version 4.0 series use a totally different internal encoding from programs developed under earlier versions. When programs using the earlier encoding are **LOADed**, MegaBasic Versions 4.0 and later perform a 100% translation of the program into the newer form.

Several enhancements to the language syntax have been made which are incompatible with pre-version 4.0 programs. However when such programs are **LOADed** the automatic translator makes all the appropriate program modifications as needed to be 100% compatible. When developing new source code under Version 4.0 and later, you will have to use the new syntax because no translation is done after its initial **LOAD**. These relatively minor modifications are all described below:

- To allow later extension of **PRINT** formatting, all static formats must be enclosed in quotes (“ or ””). Lower case as well as upper case format characters are accepted.
- Since functions may be arbitrarily named, their **DEF** statements must contain the reserved word **FUNC**, e.g., **DEF FUNC TOTAL(X,,Y)**. The translator leaves **FN** in all earlier programs only to ensure uniqueness of identifiers. You can later rename functions to anything you like.
- Since names must be unique, the earlier rule that arrays and scalars can be given the same name is no longer applicable. To ensure uniqueness of array names when translating earlier programs, MegaBasic doubles the first character of every array reference in the program. You can change their names after this to suit your own special needs. For example A(I) becomes AA(I), X3a) become XX3a), and so on.
- String parameters in **DEF** statements are passed as localized values just like numeric parameters (i.e., as local variables). To support the earlier method of simple string parameter copying which is not re-entrant, each parameter is preceded by a percent sign (%). This *is only* done in the function **DEF** statement and never in references to functions. The only reason for using this method for passing strings is to be compatible with programs that rely on the side-effects of that approach or for the slightly faster execution it provides. For the sake of completeness and consistency, numeric parameters may be passed in the same manner by flagging them with percent signs as well.
- **FILL** and **EXAM** statements (Chapter 7, Section 3) have been made consistent with **READ** and **WRITE** statements in the way they interpret their data list. Byte-oriented transfers *must* be preceded by an ampersand (&); word-oriented transfers must be preceded by an at-sign (@). If neither of these lead-in characters is present, then data is transferred just like the other file operations: numeric floating point and packed-string with a string header.

Running North Star BASIC Programs

- Users with programs developed under North Star BASIC can run them under MegaBasic with very little work. However, do not load them as text files or type them in from the keyboard. The procedure described below explains how to convert your binary North Star BASIC programs to binary MegaBasic using an automatic translator. ASCII program text files have no North Star internal structures, which are needed to tell the translator that North Star BASIC programs are being loaded and what to translate. By loading these programs as text files, you are bypassing the automatic translator, creating an enormous conversion job for yourself that is totally unnecessary. To convert any North Star BASIC program properly, just perform the following steps:
- Through whatever means available to you, get your North Star BASIC program files off of their North Star diskettes (DOS or CP/M-80) and onto files under your 8086 operating system (e.g., CP/M-86 or preferably MS-DOS). MegaBasic cannot help you with this step.
- RUN your North Star BASIC program through the NSB2ZBA.pgm utility provided with MegaBasic. This converts your program into ZBASIC format, the Z80 version of MegaBasic.
- LOAD the resulting program file into MegaBasic. MegaBasic detects ZBASIC programs and automatically converts them to MegaBasic internal program format.
- **SAVE** the translated program back to a disk file. Now test your the program and make any other changes to it that might be necessary for it to operate correctly in the 8086 environment, then be sure to **SAVE** the corrected program again. See the list of incompatibilities provided below for reference. Also read Appendix D, Section 5 earlier for other changes that may be required.

This process is very easy and you will find that you can convert dozens of programs in just one sitting after getting used to it. Be sure to read the rest of the MegaBasic Reference Manual because there are many capabilities that go far beyond North Star BASIC that are too numerous to go into here. All the commands in North Star BASIC are supported, but in a much expanded form. See Chapter 2 for details on the entire MegaBasic command set. The North Star BASIC incompatibilities to look for include:

- Enhancements to MegaBasic have been incorporated to areas that have heretofore been considered errors in standard BASIC. Programs that rely on such errors, so that ERRSET recovery techniques can switch to alternate routines, might run into difficulty since such *errors* may no longer exist.
- All **ERRSETS** (Chapter 6, Section 4) must be examined for compatibility. In particular, calls to **GOSUBS** or **FNS** that setup **ERRSETS** for the program will not work, as the scope of each **ERRSET** is confined to the execution of the invoking subprogram.
- The **DATA-READ** pointer is preserved during **GOSUB** and **FN** calls. If the subroutine itself alters the **READ** pointer for its purposes, this mechanism conveniently localizes it until a normal **RETURN** is processed. Thus **GOSUB** calls expected to revise the **READ** pointer before returning will not work.
- Token definitions since Release 5.0 will usually be treated differently in standard BASIC than in MegaBasic. As of Release 5.2, only three examples are known: **LET**, **FILEPTR** and **FILESIZE**. The North Star **LET** token lists as **WHILE** and must be deleted, **EILEPTR** ends up as A MegaBasic **MOD** function, and **FILESIZE** becomes the MegaBasic **SWAP** statement.

- **MEMSET** and **LINE** statements do not exist because they are completely unnecessary. But in MegaBasic **LINE** lists erroneously as **LOCAL** and such statements must be deleted from the program.
- Your program **REMarks** may have strange spelling errors due to the keyword differences in MegaBasic. The quickest and easiest way to fix these is with the command: **EDIT REM** which extracts all **REMarks** for your editing (Chapter 2, Section 3).
- Do not attempt any of the North Star Personalization procedures on MegaBasic as they are not the same. Instead, use the **CONFIG** program described in Appendix C, Section 3, which implements all personalization options available.
- Since MegaBasic format strings are always string expressions, format specifications that include a dollar sign ($) may appear to MegaBasic as dynamic formats, which will subsequently execute incorrectly. Such formats appear to begin with a string variable (e.g., Z$12F2, C$8I, etc.). You can fix this problem by surrounding each such format with quotes ( ), for example: %“Z$12F2”, %“C$8I”, etc.
- **FNDEFinitions** must appear as the 1st statement on the line that you define them. North Star BASIC allows it to appear anywhere in the line which results in much more time spent in the load-up process prior to program execution.
- Through system errors in some North Star BASICs, program lines may erroneously contain control characters. MegaBasic will display such bad characters as question marks (?) for your correction.
- Lines, commands, and direct statements may be entered in any combination of upper and lower case. MegaBasic converts all lower case letters not inside quotes ( ) to upper case before proceeding.
- User defined names with more than one character (e.g., Tl, A$, FNS6, Z3$, etc.) must be entered run together without any inserted spaces to avoid a syntax error. The left parentheses following function, string, or array names is considered part of the name when applying this rule.
- Line-feeds may be entered into the program text for longer lines, and so that spaces can be inserted between lines to make the text more readable. When listed, each line-feed expands into a LF-CR sequence.

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z