MAGIC/L™

User's Manual

LOKI ENGINEERING, INC.
55 Wheeler St., Cambridge, MA 02138 (617) 576-0666

MAGIC/L™

User's Manual

Arnold Epstein
Jeffrey D. Morris


LOKI ENGINEERING, INC.
55 Wheeler St., Cambridge, MA 02138 (617) 576-0666

Loki Engineering, Inc.
55 Wheeler Street
Cambridge, MA Ø2138
(617) 576-Ø666

Revision 2.5Ø.1
January 1984

Table of Contents

MAGIC/L (pronounced magical) is a general purpose interactive language. It is an outgrowth of STOIC which itself sprouted from FORTH. MAGIC/L was evolved as a "programmer's language" primarily because it was being used as it evolved. Many of the features of MAGIC/L were implemented and re-implemented until a proper mix of readability and ease of programming was accomplished.

We feel that we have created a unique and extremely powerful programming tool. We have used it for a broad range of applications programs including a mailing-list program, hardware diagnostics and debugging routines, an interactive image processing system, an interactive data-base query system, a turn-key interactive process control system, and a cross- assembler program. In short, the utility of MAGIC/L is limited only by the programmer's imagination.

MAGIC/L defies categorizing. We have attempted to include desirable concepts and features from many programming languages, but tried to avoid the painful side-effects associated with any single language. It is both a very high level language and a low level language at the same time. MAGIC/L is equally adaptable to bottom-up and top-down programming. MAGIC/L is not strictly a compiler or an interpreter. We call it an "incremental compiler."

The original implementation of MAGIC/L (it was called MAGIC at the time) was on a Data General Eclipse computer running the AOS operating system. Since that original breakthrough from the reverse-polish notation of FORTH into the more conventional (and more readable) algebraic (forward) notation, MAGIC/L has evolved to the point where it is a mature, stable, and easy to use programming environment.

This manual is meant to help the reader learn to program in MAGIC/L. The getting started section in Chapter Ø should be used to gain immediate hands-on experience in programming MAGIC/L. It will quickly become obvious that the power of MAGIC/L stems not from any individual feature, but from the coalescence of these features into an interactive extensible programming environment.

Most sections of this manual serve as reference material. We encourage the reader to try out the various examples and to experiment with his/her own. Since MAGIC/L provides such a broad spectrum of programming tools, it is impossible to gain a complete understanding of the language from a single reading. We recommend that the manual be reviewed often as your experience with MAGIC/L grows. You can consider yourself an expert when a complete reading of the manual doesn't result in new insight.

The question often arises: "Does MAGIC stand for something?" The answer is that we have come up with a couple of phrases for which MAGIC is an acronym such as "Mnemonics Are Generally Idiotic Concoctions" or "Multi-purpose Assembler and General-purpose Interactive Compiler." These names were invented after the language was named. The real origin of the name comes from the almost unanimous first reaction to seeing MAGIC in operation: "How did you do that?" The answer we kept giving was simply: "It's magic!"

A large number of friends and colleagues have provided thoughtful comments both in the development of MAGIC/L and in the preparation of this manual. We would like to give special thanks to these early believers: Israel Brill, Maureen Conroy, Arthur Fox, J. Garrett Jernigan, Dave King, Eric Mandel, Steve Murrey, Denis Pelli, Ellen Ralph, Johanna Rothman, Ethan Schreier, Joe Schwarz, and J. Spangler. We are forever indebted to Marian Sonnenfeld Epstein for opening her home to a fledgling company.

Arny Epstein
Jeff Morris

August 1981

You are about to enter the world of MAGIC/L. Programming in MAGIC/L is unlike programming in ordinary compiler languages. The high level of interactivity provided by MAGIC/L will tend to change the way you view your relationship with the computer.

After reading this manual, the new MAGIC/L programmer will be creating some simple programs within a day or two, but understanding of how to take advantage of the MAGIC/L interactive environment is a gradual process. Within a week or two most programmers gain an appreciation of the MAGIC/L environment to the extent that it becomes tedious to return to a traditional compiler language.

This chapter is meant to be an interactive tutorial to give the flavor of the MAGIC/L environment to the new MAGIC/L programmer. We strongly suggest that you try each example in the "Getting Started" section as you proceed.

Following the tutorial is a summary of the structure of the remainder of this manual and a description of additional documentation which is available.

Getting Started

Before getting into the formal specification of MAGIC/L, it would be best to show a sample dialog at the keyboard. If you have access to a computer that has MAGIC/L, you should enter the examples given below as you go. If some of the terms used in this section are not familiar, bear with us, a full explanation is in the chapters that follow.

To enter the MAGIC/L environment in most systems, simply type MGL in response to the system prompt. If this does not work, refer to the installation guide supplied with your release material or ask your system manager. When MAGIC/L is initialized, a message will appear on the screen:

```
MAGIC/L  Rev X.XX
   nnnnn Bytes Left
mgl>
```

The angle bracket "mgl>" is the prompt issued by MAGIC/L. In this sample dialog, all user entered characters will be on the

lines that begin with a prompt. In the sample dialog below, input to MAGIC/L is printed in UPPER case while our commentary is in lower case (You may use either since MAGIC/L is entirely case-insensitive).

In addition, MAGIC/L supports the system's keyboard control characters (rubout, kill line, interrupt, etc.). Each line you type in should be terminated by a carriage return. The carriage return tells MAGIC/L to compile one line of text. If the compiled code is a complete unit, it is executed immediately. In any case, a new prompt is given requesting more input.

If you make a mistake entering the dialog, MAGIC/L will respond with an ERROR or WARNING message. If you wish to abort a multi-line input sequence, type your system's standard keyboard interrupt sequence. For CP/M this is a control-A, other systems use control-C. This interrupt will abort any function and completely reset MAGIC/L. Occasionally, serious errors will damage the program so that it is necessary to leave the program and restart it. Under CP/M this is accomplished with a control-C which forces a "warm boot."

Type in several carriage returns.

```
mgl>                        commentary will appear out here
mgl>
mgl>
```

MAGIC/L responds with several prompts.

Now ask MAGIC/L to type out a number.

```
mgl> PRINT 5
       5
mgl>
```

That was easy, now try some simple arithmetic. The spaces are required between each word:

```
mgl> PRINT 2 * 3 + 5
      11
mgl>
```

The PRINT command will be used repeatedly in this dialog so that you can see the results of all the computations.

At this point it would be useful to define some variables. In MAGIC/L, all variables must be defined before they are used.

```
mgl> INTEGER X Y Z              Define three variables
mgl> X := 4                     The value 4 is assigned to X
mgl> PRINT X                    Ask MAGIC/L to type the value of X
       4
mgl> Y := X + 2                 Assign a value to Y
mgl> PRINT X , Y                The comma separates arguments
       4       6
mgl> PRINT MIN ( X , Y )        Type the value returned by the
       4                        minimum function
mgl> PRINT MAX ( X , Y )        and the maximum function
       6
mgl>
```

Although each line is compiled as it is entered, the compiled code is not always executed immediately. If the syntax of the line is not complete, the execution is delayed until further lines complete the syntax. When this occurs, multiple brackets will be displayed ">>" in the prompt to signify that additional input is needed before execution.

MAGIC/L can iterate using a DO-LOOP structure. As will be explained in a later section, DO-LOOPs have their own index counter called `I , which are treated as INTEGERs in expressions.

```
mgl> DO Ø , 5                   start I at zero
mgl>> PRINT I                   print I at each iteration
mgl>> LOOP                      this terminates the structure
       Ø                        and allows execution
       1
       2
       3
       4
       5
mgl>
```

MAGIC/L also has a variety of control structures. The simplest is the IF-ENDIF structure:

```
mgl> IF ( X == 4 )
mgl>> PRINT X
mgl>> ENDIF
       4
mgl>
```

Control structures may be nested:

```
mgl> DO Ø 7
mgl>> IF ( I > X )
mgl>>> PRINT I                    note the extra ">"
mgl>>> ENDIF
mgl>> LOOP
        5
        6
        7
mgl>
```

While the examples shown so far may seem rather simplistic, they are very typical of the type of dialog that is normally entered from the keyboard. As a program is being developed, it is very easy to test each subroutine as it is written. While subroutines are normally entered into a source file with a text editor and then compiled, it is also possible to write subroutines at the keyboard.

Let's write a simple routine to print the lesser of X and Y. Notice as you enter the code that the prompt changes. The "-" in the prompt indicates that you are within a DEFINE-END block.

```
mgl> DEFINE MINXY
mgl-> PRINT MIN ( X Y )
mgl->> END
mgl>
```

At this point a new definition has been added to the MAGIC/L vocabulary. Since there are no input or output arguments to MINXY, all that is needed to execute this routine is to enter its name:

```
mgl> PRINT X , Y                  Lets remember the values of X and Y
        4       6
mgl> MINXY                        Execute the new routine
        4
mgl>
```

Now change the value of X and repeat the process:

```
mgl> X := 7
mgl> MINXY
        6
mgl>
```

Now try something more advanced! Let's define a function:

```
mgl> DEFINE CELSIUS INTEGER      convert Fahrenheit to Celsius
mgl-> INTEGER FARENHEIT          declare the input argument
mgl-> CELSIUS := ( FARENHEIT - 32 ) * 5 / 9
mgl->> END
mgl>
```

The first line of this definition block specifies the name CELSIUS for the function, and that it is an INTEGER function (i.e. it returns an INTEGER value). The next line specifies the calling sequence to the routine; in this case the routine accepts a single integer input value. (The DEFINE-END syntax is described in detail in chapter 4).

Your new function does not print its answer, it returns an INTEGER value. The function CELSIUS may be used anywhere an INTEGER value can be used. For example:

```
mgl> PRINT CELSIUS ( 32 )
       Ø
mgl> PRINT CELSIUS ( 78 )
      25
mgl>
```

Last but not least, lets create a COMMAND. The ability to simply specify that a routine's calling sequence not require parentheses is a unique and powerful feature of MAGIC/L. For example, let's write a command which uses the CELSIUS routine defined above and prints out the conversion:

```
mgl> DEFINE CEL COMMAND
mgl-> INTEGER FAHR
mgl-> PRINT FAHR , " DEG-F =" , CELSIUS ( FAHR ) , " DEG-C"
mgl->> END
mgl>
```

Now try out our new command:

```
mgl> CEL 78
      78 DEG-F =      25 DEG-C
mgl> CEL 32
      32 DEG-F =       Ø DEG-C
mgl> CEL 5 * 6 + 2
      32 DEG-F =       Ø DEG-C
```

At this point we have touched on most of the basic features of MAGIC/L. In the following chapters complete descriptions all aspects of MAGIC/L are given. Although the examples that follow are not presented in dialog format, ALL code in this manual can be entered interactively from the keyboard. We encourage you to try out the examples while reading the manual.

Exiting from MAGIC/L

When you wish to exit MAGIC/L simply type BYE. The BYE routine returns control to the program which executed MAGIC/L (usually the system command interpreter program).

Structure of This Manual

The remainder of this manual is logically divided into four sections.

1. Chapters 1 through 5 of this manual document the basic features of MAGIC/L. It should be possible to write simple MAGIC/L programs after completing these chapters.

2. Chapters 6 and 7 document the MAGIC/L formatted input and output facilities.

3. Chapters 8 through 12 describe more advanced features of MAGIC/L.

4. The appendices act as a reference document for the numerous routines included with MAGIC/L.

A brief description of each chapter follows:

Chapter 1 describes the basic syntax of a MAGIC/L statement.

Chapter 2 describes the intrinsic data-types supported in MAGIC/L and how to define variables of these data-types. Also described are the arithmetic, relational, and bitwise operators supported by MAGIC/L, and the combination of variables and operators into expressions.

Chapter 3 describes the various control structures contained in MAGIC/L. Both branching and looping structures are described.

Chapter 4 supplies a detailed description of how to define functions, subroutines and COMMANDs in MAGIC/L.

Chapter 5 provides a description of how to create a MAGIC/L program and programming environment.

Chapter 6 gives a complete description of the MAGIC/L formatted print facility. The PRINT command and all of its associated directives as well as the ENCODE routine are discussed here.

Chapter 7 describes the INPUT facility which complements the PRINT facility.

Chapter 8 describes the MAGIC/L record structure facility. Both the creation (using the RECORD statement) and the usage (using WITH or colons) of user-defined data types is discussed.

Chapter 9 documents how to create initialized data structures.

Chapter 1Ø explains the unique MAGIC/L feature of creating new classes of routines called "action classes".

Chapter 11 describes the MAGIC/L symbol table (or vocabulary) usage and explains how to create new vocabulary branches to suit the needs of an application.

Chapter 12 describes a variety of advanced features such as executing a routine through a pointer to the routine, recursion, and user definable error handling. This chapter also includes several complete programming examples.

Appendix A contains a description of the various functions provided with MAGIC/L. These functions are categorized in the appendix and the functions in each category are listed in alphabetical order.

Appendix B discusses the MAGIC/L file I/O facility. These I/O routines are supported in all implementations of MAGIC/L providing source level portability for MAGIC/L code.

Appendix C describes the error and warning reporting of MAGIC/L and lists all of the MAGIC/L error codes.

Appendix D discusses a variety of common errors and shows debugging techniques useful to the MAGIC/L programmer.

Appendix E documents the space requirements of various MAGIC/L structures and discusses coding for space or speed efficiency.

Appendix F is a brief listing of MAGIC/L syntax and the calling sequences of all of the library routines. This appendix can be extracted to serve a reference guide to MAGIC/L.

Additional Documentation

This manual is intended to be applicable to all implementations of MAGIC/L. Another document is supplied with each MAGIC/L release which describes facilities that are specific to a system or machine. This will include:

1. A release notice which contains installation instructions and a description of any program or documentation changes since the previous release of MAGIC/L.

2. An Operating System supplement which describes the Access of MAGIC/L to system calls as well as other system dependent considerations.

3. An assembler manual describing the interactive MAGIC/L assembler for the host processor.

4. A toolbox manual which describes a variety of utilities which support the MAGIC/L environment. Included in the toolbox for most implementations are routines for:

   Editing the previous command line
   Editing the previous definition
   Generating and Listing a cross reference
   Listing and modifying the current symbol table

Syntax of a MAGIC/L statement

A statement in MAGIC/L is a series of strings called "tokens". Each token in the statement falls into one of three categories:

1) a previously defined "WORD" (or "entry")
2) a literal
3) an "undefined" token

For example, in the following MAGIC/L statement:

```
PRINT  55001 , YSIZE + 5
=====  ----- = ===== = -
```

The INTEGER literals (55001 and 5) are underlined once, and the 4 WORDS (PRINT , YSIZE and +) are underlined twice.

All tokens in a MAGIC/L program fall into one of the above general categories. All tokens must be separated by special characters called word delimiters in order to be recognized as distinct strings (e.g. YSIZE+5 and YSIZE + 5 are NOT the same).

Spaces, tabs and new-lines act as word delimiters in a MAGIC/L statement. Spaces will imply word delimiters of any type in this manual.

MAGIC/L Statements

By default each line of MAGIC/L code is assumed to contain a single statement. If more than one statement is coded on a single line, the statements must be delimited by the statement delimiter ";;". If one statement is to span more than one line, then a line continuation "^" must be specified. The "^" must be surrounded by spaces. In summary MAGIC/L statements can appear in three ways:

1. Single statement on a line (default assumption)

```
<statement>
```

2. Multiple statements on a line (statement delimiter required)

```
<statement> ;; <statement>
```

3. Statement covering more than one line (line continuation required)

```
<partial-statement>  ^
<partial-statement>
```

The following example shows two assignment statements coded on a single line separated by the statement delimiter ";;".

```
X := 5 + 3  ;;  Y := X * 2
```

Here is an example of a statement which continues to a second line. (Any number of line continuations are acceptable):

```
X := Y * ( X + 5 * Z )   ^
       + ( Z * Z * Z )
```

Without the up arrow at the end of the first line, the assignment statement would have been assumed to be complete. The up arrow allowed for continuation onto the next line.

Dictionary entry names

Most of the tokens in a MAGIC/L statement will be defined WORDS. These words are maintained as an ordered list called the DICTIONARY. WORDS are often called "dictionary entries" or simply "entries". Entries are referenced by name. The entry name may be any ASCII string up to 15 characters and may be composed of any of the following characters:

Alphabetics:
A-Z or a-z (interchangeably)
Numerics:
Ø-9
Special characters:
! " # $ % & ' ( ) * + , - . / [ ] ^ ~

Only the first 5 characters of a name plus its length are maintained in the dictionary. This is typically sufficient to keep entry names unique. For example, INTERFACE and INTERLACE are not unique entry names (they differ at the 6-th character and have equal lengths) but INTERLACE and INTERACT are unique (they differ in length).

Upper and lower case alphabetics may be used interchangeably. When forming the name all ASCII characters with codes between 141 and 176 octal are equivalenced to the corresponding character between 1Ø1 and 136 octal. (e.g "{" may be used interchangeably with "[")

Note: It is suggested that names be limited to alphanumerics and the special characters dollar sign ($), question mark (?), period (.), and under-bar (_), and that names begin with non-digits. Doing this will assure compatibility with future versions of MAGIC/L with a more complex parser (and therefore more syntax restrictions).

Classes of Dictionary Entries

The basic MAGIC/L dictionary contains about 500 words. Many of these words are subroutines, others are variables. The user program combines these words to form new words that extend the dictionary. Because the user program becomes part of the evolving language, MAGIC/L is called an "extensible" language.

Following is a short list of the classes of dictionary entries. This list is not complete nor could it ever be, because it is possible to create new types of words which would not fit into any predefined category (see Chapter 10).

1. Variables and arrays.
   WORDS which store values.

2. Subroutines and Functions
   WORDS which take actions and/or return values

3. Control words
   These special WORDS (DO IF etc) act as compiler "keywords" to build control structures. Syntax checking is performed by the compiler to assure proper structures are built.

4. Operators
   A set of arithmetic operators (+, -, *, / etc.) are defined as WORDs. Relational operators (== <> etc.) are defined using BASIC style names. Operators maintain precedence and check for mixed mode operation.

5. Defining words
   This class of words allow the user to create new words of various classes. Words like INTEGER , ENTRY and DEFINE are considered defining words. RECORD is a word which defines defining words (phew!)

6. Punctuation
   Characters that are punctuation for the benefit of the compiler are defined as WORDs. This includes parentheses, commas, comment or line continuation characters, etc. This is important since it implies that punctuation words must be separated by spaces to be recognized.

7. Assembler Mnemonics

   Since MAGIC/L has a built in assembler, all the standard mnemonics are defined as WORDs. Assembly code may be interspersed with MAGIC/L code. This allows for time critical MAGIC/L routines to be recoded for speed while leaving the original MAGIC/L code as documentation.

8. System Calls

   Since MAGIC/L executes under an operating system, most system calls that are available to the assembly language programmer are also available to the MAGIC/L programmer. Thus the MAGIC/L programmer has interactive access to system functions.

9. Immediate commands

   Certain words have evolved to handle special points of syntax. These words often have no analog in other languages but have proven to be very useful in MAGIC/L. Since many of these words act on the following words of code, and their action is usually taken at compile time, they are called Instant or Immediate commands. Words such as: BASE SIZE and ASCII are Immediate commands.

Literals

In addition to words, there are various types of ASCII strings called LITERALs. All numbers are recognized by the compiler as literals. Numeric literals follow certain simple rules:

Rules for INTEGER and LONG literals

1) Optional sign character is permitted
2) The first non-sign character must be a digit.
3) The number may be terminated by a radix indicator.
4) An L as the final character explicitly forces the value to be treated as a LONG integer. (Otherwise, a value less than 65536 implies a 16-bit integer)

Rules for REAL literals

1) Optional sign character is permitted
2) The first non-sign character must be a digit.
3) The decimal point is required
4) REALs must have at least one digit following the decimal point
5) REALs are always interpreted in decimal

Conventions used in the syntax description:

[ ]	encloses optional portions
{ }	enclose a list of options (choose one)
d	represents one or more numeric digits
n	represents zero or more alphanumeric characters (since A-F are used in hexadecimal)
r	represents a radix specifier K specifies octal X specifies hexadecimal . specifies decimal no specifier means that the current radix is used.

INTEGER and implicit LONG literals:

```
[{+ -}]dn[r]
```

Explicit LONG literals

```
[{+ -}]dn[r]L
```

REAL literals

```
[{+ -}]d.d[E[{+ -}]d]
```

Examples:

-5	INTEGER	literal
55K	INTEGER	literal
+1Ø.	INTEGER	literal
1Ø5L	LONG	literal (explicit)
ØFFFXL	LONG	literal (explicit)
75ØØØ.	LONG	literal (implicit)
-5.Ø	REAL	literal
7.4E5	REAL	literal

In addition to numeric literals, string literals are supported by MAGIC/L. String literals are defined in one of two ways:

1) Surrounding the string in quotes (")

```
"This literal contains spaces"
"string-literal"
```

This allows the programmer to define strings containing spaces.

2) Beginning the string with an apostrophe (')

```
'This-does-not-have-spaces
'HELLO
```

Strings of this form are terminated by a space, and therefore cannot contain any spaces

Both forms of string literal allow any OCTAL value to be inserted by surrounding the value with angle brackets. For example, to ring the bell (code 7) on a terminal print:

```
"Ding<7>"
```

If the text following the open bracket is not a legal octal byte value, then the entire string is taken literally. For example:

```
"Who<77>"  prints as:   Who?
"<name>"   prints as:   <name>
```

Undefined Words

If a token in a MAGIC/L statement is neither a dictionary entry nor a literal, it is considered UNDEFINED. By default, undefined tokens are identified with a WARNING message at compile-time, and will cause an error on any attempt to execute the word at run-time. For example, in the following definition assume that the token XY is undefined:

```
DEFINE MULTXY
  XY := X * Y
END
```

In this case, at the time the word MULTXY is being defined the message:

```
Warning: Undefined word
Token: XY
```

will be displayed on the terminal (and the bell will ring), but compilation will continue with the next definition. However, at the time the word MULTXY is executed, the message:

```
Error: Attempt to execute undefined word
```

will be displayed and execution will be aborted.

When entering code directly at the keyboard, all WARNINGs trigger an ABORT. In this case, the UNDEFINED warning will immediately stop the compilation.

Comments

Comments may appear in two forms in MAGIC/L. The first form is convenient for short comments interspersed with code, while the second form is more convenient for longer descriptions which might span several lines. The basic comment is initiated by a semi-colon (delimited by spaces of course), and extends through the end of the current input line. The extended form is initiated by the two character sequence <* and is active until the terminating sequence *> is encountered. Since the extended comment field continues until terminated, this form may (and typically does) span several lines. Here are some examples of commenting:

```
; This is a one line comment

X := 5 * Y        ; This is a legal comment too

<*
  The extended comment is often used to give
  an overall description of a routine or a module
*>
```

All characters within the comment field are ignored by the MAGIC/L compiler. Thus, the line continuation character, "^", will not work if it follows a ";". The proper way to comment a multi-line statement is:

```
X := A * B  ^     ; Sum of ...
   + C * D        ;  products
```

[This page left blank]

Chapter 2 Variables, Operators, and Expressions

This chapter deals with variables, operators, and expressions. Variables are the basic data items used in a MAGIC/L program. Declarations are used to define the variables in a program, and to specify the data-type associated with the variable. Operators take actions upon variables. Expressions combine operators and numeric values (variables, constants, and functions) to create a new value and associated data-type.

Intrinsic data types

The types of data items that are "built-in" to MAGIC/L are known as intrinsic data-types. These items are the most basic element of data manipulation. The intrinsic data-types in MAGIC/L are:

CHAR	8-bit unsigned integer values
INTEGER	16-bit signed integer values
LONG	32-bit signed integer values
ADDRESS	Variable containing an address (Same as INTEGER for 16-bit implementations) (Same as LONG for 32-bit implementations)
REAL	32-bit (single-precision) floating point

CHAR variables take on special meaning when they are declared as arrays. As explained later in this chapter, such arrays may be used to contains STRING variables.

The ADDRESS data-type is used to provide source-level compatibility between 16-bit and 32-bit versions of MAGIC/L. (The usage of ADDRESS data-type is discussed below in the section on "Pointers.") In chapter 8 we will see how intrinsic data-types can be combined into more complex user-defined data-types (records).

Parameters

PARAMETERS are named 16-bit integer constants. PARAMETERs are assigned a value when defined. This value cannot be modified once the PARAMETER is defined. The syntax of a PARAMETER declaration is:

```
PARAMETER <name> := <expression>
```

The <expression> must evaluate to INTEGER type. In this example:

```
PARAMETER NWORDS := 1Ø
PARAMETER NBYTES := NWORDS * 2
```

two parameters are defined. Note that the second parameter, NBYTES, is defined in terms of the first parameter (NWORDS).

Declaring Variables and Arrays

MAGIC/L provides "defining WORDs" which provide for the declaration of variables. A declaration specifies both the name and data-type of a variable. Variables must be declared before they are used, but a declaration may appear anywhere within a program. Declarations are of the form:

```
INTEGER A B C
LONG D
REAL E
```

MAGIC/L supports one-dimensional arrays of any data-type (including user-defined). THE SUBSCRIPTING OF ARRAYS STARTS FROM ZERO. The declaration of an array is made by following the array name with an expression enclosed within parentheses. The evaluation of the expression must be a non-negative integer which specifies the number of elements in the array. Some examples of array declarations follow:

```
INTEGER X ( 1Ø )  Y ( 2Ø )  Z ( 4 * 2Ø )
REAL    RV ( 3 )
CHAR S ( 1Ø )
```

REMEMBER THAT SUBSCRIPTING STARTS FROM ZERO so that the array X above contains 1Ø elements which are subscripted Ø through 9. Note the use of an expression within the declaration of the array Z. This programming technique serves to enhance the readability of the source code. Using previously defined PARAMETERS within an array declaration is another way to enhance readability:

```
PARAMETER AASIZE := 1ØØ.

REAL AA ( AASIZE )
```

In the above example the number of elements in the array AA is specified by the value of the variable AASIZE. AASIZE has been previously declared and assigned the value 1ØØ. If other references to the size of the AA array (such as DO-LOOP counts) are specified in terms of the variable AASIZE, then it is a simple matter to modify the program for a different sized AA array.

Variables and arrays of the same data-type may be specified in the same declaration:

```
INTEGER C  X ( 1Ø )
```

In practice it is useful to comment each variable declaration as to its usage. This programming technique enhances program readability and maintainability. Let's re-declare the above example with comments:

```
PARAMETER AASIZE := 1ØØ.   ; Parametric size of AA
REAL AA ( AASIZE )         ; AA array to contain ...
```

One final comment: commas may be used in declaration statements to maintain compatibility with other languages. For example, the first declaration in this section could have been written:

```
INTEGER X , Y , Z
```

Introduction to Operators

Operators are WORDs which act on one or two values to create a new value and data-type. Arithmetic operators and bit-wise operators maintain the data-type of the input values. Relational (or logical) operators yield a value of Ø or -1 with INTEGER data-type.

Arithmetic Operators

MAGIC/L supports the following arithmetic operators:

+	Addition
-	Subtraction or Unary Negation
*	Multiplication
/	Division

A minus sign implies either subtraction or unary negation, depending on the context. Remember that a space is required following the minus sign when applied to a variable ("- X"), but not when applied to a numeric literal ("-5").

Integer division truncates any fractional part. The precedence of + and - is lower than that of * and / which is

lower than unary negation. Thus the expression:

```
- A * B + C * D
```

is evaluated as follows:

```
( ( - A ) * B ) + ( C * D )
```

Bitwise operators

Bitwise operators act on each bit as individual units. They are only defined for operands which have integer values (i.e. CHAR , INTEGER and LONG operands). [Who ever heard of an XOR of two floating point numbers anyway?] The bitwise operators are:

NOT	One's complement
AND	Bitwise AND
OR	Bitwise inclusive or
XOR	Bitwise exclusive or

Bitwise operators have precedence lower than all other operators.

Relational (or Logical) Operators

Relational operators determine the state of some relationship between operands or between an operand and zero. The result of such an operation is an INTEGER value of -1 (all bits set) when the relationship is true, and Ø when false.

Two-operand relational operators determine the relationship between two operands. These operators are used in the form

```
( A OP B )
```

where A and B are operands and OP is one of the following relations:

==	A is equal to B
<>	A is not equal to B
>	A is greater than B
>=	A is greater than or equal to B
<	A is less than B
<=	A is less than or equal to B

Single-operand relational operators determine the relationship between a single operand and zero. These operators are used in the form:

```
( A OPØ )
```

where A is the operand and OPØ refers to one of the following relations:

==Ø	A is equal to zero
<>Ø	A is not equal to zero
>Ø	A is Greater than zero
>=Ø	A is Greater than or equal to zero
<Ø	A is Less than zero
<=Ø	A is Less than or equal to zero

All relational operators have equal precedence. The precedence of relational operators is greater than that of all bitwise operators and lower than that of all arithmetic operators. For example:

```
( X > Y + Z OR A == B )
```

evaluates as:

```
( ( X > ( Y + Z ) ) OR ( A == B ) )
```

Expressions

Expressions are created by combining operands (e.g. variables, parameters and functions) with operators. An expression has two properties: a value and a data-type. All operands within an expression should be of the same data-type.

Mixed mode operations are detected and a WARNING message is given at compile time, however no type conversion is attempted. Since the result of executing a mixed-mode expression will usually be anomalous (e.g. wrong answer or program crash), MAGIC/L compiles a word that will trigger an error at runtime.

Expressions may be used in many types of statements in MAGIC/L. Use of expressions in assignment statements, command statements, and subroutine and function calls, is described in the following sections of this chapter.

Assignment statements

Assignment statements are used to assign a value to a variable. MAGIC/L supports three types of assignment operators:

1. Expression assignment operators
2. Update assignment operators
3. Initialization assignment operators

Expression assignment operators act on both an address and an expression. The syntax for expression assignment operators is:

```
<variable> <op> <expression>
```

where <op> is one of the following assignment operators:

:=	Store <expression> into <variable>
+=	Add <expression> to the contents of <variable> and store the result in <variable>
-=	Subtract <expression> from the contents of <variable> and store the result in <variable>

The += and -= operators provide an efficient way to increment or decrement the value of a variable by an arbitrary value. They eliminate the need to repeat the name of the variable on the right hand side of the assignment statement. That is:

```
A += X * 5
```

is functionally equivalent to (but more efficient than):

```
A := A + X * 5
```

Some examples of expression assignment statements are:

```
A := B + 5
A += 6
Z ( 3 ) := Z ( 5 ) AND X
```

Data type checking is performed when expression assignment statements are compiled. If the expression does not have the same data type as the destination variable, then the warning "Mixed-Mode Operation" is given. Any attempt to execute a mixed-mode assignment statement will result in the run-time error "Attempt to Execute Mixed-mode Expression."

Update assignment operators modify the current value of a variable in a particular way. The two update assignment operators supported in MAGIC/L add and subtract one from the specified variable. The syntax for these operators is:

```
<op>  <variable>
```

where <op> is one of the following assignment operators:

INCREMENT	Add one to the current value in <variable> .
DECREMENT	Subtract one from the current value in <variable> .

These commonly used assignments are provided both for efficiency and to enhance readability. Note that the following three statements (listed in order of decreasing efficiency) are all functionally identical:

```
INCREMENT X
X += 1
X := X + 1
```

Two initialization assignment operators are provided. The CLEAR operator assigns the value zero to the specified variable. The SET operator sets all of the bits in the specified variable to ones (logical TRUE or minus one). The syntax of these operators is identical to that of INCREMENT and DECREMENT. For example:

```
CLEAR X
SET Y ( 3 )
```

These operators are normally used to set a true or false value for a variable. The following two statements are functionally identical:

```
CLEAR X
X := Ø
```

CLEAR is a more efficient way of zeroing a variable. Note that SET has no meaning and is not supported for REAL variables.

String Declarations

Although STRING variables may not be declared explicitly in MAGIC/L, they are implied by the declaration of a CHAR array. For example,

```
CHAR MYSTR ( 1Ø )
```

declares a CHAR array that may be used to contain a string. MAGIC/L requires that strings be terminated by a NULL character, so the array MYSTR is large enough for a nine character string. Many computers have a requirement or a strong preference for "word aligned addressing," so it is recommended the all CHAR array declarations be of even size to maintain word alignment.

CHAR arrays may be used in the same manner as INTEGER arrays, that is, to contain a set of discrete elements. However, when a CHAR array is used in unsubscripted form, the type STRING is implied and a different set of operators will be generated by the MAGIC/L compiler.

String Assignment Operators

Two assignment operators accept string operands. The destination operand for these assignments MUST be a CHAR array. DO NOT assign a value to a string literal.

```
:=      string assignment
+=      Concatenate
```

Both operators accept two operands as follows:

```
<dest> := <src>          move <src> to <dest>
<dest> += <src>          concatenate <src> to <dest>
```

In addition, the unary assignment operator CLEAR may be used to create a null string.

```
CLEAR  <dest>            make <dest> a "null string"
```

The following sequence of string assignments is legal:

```
CHAR MYSTR ( 2Ø )        ; declare the string
MYSTR := "Hello "        ; move in a string literal
MYSTR += "there "        ; concatenate to the string
MYSTR += "folks"         ; Now has - Hello there folks
CLEAR MYSTR              ; revert to a null string
```

The following line is NOT legal:

```
"Hello " += "there"
```

String Relational Operators

Relational operators may be applied to strings. These operators perform a byte by byte comparison of two strings. If the strings are unequal, the result is equivalent to an INTEGER comparison of the first pair of unequal bytes. Note that if one string is a substring of the other, the substring has the smaller value.

```
str1 <  str2
str1 >  str2
str1 <= str2
str1 >= str2
str1 == str2
str1 <> str2
```

Some examples of string comparisons:

relation	result
"AB" == "cd"	Ø (the comparison is case sensitive)
"CD" < "cd"	-1 (Upper case is smaller than lower case)

"a" < "ab" -1 (The substring is smaller)

In addition, a string can be tested to see if it contains any characters (i.e. to see if it is a "null string").

```
strl ==Ø
strl <>Ø
```

The other comparisons to zero (<Ø, >=Ø, etc.) are not defined for strings.

STRINGs are also supported by the PRINT and INPUT statements (chapters 6 and 7) and a wide variety of library routine (see appendix A).

MAGIC/L does not formally support pointer variables. Some routines, however, require a pointer to data rather than the data itself. The POKE routine, for instance, takes two arguments, a data value and an address in which to put that value (see Appendix A):

```
POKE ( <value> , <address> )
```

MAGIC/L allows for ADDRESS variables to be used as storage for addresses. ADDRESSes may be used as pointers (i.e. The value of an ADDRESS variable may be treated as an address). For example, suppose the ADDRESS variable AZ contains the address of the zero-th element of the INTEGER array Z. Then the statement:

```
POKE ( some expression , AZ )
```

is functionally identical to:

```
Z ( Ø ) := some expression
```

You may be wondering how we assigned the address of the array Z to the variable AZ. The answer to that is the "&" operator. The "&" operator takes as an argument a variable or array element, and evaluates to the address of its argument. The syntax of the & operator is:

```
& <variable-name>
```

or

```
& <array-name> ( <subscript> )
```

where <subscript> is any INTEGER expression. Thus the assignment to AZ was made like this:

```
AZ := & Z ( Ø )
```

Had we wanted to point AZ to Z (3) we could have written:

```
AZ := & Z ( 3 )
```

Note that the argument to & must be the name of a variable (scalar or array element). & has no effect on other arguments. Expressions should NOT be used within the & construct (except as a subscript expression). Nesting & operators has no meaning; That is: & & X is the same as & X. (The & operator may also take the name of a user-defined record as an argument. This usage of & is described in chapter 8.)

The & construct has the data-type ADDRESS. The & operator is also used in subroutine calling sequences when it is desired to pass arguments by reference (by default arguments are passed by value in MAGIC/L, see chapter 4.) For example, the POKE could have been:

```
POKE ( some expression , & Z ( Ø ) )
```

An exceptional case arises here. Any un-subscripted reference to an array name ALWAYS evaluates to the address of the zero-th element of the array. [The motivation for this will become clear in the next section] Thus the assignment of the address of Z (Ø) could have been written:

```
AZ := Z
```

and, in fact, the POKE could have been:

```
POKE ( some expression , Z )
```

This form is NOT recommended since the & directive serves to document what is actually happening. The unsubscripted reference does not save space as it generates the same code as & Z (Ø).

Subroutine and Function calls

Expressions may be used in the argument lists of subroutine and function calls. The only distinction between subroutines and functions is that functions have an associated data-type and value and may be used within expressions.

The simplest calling sequence consists of only the name of the routine (i.e. the routine requires no arguments). The routine DATE (which prints today's date on the terminal) is an example of such a routine. The calling sequence for routines which require arguments has the following form:

```
<routine-name> ( <expr> [, <expr> ...] )
```

where <routine-name> is the name of the subroutine or function, and <expr> is an expression which is the argument.

Examples of subroutine and function calls:

```
(1)     DISP ( Z , X * 3 )
(2)     A := MIN ( X , Y ) + 3
```

In example (1), the routine DISP was called with two arguments (DISP is described in Appendix A). Two of the arguments are expressions. Commas are used to separate the multiple arguments.

In example (2), the INTEGER function MIN was used within an expression within an assignment statement (MIN is described in Appendix A). Note that the only difference between subroutine calls and function calls is contextual (i.e. it is not necessary to use the keyword CALL).

Commands

Commands are a unique feature of the MAGIC/L language. Any subroutine (routine which does not return a value) may be defined by the programmer to be a command. This is an extremely powerful tool for creating a simple interactive environment for the end-user. The calling sequence for commands has the following form:

```
<command-name> <expr> [ , <expr>  ... ]
```

Observant readers will notice that the calling sequence is the same as that for subroutines except that the parentheses are missing. When working in an interactive environment, a syntax of the form:

```
COMMAND argument , argument ...
```

is most natural. To create a routine which supports such a syntax in MAGIC/L, you need only declare it a COMMAND when you define it. The details of this declaration are discussed in chapter 4.

There are two differences between subroutine calls and COMMAND statements. First, COMMAND statements are provided with the number of input argument words (should the programmer wish to make use of such information). This feature provides an added dimension to interactive programming. For example, COMMANDs can check for the right number of arguments, or take different actions depending on the existence of arguments.

Second, COMMANDs provide the facility for iterating through a list of input arguments.

The PARSED command is a variant form of the COMMAND. As with COMMANDS, PARSED commands do not require parenthesis. The arguments to a PARSED command are not arithmetic expressions,

however, they are strings. Since strings are normally the only possible argument to a PARSED command (see chapter 4 for the exception) quotes are not used. This allows for a "CLI" or "Shell" -like syntax of the form:

```
TYPE MYFILE
```

for file manipulation commands. Commas can be used to separate arguments to PARSED commands, but they serve no purpose and may be omitted.

Summary of Routine Types

To clarify the terminology surrounding subroutines and functions the following table is provided:

ROUTINE	Any of the following.
SUBROUTINE	A routine that does not return a value.
FUNCTION	A routine that does return a value.
COMMAND	A special form that does not use parenthesis.
PARSED command	A COMMAND that accepts string arguments.

Statement Delimiters Revisited

As we discussed in chapter 1, if multiple MAGIC/L statements are coded on a single line, they must be separated by the statement delimiter ";;". An exception to this rule exists in that certain constructs in MAGIC/L are implied statement delimiters. In these cases, the statement delimiter ";;" may (BUT NEED NOT) be omitted.

The constructs which are implied statement delimiters are:

1) COMMANDs
2) The Control Function Keywords:
   IF ELSE ENDIF WHILE REPEAT UNTIL FOREVER CASE
   DO UDO ITER UITER and LOOP
3) The PRINT and INPUT statements

As a general rule, the statement delimiter ";;" is not needed except when an assignment statement follows another MAGIC/L statement For example:

```
PRINT X  ;;  Y := X * 3
```

COMMAND words may act as a statement delimiter. This is syntactically possible since a command may not appear within another command or within an assignment statement. We recommend that the statement delimiter word ";;" be used between commands to enhance readability. Both of the following lines are legal (assuming XX is a command), but the second is much more readable.

```
XX 5 * 3  XX 7  XX 14 * ( X + 2 )

XX 5 * 3  ;;  XX 7  ;;  XX 14 * ( X + 2 )
```

Most control function keywords (ELSE ENDIF REPEAT FOREVER UNTIL and LOOP) act as statement delimiters. For example:

```
IF ( X == Y )  X := Y * 5  ELSE  Y := X * 5  ENDIF
```

Remember: WHEN IN DOUBT USE THE STATEMENT DELIMITER ";;"

[This page left blank]

MAGIC/L supports a variety of control structures. Often the user has the freedom to choose the syntax that best suits the algorithm used. All conditional control structures evaluate a "test expression". The test expression must have data-type INTEGER. The test expression is considered to be "true" if it has a non-zero value.

Be careful when using non-zero as true. Properly, a logical TRUE should be a -1, since only this value works correctly in conjunction with the bitwise operators (NOT, OR, etc.). For example,

```
( X AND Y )
```

will be TRUE only if X and Y share some common bits. A test for both being nonzero could be coded as:

```
( X ==Ø AND Y ==Ø )
```

IF-ENDIF

The IF-ENDIF structure allows for code to be executed only if the test expression is true:

```
IF ( test-expression )
   .. (t) ..
ENDIF
.. (c) ..
```

Code (t) is executed if the test-expression is true. Control continues with code (c).

Example - X is printed only if it is not zero

```
IF ( X <>Ø )
  PRINT X
ENDIF
```

Note that if X is an INTEGER variable, then this could have been written:

```
IF ( X )
  PRINT X
ENDIF
```

because the test-expression is a non-zero INTEGER in either case.

IF-ELSE-ENDIF

```
IF ( test-expression )
   .. (t) ..
ELSE
   .. (f) ..
ENDIF
 .. (c) ..
```

Code (t) is executed if the test-expression is true.
Code (f) is executed if the test-expression is false.
Control continues with code (c).

Example - print the maximum of X and Y

```
IF ( X > Y )
  PRINT X
ELSE
  PRINT Y
ENDIF
```

[Yes, there are simpler ways of doing this!]

WHILE-REPEAT

The WHILE-REPEAT structure provides a mechanism for repeating code depending on an initial test. The conditional code may not even be executed once if the test-expression is false on entry:

```
WHILE ( test-expression )
  .. (r) ..
REPEAT
.. (c) ..
```

Code (r) is repeated while the test-expression is true. The test-expression is re-evaluated each time code (r) is executed. Control continues with code (c).

Example - a non-recursive factorial function

```
VAL := some value
FACT := 1
WHILE ( VAL > 1 )
  FACT := FACT * VAL
  DECREMENT VAL
REPEAT
PRINT FACT
```

BEGIN-UNTIL

The BEGIN-UNTIL structure is another mechanism for looping. It differs from WHILE-REPEAT in that the conditional code is always executed at least once.

```
BEGIN
  .. (i) ..
UNTIL ( test-expression )
.. (c) ..
```

Code (i) is repeated until the test-expression is true. Note that code (i) is always executed at least once. Control continues with code (c).

Example - The outer loop of a game program

```
BEGIN
  RUNGAME
  PRINT "Do you wish to quit now? " , #Z
  INPUT ANSWER
UNTIL ( ANSWER == "yes" )
PRINT "I've had enough too!!!"
```

In this example the word RUNGAME is assumed to run the game [not surprising!]. After the game is run, the player is asked if he wants to quit. If not, the game is run again. If so, the smart-aleck computer says that it's tired too.

BEGIN-FOREVER

This is an unconditional loop. It was created when we got tired of typing UNTIL (Ø) at the keyboard.

```
BEGIN
  .. (r) ..
FOREVER
```

BEGIN-IF-REPEAT

The BEGIN-IF-REPEAT is the most general form of looping structure. It allows for a block of code which is always executed and another block that may never be executed.

```
BEGIN
  .. (i) ..
IF ( test-expression )
  .. (r) ..
REPEAT
.. (c) ..
```

Code (i) is executed. While the test-expression is true, then code (r) is executed and control passes back to the BEGIN causing (i) to be executed again. Control continues with code (c) when the test-expression evaluates false.

Notes on the BEGIN-IF-REPEAT structure:

- Code (i) is always executed one more time than code (r).
- If code (i) is non-existent then BEGIN-IF-REPEAT reduces to a WHILE-REPEAT structure.
- If code (r) is non-existent the BEGIN-IF-REPEAT is similar to BEGIN-UNTIL except that the exit is on false instead of true.

The BEGIN-IF-REPEAT is often used when the test condition of a WHILE-REPEAT cannot be stated as a simple expression. For instance:

Example - The outer loop of a game program

```
BEGIN
  PRINT "Do you wish to play a game? " , #Z
  INPUT ANSWER
IF ( ANSWER == "yes" )
  RUNGAME
REPEAT
```

Note that in this example, unlike the earlier example, the question is asked before the first game is played.

Although the BEGIN-IF-REPEAT structure is not often used, it is well worth learning because certain types of algorithms are very conveniently coded in this form.

CASE-ENDIF

The CASE-ENDIF structure provides for the runtime selection of an action routine to be executed. Each time the CASE statement is encountered, one of an ensemble of routines is executed depending on the value of an INTEGER expression. All of the routines within the ensemble must have identical calling sequences. If the routines accept arguments, the arguments are listed before the selection-expression in the CASE statement.

```
CASE ( arg1 , .. ,  expression )
   SUBØ
   SUB1
     .
     .
   SUBn-1
ENDIF
```

DO-LOOPs and ITER-LOOPs

DO-LOOPs are supported in MAGIC/L, but with a rather different syntax than other languages. The pieces of the DO-LOOP construct are:

1) The DO or ITER statement
2) The Body
3) The LOOP statement

The DO statement

The DO statement is a command statement. [Remember that a statement terminator is required if other statements appear on the same line.] The DO command takes two arguments which specify the number of times that the body of the loop will be executed, and the value of the loop counter on each iteration.

```
DO LL , UL
```

where:

LL is the Lower-Limit for the loop counter UL is the Upper-Limit for the loop counter

The loop counter is initially assigned the value of LL. The body of the loop is executed as long as the loop counter value is less than or equal to UL. This means that the body of the loop is not even executed once if LL is greater than UL. [N.B. this is different from FORTRAN - and more useful] Examples:

```
DO Ø , 1Ø          ; The loop counter runs from Ø to 1Ø
DO 5 , 5           ; The loop is run once (counter is 5)
DO 5 , 4           ; The loop is never entered
DO -3 , 16         ; The loop counter runs from -3 to 16
```

The ITER statement

Iteration loops may also be invoked using the ITER statement. The ITER statement take a single argument which determines the number of times the body of the loop will be executed. The statement:

```
ITER n
```

is completely equivalent to (but more efficient than) the statement:

```
DO Ø , n - 1
```

The ITER-LOOP is useful for looping through arrays since array subscripting begins at ZERO (e.g. a 1Ø element array is subscripted Ø through 9). In further discussion the term DO-LOOP will refer to loops initiated with either a DO statement or an ITER statement.

The Body and LOOP counters

The body of a DO-LOOP may be any legal MAGIC/L statements (including other DO-LOOPS). There are some words defined in MAGIC/L which are for use only within the body of a DO-LOOP.

I J and K	are internal DO-LOOP counters. They count up from LL to UL.
I' J' and K'	are internal DO-LOOP counters. They count down from UL to LL.
EXIT	forces an exit at the next LOOP statement.

Since MAGIC/L maintains internal DO-LOOP counters, it is not necessary to specify a counter variable in the DO statement. This feature takes a bit of getting used to, but it soon will seem quite natural (and convenient!).

The counters I J and K have a meaning that depends on context. Using the counter I always refers to the count for the most recent unclosed DO statement. The simplest case is a single DO-LOOP. In this case the counter I is used. For example to loop through an array, a DO-LOOP might have the following form:

```
ITER AASIZE
  Z ( I ) := some expression
LOOP
```

The counters J and K are used in situations where DO-LOOPs are nested. J and K refer to the count for the second and third most recent unclosed DO-LOOPs respectively. The following example illustrates the use of counters within nested DO-LOOPs:

```
(d1)    ITER AASIZE
(1)       Z ( I ) := Ø
(d2)      ITER 3
(2)         Z ( J ) := Z ( J ) + FUNCT ( I , J )
(12)      LOOP
(3)       PRINT Z ( I )
(11)    LOOP
```

In line (1), the counter "I" refers to the DO statement (d1). Thus all array elements between Z (Ø) and Z (AASIZE - 1) are referenced (and set to zero).

Line (2) is a bit more complicated. In this statement, "I" refers to the ITER statement (d2) and "J" refers to the next most recent ITER statement which is (d1).

In line (3), The use of counter "I" in the PRINT statement also refers to (d1) since it is the most recent unclosed ITER statement because the ITER statement (d2) is closed by the LOOP statement (12) before line (3).

The fact that only three levels of counters may be referenced does not imply that DO-LOOPs cannot be nested more than three deep. In the current implementation DO-LOOPs may be nested ten deep. While it may be disconcerting at first that a value known as "I" in one line of code suddenly becomes known as "J" a few lines further down, the user will quickly appreciate the lack of ambiguity that results. [In practice, the counter "K" is almost never used regardless of the depth of nesting]

The words I' J' and K' are counters which run from the high limit down to the low limit and provide for "reverse" DO-LOOPs. These counters may be used interchangeably with the counters I J and K. In the following example the values of I and I' are printed for each iteration of the DO-LOOP:

```
DO 2 , 6
  PRINT I , I'
LOOP
     2      6
     3      5
     4      4
     5      3
     6      2
```

The LOOP statement

The LOOP statement may take two forms:

```
        LOOP
or
        LOOP ( increment-expression )
```

In the first instance, where LOOP is not followed by an expression in parentheses, the LOOP counter is incremented by one.

If an increment-expression is specified, the expression is evaluated on each iteration. The expression must have INTEGER data-type. The result of the expression is added to the LOOP counter value. For example the DO-LOOP:

```
        DO 2 , 7
          PRINT I
        LOOP ( 2 )
```

will result in the values 2, 4, and 6 being printed. The increment expression need not be a constant. In fact, it can evaluate to a different value on each iteration of the LOOP.

After the LOOP counter is incremented, it is compared to the upper limit specified in the DO statement. If the LOOP counter value is greater than the upper limit, the control passes to the first statement following the LOOP statement. Otherwise the LOOP continues with the first statement following the corresponding DO statement.

UDO and UITER - Unsigned DO-LOOPs

The DO and ITER statements treat arguments as signed 16-bit INTEGERs (which range from -32768 to 32767). Thus DO-LOOPs with exclusively negative loop counter values are possible:

```
        DO -1Ø , -5
          PRINT I
        LOOP
```

will output:

```
        -1Ø
         -9
         -8
         -7
         -6
         -5
```

It is sometimes desirable to treat the DO-LOOP counter as an unsigned integer. In this case, use the UDO statement instead

of the DO statement:

```
UDO  32000 , 33000      ; This will loop
  .
LOOP

DO   32000 , 33000      ; This will not loop
  .
LOOP
```

The second statement will exit immediately since it is actually:

```
DO   32000 , -32536
  .
LOOP
```

The UITER statement allows for loops which iterate more than 32767 times.

EXIT - Exiting from DO-LOOPs

Exiting from within the range of DO-LOOP is accomplished with the word EXIT. The execution of EXIT does not immediately exit the loop, instead it causes a termination of the DO-LOOP the next time that the LOOP statement is encountered. All statements between the EXIT statement and the LOOP statement are executed a final time, but the values of the counters I and I' are undefined. [So be careful not to use them!] For example:

```
DO LL , UL
  .. code (a) ..
  IF ( relational-expression ) EXIT ENDIF
  .. code (b) ..
LOOP
```

In the previous example code (b) will be executed even if the relational expression is true and the DO-LOOP is terminated pre-maturely. If you wish to prevent (b) from being executed if the LOOP is to be terminated, you can use the IF-ELSE-ENDIF structure:

```
DO LL , UL
  .. code (a) ..
  IF ( relational-expression )
    EXIT
  ELSE
    .. code (b) ..
  ENDIF
LOOP
```

In addition, the RETURN statement, described in the next chapter, may be used within the range of a DO-LOOP. The RETURN statement causes an unconditional exit of the routine being executed.

Programs in MAGIC/L are constructed from building blocks. The two major building blocks are variables (variables have already been discussed in chapter 1) and routines. Most programs in MAGIC/L are built of many small routines rather than a few large ones. The types of routines which can be written in MAGIC/L include subroutines (which act on input arguments), functions (which return values), and commands (which are subroutines with a modified calling sequence). MAGIC/L also allows for routines which neither accept nor return values.

The combination of routines and variables into a MAGIC/L program is described in chapter 5.

Syntax Overview

The syntax of a MAGIC/L routine definition may be divided into five different sections:

1. The DEFINE statement
2. The input argument declarations
3. The local variable declarations
4. The code section
5. The END statement.

Only the DEFINE and END statements are required. The other sections are optional. A schematic of a routine definition follows. Sections enclosed in square brackets are optional, lower case text is descriptive, upper case text indicates a key-word which is required by the syntax:

```
DEFINE routine-name  [routine-type]
        [ input declaration section ]
  [ LOCAL
        local variable declaration section  ]
  [     .. code section ..
                ..              ]
        END
```

The DEFINE Statement

The purpose of the DEFINE statement is to associate a name with a block of code. In addition, a routine may be assigned a data-type (such as an INTEGER function) or be designated a special type of routine (e.g. COMMAND). The optional routine type specifier is discussed in more detail below.

We are now in a position to write the simplest type of routine; one which neither accepts nor returns arguments. In this case, the routine initializes the value of a global variable:

X must have been declared

```
; INITX - Routine to initialize the value of X to 2
DEFINE INITX
   X := 2
END
```

The key-word DEFINE tells the compiler that the following string is to be the name of a new routine which comprises other MAGIC/L routines. The key-word END tells the compiler that the definition is complete. If INITX is executed later (either by typing it at the keyboard, or by executing another routine that calls it), X will be set to 2.

Input Argument Declarations

Most subroutine calls include a list of arguments to the subroutine. When writing a subroutine, it is necessary to know the number and type of the input arguments. For example, in the call:

```
FILL ( AA , AASIZE , 5 )
```

Each element of the INTEGER array AA of length AASIZE words is to be filled with the INTEGER value 5. (FILL is described in appendix A). The routine FILL then accepts three arguments; an INTEGER array (an address), an INTEGER size, and an INTEGER fill value. The FILL routine could be written as follows:

```
DEFINE FILL
        INTEGER IARRAY ( 1 ) ISIZE IVAL
  ITER ISIZE
    IARRAY ( I ) := IVAL
  LOOP
END
```

The argument declaration section directly follows the DEFINE statement. The input arguments must be declared in the order which follows the expected calling sequence.

Since all of the arguments in this example are of the same data-type, they were declared on a single line. The argument

section may continue over several lines. The argument declaration section extends until either a LOCAL statement or the first line of code (a line without a variable declaration). Thus the same routine could have been written with a little documentation added:

```
; FILL  - Routine to fill an array with a value
DEFINE FILL
        INTEGER IARRAY ( 1 )    ; Array address
        INTEGER ISIZE           ; # of words to fill
        INTEGER IVAL            ; Fill value
  ITER ISIZE
    IARRAY ( I ) := IVAL
  LOOP
END
```

[The indentation conventions used in these examples are personal taste. The tab really serves to separate the declarations from the code.]

All names declared as input arguments are local names. This means that input argument names may NOT be referenced from the keyboard or from other routines.

The specification of the input array with size of one may be confusing. The size may be declared as any positive number and indicates that the input argument is an address that will be referenced as an array. The array size will be disregarded by the compiler. A size of one is often used to imply that the size is unknown at compile time and will be determined at runtime (in this case by passing another argument). In cases where the size is known, we reccomend that it be specified to enhance readability.

Input arguments are Values (not pointers.. usually)

In most cases, argument passing is "by value". Passing arguments "by value" implies that a subroutine has access to only the argument value, not to the memory location containing that value. This means that attempting to modify an input argument will change only the value of the argument as the subroutine sees it, not the value of the variable named in the subroutine calling sequence. For example, here is a naive attempt to double a number:

```
DEFINE IDBL
        INTEGER IVAL    ; Input value
  IVAL := IVAL * 2
END
```

This routine WILL NOT have any effect whatsoever. A call to this routine of the form:

```
IDBL ( X )
```

where X is an INTEGER variable, will not affect the variable X since only the current value of X is passed to the routine. [How can the variable X be modified if the called routine doesn't know the location of the variable?]

It is sometimes useful to be able to pass a pointer to a variable (or array) instead of the value. This is called passing "by reference." The procedure for passing arguments "by reference" in MAGIC/L is described in the following section.

Argument passing "by reference"

As we have seen in a previous section, the address of a variable or an array element may be formed using the & operator. The & operator may be used within a subroutine calling sequence to pass an argument "by reference" instead of "by value":

```
FILL ( & AA ( Ø ) , AASIZE , 5 )
```

Remember that un-subscripted array names are always addresses. Thus in the calling sequence to FILL shown above:

```
FILL ( AA , AASIZE , 5 )
```

The array AA is passed "by reference" and the arguments AASIZE and 5 are passed by value.

Although the caller can easily pass an argument by reference using the & operator, there is no complementary operator for the subroutine to use. The problem is solved by declaring the input argument as an array (even though it is known there is only one value). The array declaration provides the "level of indirection" needed when arguments are passed by reference. Thus the IDBL routine can be written as though an array were the argument:

```
DEFINE IDBL
        INTEGER IVAL ( 1 )        ; Input pointer
  IVAL ( Ø ) := IVAL ( Ø ) * 2
END
```

IDBL must be called with a pointer to the variable and not the value of the variable:

```
IDBL ( & X )
```

Local Variable Declarations

Local variables and arrays may be defined for use within the scope of a single MAGIC/L routine definition. These local variables are declared following the input argument declaration section (if any). The beginning of the local variable declaration section is indicated by the LOCAL statement. Following the LOCAL statement, any number of variable and array definitions may be made. As with the input argument names, these names are accessible only within the routine being defined.

The size of the local variable area is limited only by the amount of available memory. The actual space for the local variables is allocated each time the routine is called. If a routine is recursive, a new local variable area is allocated on each recursion. The space is de-allocated on exit from the routine.

There is no initialization of the variables within the local area. Further, the values of local variables will not be maintained between calls to the routine. [They may by chance remain unchanged. Such assumptions often result in the well known programmer's lament "I just made one simple change ..."]

Here is an example of a routine which uses local storage. The local variables are used to store intermediate results before the final PRINT statement:

```
; PDIST - Print an approximation to SQRT ( X*X + Y*Y )
; The approximation is good to better than 15%
DEFINE PDIST
        INTEGER X Y
LOCAL
        INTEGER MAXVAL MINVAL
  MAXVAL := MAX ( ABS ( X ) , ABS ( Y ) )
  MINVAL := MIN ( ABS ( X ) , ABS ( Y ) )
  PRINT  MAXVAL + MINVAL / 2
END
```

In this case, each time PDIST is called two storage words are allocated (for MINVAL and MAXVAL). After the calculated value is printed, the two words are de-allocated. Note that no assumptions are made as to the initial values of MINVAL and MAXVAL.

Functions - Routines which Return Values

Functions are routines which return values and can be treated in expressions as data items. Functions in MAGIC/L have an associated data-type. A function can be declared to be any intrinsic data-type. The function type is specified in the DEFINE statement following the routine name.

A simple function is written below. It returns twice the absolute value of the input argument:

```
DEFINE POSX INTEGER
        INTEGER IVAL
  POSX := 2 * ABS ( IVAL )
END
```

In the DEFINE statement, the word INTEGER declares the routine POSX as an INTEGER function. In addition, the name POSX is used within the code section to indicate the name of the "variable" to receive the returned value. [This is the same way as Fortran and Pascal does it.]

Another example of a function is the CELSIUS function defined in Chapter Ø and repeated here:

```
DEFINE CELSIUS INTEGER
        INTEGER FARENHEIT
  CELSIUS := ( FARENHEIT - 32 ) * 5 / 9
END
```

Now let's see an example of everything we've covered in this section. If we change the PDIST function above to return the value instead of printing it, we have a routine which has input arguments, local variables, and returns a value!

```
DEFINE DIST INTEGER
        INTEGER X Y
LOCAL
        INTEGER MAXVAL MINVAL
  MAXVAL := MAX ( ABS ( X ) , ABS ( Y ) )
  MINVAL := MIN ( ABS ( X ) , ABS ( Y ) )
  DIST := MAXVAL + MINVAL / 2
END
```

Of course if DIST is defined, then PDIST becomes:

```
DEFINE PDIST
        INTEGER X Y
  PRINT DIST ( X , Y )
END
```

Now we're starting to see why MAGIC/L programs are built of many short routines.

COMMANDs - A special calling sequence

MAGIC/L is designed to support the programming of interactive routines simply and easily. The COMMAND type declaration allows the programmer to specify that a routine is called in a command statement instead of within parentheses. For example the routine PDIST which prints a value would ordinarily be invoked like this:

```
PDIST ( 5 , 7 )
```

This sort of calling sequence is fine for a source code module, but when working at the keyboard it is much easier to type:

```
PDIST 5 7
```

This change in calling sequence (both sequences work) is accomplished simply by changing the DEFINE statement for PDIST to read:

```
DEFINE PDIST COMMAND
   .
   .
END
```

COMMANDs have one important limitation: they may not be declared as functions and thus cannot return an argument.

PARSED Commands

The PARSED word type provides a built-in facility for accepting string arguments. This facility allows for extremely simple coding of string oriented commands. For example, a "type-a-file" routine that requires a string argument (the quoted string is functionally equivalent to a CHAR array containing a string):

```
TYPE ( "MYFILE.MG" )
```

can serve as the primitive for a more natural form that is a PARSED command:

```
DEFINE TY PARSED
        ADDRESS FILENAME
  TYPE ( FILENAME )
END
```

which is used simply as:

```
TY MYFILE.MG
```

Although PARSED commands were designed for use highest level,

it is occasionally useful to pass string arguments to a PARSE command (especially when a primitive like TYPE is not available). The NOPARSE directive tells the compiler to treat the following PARSED command as a normal COMMAND. This allows for:

```
NOPARSE TY "MYFILE.MG"
```

A more useful example is a special type routine that appends a ".MG" to each argument:

```
DEFINE MTY PARSED
        CHAR FILENAME ( 40. )
  LOCAL
        CHAR TEMPFILE ( 40. )
  TEMPFILE := FILENAME   ; move string to a temp
  TEMPFILE += ".MG"      ; concatenate the .MG
  NOPARSE TY TEMPFILE    ; invoke a PARSED command
END
```

Handling a Variable Number of Arguments

Routines which are defined as COMMAND or PARSED type are automatically provided with a count of the number of argument words. The number of argument words (longs and real count as two) that follow the command name are placed in the variable CMDCNT at runtime. This count is quite useful for writing routines in which you wish to:

1) Verify the argument count
2) Take different actions depending on the argument count
3) Iterate through a variable length list of arguments

To illustrate the argument count verification, we will re-code the TY routine above to verify that exactly one argument was specified. The word ADRSIZE is a parameter that indicates the number of words per address (1 on 16-bit addressing machines, 2 on 32-bit machines).

```
DEFINE TY PARSED
        ADDRESS FILENAME
  IF ( CMDCNT <> ADRSIZE )      ; address size on 16-bit machines
    PRINT "INCLUDE requires exactly one argument"
    ABORT
  ENDIF
  TYPE ( FILENAME )
END
```

[The ABORT routine aborts further execution and returns control to the keyboard for input]

Since the IF...ENDIF clause used above occurs often, a word,

commands can set globals but do not return values

CHKCNT, has been defined. The four lines of the IF clause could be written:

```
CHKCNT ( ADRSIZE )
```

If CMDCNT is not equal to one address, a "Wrong Number of Operands" error message will be given, followed by an ABORT.

You can easily write a routine which takes different actions depending on the number of arguments:

```
INTEGER $ZOOM

DEFINE ZOOM  COMMAND
        INTEGER ZFAC
  IF ( CMDCNT ==Ø )
    PRINT "Zoom Factor: " , $ZOOM
  ELSE
    $ZOOM := ZFAC
  ENDIF
END
```

In the above example, the current zoom factor is maintained in the global variable $ZOOM. The ZOOM command displays the current zoom factor if no arguments are specified, and otherwise modifies the zoom factor. Thus:

```
ZOOM 8
```

sets the zoom factor to 8, while:

```
ZOOM
```

would display the message:

```
Zoom factor: 8
```

The third major use for the CMDCNT word is to allow processing of a list of input arguments. In this case, a single version of the input argument is specified, and the routine NXTARG is used to step through the argument list. The following example prints the sum of a variable number of arguments:

```
DEFINE SUM COMMAND
        INTEGER TERM     ; Input is a list of INTEGERs
  LOCAL
        INTEGER PARTIAL
  CLEAR PARTIAL
  ITER  CMDCNT
    PARTIAL += TERM
    NXTARG
  LOOP
  PRINT "The sum is: " , PARTIAL
END
```

A typical call to sum might be:

```
SUM 5 , 3 * 4 , 7
The sum is: 24
```

Remember that the CMDCNT word contains the number of argument words, not the number of arguments. Thus if the arguments are all LONG values (which are two words long each), CMDCNT is twice the number of arguments. For example, we modify the above SUM routine to sum LONG integers:

```
DEFINE LSUM COMMAND
        LONG TERM          ; Input is a list of LONGs
  LOCAL
        LONG PARTIAL
  CLEAR PARTIAL
  ITER  CMDCNT / 2
    PARTIAL += TERM
    NXTARG
  LOOP
  PRINT "The sum is: " , PARTIAL
END
```

Note that in the routine LSUM, it was necessary to specify the ITERation count as CMDCNT / 2.

A final comment on the NXTARG routine. NXTARG will by default point to the next argument in the input list. NXTARG knows from the input declaration section the word length of each argument in the list. If you wish to force a different word length for NXTARG, it can be specified as an argument to NXTARG. The main usage of this feature is to allow more than one scan through the argument list:

```
DEFINE FUNC
        INTEGER ARG
  ITER CMDCNT
    . process each argument
    NXTARG
  LOOP
  NXTARG ( - CMDCNT )    ; back up to first argument
  ITER CMDCNT
    . process each argument again
    NXTARG
  LOOP
END
```

This usage of NXTARG allows for some curious processing of arguments. NXTARG is usually used when more arguments were passed than declared. But what if fewer arguments are passed than declared? As we saw in the ZOOM example, it is legal in COMMANDs to declare arguments that are not passed. In a more general case there is no problem if TRAILING arguments are not

passed. Thus the test routine FARGS works with no special tricks:

```
DEFINE FARGS COMMAND
        INTEGER X Y
  IF ( CMDCNT == 1 )
    PRINT "First arg:" , X
  ENDIF
END
```

This works because the missing argument is Y, the trailing argument. If we wished to treat X as the missing argument, it would be coded as:

```
DEFINE LARGS COMMAND
        INTEGER X Y
  IF ( CMDCNT == 1 )
    NXTARG ( -1 )
    PRINT "Last arg:" , Y
  ENDIF
END
```

This is a rather odd command that prints the second of one argument!

Important Note - When there are missing arguments (in either of the two forms above) the "missing" variable must not be referenced. Any attempt to assign to the variable will damage MAGIC/L. This situation occurs when missing arguments imply default values and should be handled as follows:

```
DEFINE MYCOMM COMMAND
        INTEGER X
  LOCAL
        INTEGER TX
  IF ( CMDCNT == 1 )
    TX := X
  ELSE
    TX := <default value>
  ENDIF
  ... code using TX as value ...
END
```

RETURN - Abnormal returns from a Subroutine

Often the user will want to exit a subroutine in the middle of the code, instead of at the bottom. This may be accomplished with the RETURN statement. When the RETURN statement is encountered at run-time, control passes to the END statement without any other code being executed.

An example of the use of RETURN is the CLM function which determines whether a value is between two other values (see Appendix A):

```
DEFINE CLM INTEGER
        INTEGER IVAL
        INTEGER MINV MAXV
  SET CLM                   ; Initialize CLM
  IF ( IVAL < MINV ) CLEAR CLM RETURN  ENDIF
  IF ( IVAL > MAXV ) CLEAR CLM  ENDIF
END
```

In this case if the first condition is met, the second cannot be true. The RETURN exits immediately thereby avoiding executing a test which will always be false. [An ELSE clause would serve here, but the RETURN is more readable (and generates less code)]

Another use of the RETURN is to exit from within a control structure. A skeletal example of this usage follows:

```
; This routine returns a Ø if it found an abnormal
; condition; otherwise, it returns a -1
DEFINE SAMPLE INTEGER
  SET SAMPLE                ; assume a good condition
  DO ...
    .
    IF( BAD-CASE )
      CLEAR SAMPLE          ; found a bad case
      RETURN
    ENDIF
    .
  LOOP
END
```

Although we have stressed the interactive aspect of MAGIC/L, it is vital to be able to write a program source file outside of the MAGIC/L environment for later use with MAGIC/L. This is easily accomplished, since MAGIC/L will accept input from a text file as though it were typed on a keyboard.

A MAGIC/L source file is a text file which you create on your system using a text editor (or word processor) program. By convention, MAGIC/L source filenames end with the extension ".MG". The EXT routine, described below, initiates input from a specified source file. That act of compiling a file is often refered to as "loading a file" in this manual.

Although text in a MAGIC/L source file may be entered exactly as though it were being typed interactively, text in a source file will typically differ cosmetically from a keyboard dialog. In particular a source file will typically contain documentation aids such as comments (which are not normally typed interactively). Also, routines in a source file will typically be formatted with indentation for increased readability.

In addition to the routines that deal directly with source files, this chapter covers several other routines that are normally used in a program to set up the user environment. A large number of advanced environment oriented routines are described in chapter 12.

Compile-time and Run-time

One aspect of MAGIC/L, the distinction between compile-time and run-time, is often confusing to the beginning MAGIC/L programmer. In most languages, a source file is compiled into a binary form, then one or more binary files are linked into an executable file. The executable file can then be run. This 3 step compile-load-go sequence, which is tedious, is completely alien to MAGIC/L.

In contrast, MAGIC/L compiles each input line (whether from the terminal or a source file) and executes the compiled code before the next input line is read. The execution of the compiled code is delayed if any of the following conditions is true:

1) a control structure is un-closed
2) a DEFINE-END structure is un-closed
3) an up-arrow (^) appeared on the input line
   (indicating continuation onto the next line)

In these cases, each subsequent input line is incrementally compiled (i.e. the compiled code is appended to that of the previous line(s)). When none of the above conditions is true, the entire block of code is executed. Since the concept of an "incremental compiler" is new to most programmers, some more discussion is in order.

We have used many of the control structures interactively in earlier chapters [and hopefully you've tried out our examples]. When a control structure is encountered, for example:

```
WHILE ( X < 10 )
  .
  .
  INCREMENT X
REPEAT
```

none of the code between WHILE and REPEAT is executed until the matching REPEAT statement is compiled. Each line of code within the WHILE-REPEAT structure is incrementally compiled.

The DEFINE-END structure is another difficult concept. This structure can be considered just another control structure. After the DEFINE statement, all code is incrementally compiled until the matching END statement. When the END statement has been compiled, the entire block is not executed, it is saved under the specified name. One can view the saving process as the "execution" of the DEFINE-END structure.

For example, line of code:

```
X := Y * 6
```

is compiled and then the compiled code is executed (and discarded), whereas in the sequence:

```
DEFINE SXY
  X := Y * 6
END
```

The identical line of compiled code is not executed, but saved under the name SXY. A call to the routine SXY can then be compiled (and executed) at any time.

The purpose of this long-winded discussion is to point out a fundamental difference between MAGIC/L and most other compiled languages. In MAGIC/L, compilation occurs as each input line is read (incremental compilation). This is why new routines

may be defined interactively at the keyboard (not your typical "run-time" action), and why any previously defined routine may be executed while a source module is being loaded (not your typical "compile-time" action).

EXT - Loading a MAGIC/L source file

EXT is a parsed command which forms a MAGIC/L source filename and attempts to open that file for compiling. For example:

```
EXT MYFILE
```

will attempt to open MYFILE.MG. A message will be displayed with the file name and two numbers:

```
MYFILE          24123   21678
```

The first number indicates the memory address (in bytes, on most systems) where code will be placed. The second number indicates how much memory remains before compilation of the file. (The two numbers will not add up the the same value because MAGIC/L uses additional memory elsewhere to hold the symbol table.)

If the file is not found, an error message will be given following the memory display.

Compiling from a Utility Directory

It is useful to keep commonly used files in a directory separate from the working directory. MAGIC/L allows for a "utility" directory to be specified as a place to look if the file is not found in the current directory. On systems that do not have a directory structure, such as CP/M, a drive name is used instead of the directory name. (Refer to the system supplement.)

The utility directory is specified by assigning a string to the CHAR array XUTIL. That string is pre-pended onto the argument to EXT if the file is not found in the current directory. For example, in a UNIX-like directory structure, one might allocate a directory "/mutil" as a utility directory. The XUTIL array would be initialized by:

```
XUTIL := "/mutil/"
```

On CP/M, drive A: might be the utility drive:

```
XUTIL := "A:"
```

The EXT command above would first search for MYFILE.MG in the current directory. If it isn't found, EXT will then try

LOADing the file "/mutil/MYFILE" (or "A:MYFILE"). Failing this, an error is reported. If the file is not found, the memory display mentioned above will show the filename with the directory or drive name prepended.

Terminating a Source File

The source file is normally closed automatically with the end of the file is reached. It is wise to always have a carriage return on the last line since some systems have trouble with "partial" lines at the end of a file. In addition, all definitions or control structures must be complete when the file ends.

A "logical" end-of-file may be specified with an ENDFILE in the middle of a file. This allows for comments or unneeded code to be placed at the bottom of a file. Often debugging or benchmark code is place below an ENDFILE.

ENDFILE should not be used within a definition to signal EOF. (this usage is described in chapter 12.) It may, however, be used within a conditional to give a degree of "conditional compilation:"

```
IF ( DEBUG ==Ø ) ENDFILE ENDIF
```

Code placed below this statement will be compiled only if DEBUG is non-zero.

Additional Notes on EXT

The action of the EXT routine is to save the state of the current input file (channel number and current input line number) on a stack and to open the specified file as the new input file. Thus when the next input line is requested, it is read from the file just loaded. The loaded file remains the current input file until:

1) End-of-file
2) The word ENDFILE is executed
3) A fatal error is encountered
4) Another file is loaded with EXT

In the case of an error, all currently opened input files are closed, and input is returned to the keyboard. When an error is encountered, the line number containing the error is printed along with the error message. The line number is printed in decimal.

If another EXT is encountered, the state of the current input file is saved until that EXT is complete at which point the original state is restored and loading continues where it left

off. EXTs may be nested 8 levels deep.

Large programs are usually broken into many small files for editing convenience. In such a case, we reccommend that a "load" file be created. The load file will typically contain an EXT call for each of the modules in the program with a comment describing the contents of each module.

One last point about EXT needs to be emphasized. The EXT routine does NOT immediately begin reading the specified file, it just sets things up so that next line of input is taken from the specified file. Thus in code such as this:

```
EXT NEXTFILE ;; X := 1Ø
```

the variable X is set to 1Ø before a line is actually read from the file NEXTFILE.MG (although the EXT executes before X is set to 1Ø). EXT should be thought of as "open a file" rather than "compile a file."

Creating an executable disk image

The final stage in developing a program is the creation of an executable core image saved on disk. The SAVE command is used to perform this function. The SAVE command is invoked as follows:

```
SAVE <image-name>
```

where <image-name> is the name of the executable file you wish to create (without extension). The SAVE command will append the standard filename extension for an executable file to <image-name> and then create a file on disk which is an executable image of the currently running MAGIC/L program. Any previous version of that filename is deleted by the SAVE command. The file created by the SAVE command is immediately runnable without modification.

SAVE may be used at any time to create a core image of the currently running MAGIC/L. All variables will be saved with their most recently set values as well as all definitions, whether compiled from a file with EXT or entered at the keyboard. All files, however, are closed by SAVE and must be reopened when the program is later run. Normally, SAVEd programs are are built entirely from disk files so that the program can be reproduced later if modifications are needed.

Modifying the Prompt

The PROMSTR array allows the user to specify the prompt string to precede the standard prompt characters (limit of 15 characters). This string is used to specify the environment or program which is currently being run. For instance, if you wish to replace the "mgl" prompt with "ok" you would used the following dialog:

```
mgl> PROMSTR := "ok"
ok>
```

The angle bracket(s) will still be displayed.

Program startup

MAGIC/L provides a user definable startup routine. The is one of a set of user definable routines that will be described in chapter 12. The restart function is controlled by a variable, $RESTART, which contains the base address of some routine defined by the user. It is setup as follows:

```
DEFINE START
  PRINT "Welcome to my program"
END

$RESTART := BASE START
```

The special word BASE converts the word that follows into the address of the routine.

START could have been used to perform such one-time initialization tasks as opening files, selecting options, or pre-setting variables. The default restart routine is RSTØ, which prints the copyright message and the number of bytes available.

$RESTART is usually setup at the bottom of the "load" file, just before the SAVE command.

A Sample Program Module

The following example is a complete program. The file is LOADED and the routines are defined. The next to the last line sets up $RESTART so that on program restart the HELP routine is executed. $GO is unchanged so the standard MAGIC/L environment is active after the HELP routine completes. The last line in the file causes the current contents of memory to be copied into an executable file on disk. The SAVE routine causes an exit from MAGIC/L.

```
;  ------------------------------------------------------------
;   This   file   contains   a   utility   program   for   ;
Fahrenheit-Celsius conversion.

;  CELSIUS -  Convert Fahrenheit to Celsius
;        <temp in C> := CELCIUS ( <temp in F> )
;
DEFINE CELSIUS INTEGER
        INTEGER FAHRENHEIT
  CELSIUS := ( FAHRENHEIT - 32. ) * 5 / 9.
END

;  FAHRENHEIT -  Convert Celsius to Fahrenheit
;        <temp in C> := FAHRENHEIT ( <temp in F> )
;
DEFINE FAHRENHEIT INTEGER
        INTEGER CELSIUS
  FAHRENHEIT := 9. * CELSIUS / 5  +  32.
END

;  F2C - A command to print a Celsius temp. given
;         a Fahrenheit temp.
;        F2C <degrees-F>
;
DEFINE F2C COMMAND
        INTEGER\DEGF
  PRINT CELSIUS ( DEGF )
END

;  C2F - A command to print a Fahrenheit temp. given
;         a Celsius temp.
;        C2F <degrees-F>
;
DEFINE C2F COMMAND
        INTEGER DEGC
  PRINT FAHRENHEIT ( DEGC )
END

; HELP - Print some useful information
DEFINE HELP
  PRINT "Temperature Conversion Utility"
  PRINT "Commands:"
  PRINT "  C2F <deg-C>"
  PRINT "  F2C <deg-F>"
  PRINT "  HELP"
END

PROMSTR := "ctemp"       ; setup special PROMPT

$RESTART := BASE HELP    ; Give HELP on program initialization

SAVE CTEMPS              ; Create a disk image
```

After the source is created, it is compiled and run with the

following dialog (CP/M is assumed here):

```
A>MGL                                   run MAGIC/L
... copyright message ...
mgl> EXT CTEMPS                         compile file
CTEMPS          24123   21678           file is loaded
A>                                      SAVE returns to CP/M
A>CTEMPS                                run the new program
Temperature Conversion Utility          restart routine
Commands:
  C2F <deg-C>
  F2C <deg-F>
  HELP
ctemp>                                  new prompt
```

MAGIC/L supports a complete formatted output facility. Using the PRINT command, it is easy to format any desired output line. Integers can be converted as signed or unsigned values, and formatted into any width field either right or left justified. Reals may be formatted in either fixed point or scientific notation. Strings may be either right or left justified within a field. A columnar tabbing facility allows absolute or relative alignment of fields within the line. This chapter will describe each of these facilities in detail. The examples in this chapter are printed using the keyboard prompts.

The PRINT command

The PRINT command is used to provide formatted output in MAGIC/L. The arguments to the PRINT command are a list of items separated by commas. Each item is either an expression the value of which is to be converted to ASCII and output, or a format directive:

```
PRINT item [ , item ... ]
```

A simple example of the use of PRINT is to display the result of a calculation:

```
mgl> PRINT 5 * ( 7 + 3 )
      5Ø
mgl>
```

The default format for INTEGERs is right justified within an 8 character field. Remember that items must be separated by commas:

```
mgl> PRINT 5 * ( 7 + 3 ) , 6 * 6 * 6
      5Ø     216
mgl>
```

String Output from arrays (STR)

The PRINT statement can be used to output strings. Strings may be referenced in three ways in MAGIC/L; implicitly by entering a string literal, or by specifying an unsubscripted CHAR array, or explicitly by passing the address of a MAGIC/L string.

In the first two cases, MAGIC/L will know that the data type is STRING. (This is an internal data type, STRING variables may not be declared, they are implied by a string literal or unsubscripted CHAR arrays.) In these cases no special commands are needed:

```
mgl> CHAR TITLE ( 30 )
mgl> TITLE := "MAGIC/L Users Manual"
mgl> PRINT "This is the " , TITLE
This is the MAGIC/L Users Manual
```

In the third case, any address maybe used as a string pointer. This means that INTEGER arrays, LABELs, or ADDRESS variables can be used to point to a string to be output. In this case, the STR function must be used to indicate the mode of printing:

```
mgl> INTEGER TITLE ( 30 )
mgl> MVSTR ( "MAGIC/L Users Manual" , TITLE )
mgl> PRINT "This is the " , STR ( TITLE )
This is the MAGIC/L Users Manual
```

Note that the STR function (like the & operator) generates no code, it is used only at compile time to indicate the type. Often it is used in conjunction with the & operator to output a partial string:

```
mgl> CHAR TITLE ( 30 )
mgl> TITLE := "MAGIC/L Users Manual"
mgl> PRINT "This is the " , STR ( & TITLE ( 8 ) )
This is the Users' Manual
```

If STR were not used, the address of TITLE (8) would have been output as an integer. If neither STR nor & were used, the ASCII value of the contents of TITLE (8) would have been output.

Format Directives

Control over the output format is achieved using format directives. These mini-commands are used within a PRINT statement to specify the desired formatting. The format directives are listed below along with the default values (where applicable). Each of the directives is described in detail in the following sections. The action of all directives remains in effect for the rest of the current PRINT statement unless modified by another occurrence of the same directive.

directive	action
#A a	Insert the ASCII character value "a" into the output line
#C chn	Output the text to file opened on channel "chn"
#F n m	Set the field width and justification for REAL conversion (default n=12, m=2)
#I n	Set the field width and justification for integer conversion (default n=8)
#N	Do not output the formatted line (used for ENCODE or continuation)
#P a	Pad Character used to pad fields (default is space)
#R n	Specify conversion radix for integers (default is current RADIX value)
#S n	Specify field width and justification for string output
#T n	Tab function
#UI n	same as #I except unsigned conversion
#Z	Output without a carriage return

Integer Conversion (#I and #UI)

The format directives #I and #UI are used to specify the field width and justification for INTEGER and LONG values. The form of the directive is:

```
#I n
```

where n is an integer expression. The meaning of the argument value depends on whether it is positive, negative, or zero.

If n=Ø, only significant digits are placed into the output line. The field width is exactly the size required to display the integer value. For example:

```
mgl> PRINT 4 , 5 , #I Ø , 717 , 88
       4       571788
mgl>
```

The first two numbers were converted with the default field width of 8, the next two had a field width of zero.

A non-zero argument to #I specifies the field width for the number conversion. If n is positive, then the number is right-justified within the field. If n is negative then the integer is left-justified within a field width of ABS (n). Example:

```
mgl> PRINT "_" , #I 4 , 123 , "_" , #I -4 , 321 , "_"
_  123_321 _
mgl>
```

The previous PRINT command contains the following elements:

1. Output the string: "_"
2. Set right-justified, field width of 4
3. Convert and output the number 123
4. Output the string: "_"
5. Set left-justified, field width of 4
6. Convert and output the number 321
7. Output the string: "_"

If the number being converted has more significant digits than the field width provides for, the most significant portion of the number is placed into the field, and an asterisk is output as the right-most character. Example:

```
mgl> PRINT #I 4 , 12345
123*
mgl>
```

The Conversion Radix (#R)

The #R directive is used to specify the radix (number base) to be used for integer number conversion. By default, the current operating radix (the value of RADIX) is used. The #R directive is used to over-ride this default. This feature can be used for a quick number-base conversion, for example:

```
mgl> PRINT #R 8 , 1ØØ , 7 * 7
     144      61
mgl>
```

The #R directive only has meaning for integer number conversion.

REAL Conversion (#F)

The #F directive is used to specify the format of REAL (floating point numeric conversion. The form of the directive is:

```
#F n m
```

where n is the full field width, and m is the number of digits following the decimal point. The must NOT be a comma between the values of n and m. If m is positive then the conversion is in fixed-point notation. REAL numbers are always right-justified and displayed in decimal. For example:

```
mgl> PRINT #F 1Ø. 3 , 1ØØ.Ø / 3.Ø
    33.333
mgl>
```

If m is negative then conversion is in scientific notation. For example:

```
mgl> PRINT #F 12. -3 , 1ØØ.Ø / 3.Ø
   3.333E+Ø1
mgl>
```

The number of significant digits which are displayed is:

```
(n-m-1-s)       fixed point notation
(n-m-5-s)       scientific notation
```

where:

s=Ø if the value is positive s=1 if the value is negative

String formatting (#S)

The format directive #S is used to specify the field width and justification for string output. The form of the directive is:

```
#S n
```

where n is an integer expression. The meaning of the argument value depends on whether it is positive, negative, or zero.

If n=Ø, the string is placed into the output line with no padding. The field width is exactly the length of the string. (n=Ø is the default.) For example:

```
mgl> PRINT "tic-tac" , "-toe"
tic-tac-toe
mgl>
```

A non-zero argument to #S specifies the field width. If n is positive, then the string is right-justified within the field. If n is negative then the string is left- justified within a field width of ABS (n). For example:

```
mgl> PRINT #S -3 , "AA" , "BB" , #S 3 , "CC"
AA BB  CC
mgl>
```

The first two strings are left-justified within a 3 column field. The last string is right-justified within a 3 column field.

The Pad Character (#P)

The #P directive is used to change the current pad character. The default pad character is a space. The pad character is used to pad unfilled columns in a justified field. The pad character is also output into all columns skipped over by the tab (#T) directive (see below). For example, we could change the pad character in our last example to a period:

```
mgl> PRINT #P 56K , #S -3 , "AA" , "BB" , #S 3 , "CC"
AA.BB..CC
mgl>
```

In numeric output, the pad character is often used to provide leading zeros for octal number conversion:

```
mgl> PRINT #P 6ØK , #R 8 , #UI 6 , 1ØØ
ØØØ144
mgl>
```

Output an ASCII code (#A)

The #A directive is used to output an ASCII code. The form of the directive is:

```
#A cc
```

The byte value "cc" is appended to the output line. This directive is primarily used when the character to be inserted is not known at the time the PRINT statement is coded. For example:

```
DEFINE DCHAR
        INTEGER ICHAR
  PRINT "The character is: " , #A ICHAR
END
```

The routine DCHAR will append its argument value directly into the output line. DCHAR would be used like this:

```
mgl> DCHAR ( 1Ø2k )
The character is: B
mgl>
```

Columnar Tabbing (#T)

The #T directive is used to specify tabbing (or columnation) within the output line. The form if the #T directive is:

```
#T n
```

The action taken depends whether n is positive or negative. If n is positive, pad characters are appended until column "n" is reached. If n is negative then ABS (n) pad characters are appended to the output line.

Suppress Carriage-Return on Output (#Z)

If the #Z directive is used within a PRINT statement, the line will be output without a carriage return. A simple usage of #Z is to leave the cursor in the middle of a line while a task is performed:

```
PRINT "Please wait ..." , #Z
        .
        .
        .
PRINT " Complete"
```

The function YESNO defined below is a good illustration of the use of the #Z directive. The YESNO function accepts a query string as an input argument. The string is output as a yes/no

question, and the value of the function is true if a Y (or y) followed by a carriage-return was typed in response. The INPUT statement is described in the next chapter.

```
DEFINE YESNO INTEGER
        ADDRESS ISTRING
  LOCAL
        CHAR INBUF ( 1Ø. )
  PRINT STR ( ISTRING ) , "(Y/N)? " , #Z
  INPUT INBUF
  YESNO := ( INBUF ( Ø ) AND 137K ) == ASCII Y
END
```

The YESNO function might be used as follows:

```
IF ( YESNO ( "Do you want to exit MAGIC/L" ) )
  BYE
ENDIF
```

Suppress Output (#N)

The #N directive is used to suppress output of a partially formatted string. If the #N directive is used in a PRINT statement, the previously formatted text is left in the conversion buffer. This text may be appended to and/or printed by a subsequent PRINT command, or it may be transferred to an array using the ENCODE function (see below). One use of the #N directive is to build an output line within an iteration loop:

```
mgl> ITER 1Ø
mgl>>   PRINT #I Ø , I , #N
mgl>> LOOP
mgl> PRINT
Ø123456789
```

The final PRINT command forced the previous formatted line to be displayed.

A special case arises when using the #N directive. If the number of characters in the output buffer exceeds 132, they will be printed. Since #N if often used when printing is to be diverted away from the normal output channel, this will sometimes give erroneous results.

The ENCODE function

The ENCODE function transfers previously formatted data to a specified array (instead of outputting the string). The ENCODE function has the following form

```
ENCODE ( dest )
```

where "dest" is an array into which the formatted string will be moved. ENCODE is used after one or more PRINT commands which incorporate the #N directive. You must be sure that the destination array is large enough to contain the entire string. In addition to transferring the string, ENCODE clears the output buffer.

We can re-code the above example to save the formatted string in an array:

```
mgl> CHAR ISTRING ( 2Ø. )
mgl> ITER 1Ø
mgl>>   PRINT #I Ø , I , #N
mgl>> LOOP
mgl> ENCODE ( ISTRING )
```

The ENCODE function transfers the string into the array ISTRING instead of printing it. We can now print the string any time we like without having to re-format:

```
mgl> PRINT ISTRING
Ø123456789
```

The IFPRINT function

When the #N directive is used to build an output line, it is often desirable to force output when the line exceeds a certain length. This is accomplished with the IFPRINT routine:

```
IFPRINT ( <column> )
```

If the current column is greater than or equal to the argument, the line is output as though a PRINT were issued with no arguments. An IFPRINT (Ø) will always force output.

PRINT Command Primitives

There are three primitives which are used internally by the PRINT command but which may be useful for the user to access.

```
#COLUMN is the output column counter
#TYO    adds a single character to the output
#FIELD  is the primitive string justifier
```

The variable #COLUMN is maintained internally by the PRINT command. Its value is the number of characters currently in the output string. #COLUMN is reset when a carriage return is output or by an ENCODE. A typical use for #COLUMN is for multiple column output within a DO-LOOP. The #COLUMN variable can be used to initiate a carriage return:

```
mgl> ITER 1Ø
mgl>>   PRINT I * I , #N
mgl>>   IF ( #COLUMN >= 4Ø ) PRINT ENDIF
mgl>> LOOP
       Ø       1       4       9      16
       25      36      49      64      81
mgl>
```

This same task could have been performed by IFPRINT, but if the desired action is an ENCODE rather than PRINT, #COLUMN must be used.

The #TYO primitive is used by the #A directive of the PRINT command. The call:

```
#TYO ( value )
```

is functionally equivalent to:

```
PRINT #A value , #N
```

The #TYO primitive should only be used in cases where a single character is being added to the output string and printing is to be suppressed.

The #FIELD routine is used by the numeric and string output routines of the PRINT command. It is used to append a justified string to the output line. The #FIELD routine has the following form:

```
#FIELD ( BPTR , SLEN , FWIDTH )

BPTR    is a byte-pointer to the string to be output
SLEN    is the number of bytes to include
FWIDTH  is the width of the field into which the string
         is inserted.
        If FWIDTH is zero, the entire string is
         added to the output line.
        If FWIDTH is positive, the SLEN characters
         are right-justified within a FWIDTH
         character field.
        If FWIDTH is negative, the SLEN characters
         are left-justified within a ABS (FWIDTH)
         character field.
        If SLEN is larger than ABS (FWIDTH), only the
         first FWIDTH characters are included.
```

The #FIELD routine is described here primarily because there is one condition which cannot be handled by the PRINT command; output of sub-strings. The PRINT command can only output null-terminated strings, but the #FIELD routine may be used to output specified characters from within a string.

The date formatting routine #DATE is an example of the use of #FIELD to output a substring. The #DATE routine formats numeric values for month day and year into a string of the form dd-mmm-yy where "-mmm-" is an ASCII string describing the month:

```
CHAR DATSTR ( 4 * 12. + 2 )
DATSTR := "-JAN-FEB-MAR-APR-MAY-JUN-JUL-AUG-SEP-OCT-NOV-DEC-"

DEFINE #DATE
        INTEGER MONTH DAY YEAR
  PRINT #R 1Ø. , #P 6Øk , #I 2 , DAY , #N
  #FIELD ( & DATSTR ( 4 * ( MONTH - 1 ) ) , 5 , 5 )
  PRINT #R 1Ø. , #P 6Øk , #I 2 , YEAR , #N
END
```

The first line of code in #DATE formats the DAY in a two character field (in decimal) with ASCII zero as a pad character. The second line uses #FIELD to add a substring of the entire string defined above. The byte pointer within the string is determined by the number of the month. Finally YEAR is added to the string exactly as was DAY. Thus #DATE (6 , 5 , 79) will format the string "Ø5-JUN-79"

PRINTing to a File

An individual PRINT statement may send its output to any open file using the the #C print directive:

```
PRINT #C CHAN , ...
```

where CHAN is the channel number corresponding to an open file. (See appendix B for a descpription of opening and closing files.)

The #C channel is temporarily installed as the output channel for the duration the PRINT statement and thus must be specified with each PRINT. For clarity, #C should be the first item in the PRINT statement. If more than one #C is used, only the last will take effect. The #C directive is normally ignored in PRINT statements that end with a #N, but should be included if there is a possibility of overflowing the output buffer (see section on #N).

Diverting PRINTs to files

The #C directive allows individual PRINTs to be diverted to a file. Sometimes, however, it is useful to change the default output channel for a range of PRINT statements.

MAGIC/L provides a facility for diverting formatted output to disk files (or other devices). In fact, the output may be either diverted or duplicated while still being displayed on the terminal. All output from the PRINT statement is send to the file whose channel number is in OCH (unless overridden by #C). If the variable LOCH is other than -1, then the output is duplicated to that channel also. (APUSH and APOP, used below, are described in chapter 12.)

Example 1 - Diverting a PRINT to a file

Assuming a file is opened on channel FCHN, the output of a PRINT statement would be sent to that file as follows:

```
APUSH OCH          ; Save previous OCH
OCH := FCHN        ; Divert output to FCHN
PRINT  . . .
  .
  .
  .
APOP               ; Restore OCH
```

All output between the assignment to OCH and the APOP will be directed to the file opened on channel FCHN. Note: If this is to be done at the keyboard (instead of within a definition) then the above code should be imbedded in an ITER 1 ... LOOP structure to suppress execution until the entire section is

compiled. Otherwise all prompts will also be diverted to FCHN.

Example 2 - Duplicating PRINTed output to a file

Assuming a file is opened on channel FCHN, the output of a PRINT statement would be sent to that file as in addition to the current OCH file as follows:

```
APUSH LOCH        ; Save previous LOCH
LOCH := FCHN      ; Divert output to FCHN
PRINT  . . .
  .
  .
  .
APOP              ; Restore LOCH
```

All output between the assignment to LOCH and the APOP will be directed to the file opened on channel FCHN. The same restriction as in example 1 applies to keyboard invocation of this feature.

Limitations of Recursive PRINTs

It is possible to execute PRINT statements recursively by printing a function that contains a PRINT statement, as follows:

```
INTEGER X

DEFINE XX INTEGER
  XX := X
  PRINT X
END

PRINT XX
```

Although this will appear to work properly in simple cases, complications arise if the last PRINT statement uses print directives, i. e.:

```
PRINT #R 8. , #I 3 , XX
```

Although this specifies a new radix and field width, the function XX will reset these parameters to the default values before printing. This is because each usage of PRINT executes a "print init" before processing any values to be printed. This recursive use of PRINT should be avoided.

[This page left blank]

The INPUT statement is used to convert ASCII text into one or more items of binary or string data. The basic form if the INPUT statement is a list of variables to receive the converted data. INPUT will attempt to convert the ASCII text to the desired form and will prompt the user if there is either missing or illegal input text. More advanced error handling is also possible for applications which have more complex requirements. This chapter will first describe the syntax and usage of the INPUT statement. Next there is a discussion of how the INPUT statement interprets the text entered to it. Finally, the advanced error handling features of INPUT are described.

INPUT statement syntax

The INPUT statement consists of a list of items separated by commas. The commas are REQUIRED in the INPUT statement. The INPUT statement has one of the following forms:

INPUT from the keyboard:

```
INPUT item [ , item , ... ]
```

INPUT from a file:

```
INPUT #C chan , item [ , item , ... ]
```

When reading from a file, 'chan' is the INTEGER channel number of an opened file (the value returned by the OPEN call). Each 'item' in the list is either an INPUT statement directive or the name of a previously declared variable or array element. Both numeric and string data items can be read using the INPUT statement. The type of conversion for each input item is implied by the item specification as follows:

Numeric Input
- A scalar variable of any data type
- An array element of any data type

String Input
- An unsubscripted CHAR array
- Any other ADDRESS using the STR directive

Example 1: Requesting Numeric Input

The following example illustrates how numeric data would be requested:

```
INTEGER X
LONG    Y ( 5 )

INPUT X , Y ( 2 )
```

The previous INPUT statement requests two numeric data items: an INTEGER value to be stored in the scalar variable X, and a LONG value to be stored in the array element Y (2).

Example 2: Requesting String Input

The following example illustrates how string data would be requested:

```
INTEGER ASTR ( 1Ø )
CHAR    BSTR ( 4Ø )
ADDRESS CSTR

INPUT STR ( ASTR ) , BSTR , STR ( CSTR )
```

The previous INPUT statement requests three string data items. The first into the INTEGER array ASTR, the second is placed into the CHAR array BSTR, and the third into memory beginning at the address specified by the ADDRESS variable CSTR. Note that the STR directive was not necessary for the unsubscripted CHAR array BSTR.

Although not often done, it is possible to input both numeric and string items in the same INPUT statement.

Prompting for INPUT

The INPUT command does not issue a prompt. In practice it is useful to prompt the user as to the data he/she is expected to enter. The recommended method for prompting the user is to precede the INPUT statement by a PRINT statement (using the #Z directive to suppress a carriage return). For example:

```
PRINT "Input an integer and a long:  " , #Z
INPUT X , Y ( 2 )
```

will cause the specified prompt string to be displayed on the screen before the user is required to respond.

Responding to an INPUT statement

The response to an INPUT statement is a list of numbers and/or strings separated by spaces (and optionally commas). The input statement will attempt to convert the numeric responses into the requested data type. String data is accepted directly and is subject to the standard MAGIC/L interpretation as described below.

The INPUT statement uses the standard MAGIC/L RDLINE (see chapter 12.) function. If a special form of RDLINE is in use (e.g. a special line editing RDLINE) then the features of that RDLINE function are available to the user when entering data. The following two sections describe how the INPUT statement interprets numeric and string input data.

Conversion of Numeric Input

In general, numeric data which is entered to the INPUT statement is interpreted exactly as in a MAGIC/L program. However, the INPUT statement will make an attempt to convert the data type implied by the user response into the data type required by the INPUT statement. The following conversion are made automatically by the INPUT statement:

LONG input does not require the "L" specifier. (e.g. the response of 1234 to a LONG is interpreted as 1234L if a LONG was expected)

REAL input does not require a decimal point or fractional part to be specified. (e.g. the response 1234 will be interpreted as 1234.Ø if a REAL was expected)

Fixed point data types (CHAR, INTEGER, and LONG) will be converted in the current RADIX. As in MAGIC/L coding, the radix can be specified explicitly using a trailing ".", "K" or "X". An addition, the #R directive (described below) may be used to locally change the default RADIX that the input statement will use for conversion. REAL data is always interpreted in decimal.

Conversion of String Input

The request for string input data is satisfied by a word terminated by a space (or tab). If the string to be input is to contain spaces, it is necessary to surround the string in quotation marks ("). The quotation marks are stripped from the string and are not passed through as part of the input data.

In addition, special characters may be input by surrounding the octal character code within angle brackets. The conversion of these codes follows the same rules as string literals. Following are several examples of a legal response to a request for string input:

RESPONSE	INTERPRETED AS
Hello	Hello
"Hello"	Hello
<11Ø>ello	Hello
"Hello there"	Hello there

Note that the INPUT facility does not allow for entry of entire lines of text uninterpreted. To accomplish this form of input, the programmer should use the RDLINE function directly. The following is an example of an ACCEPT routine which simply reads a line of text into a specified buffer:

```
DEFINE ACCEPT
        CHAR BUFFER ( 1 )          ; destination
  IF ( RDLINE ==Ø )                ; if EOF
    PRINT "Unexpected EOF"         ; handle EOF
  ENDIF
  BUFFER := LBUF                   ; move string to dest
END
```

The ACCEPT routine might be used where a user comment (which can contain a variable number of words of text) is solicited. For example:

```
CHAR  USERCOMMENT ( 8Ø. )

PRINT "Enter comment:"
ACCEPT ( USERCOMMENT )
```

This method is most useful for keyboard input, where the EOF condition is unlikely, and therefore need not be checked. Because of the large number of variations possible in the requirements of a routine like ACCEPT above, and because of the simplicity of creating such a routine tailored to a particular application, no such routine is built into MAGIC/L.

INPUT Statement Directives

There are two directives that may be used with the INPUT statement. The first, The #E, is used to suppress error processing within the INPUT statement. second, #R, is used to specify a particular input conversion radix for one or more input variables. These directives are discussed fully in the following sections.

#E Suppress INPUT Errors

The #E directive is used to suppress error processing within the INPUT statement. If #E is specified, the INPUT statement is terminated and the user can perform error processing. The status word INPSTAT contains pertinent information about the exit condition. Error handling is described in more detail below.

#R Specify the default INPUT RADIX

In some instances, it is desirable to request numeric input data in a special radix. For example, memory locations and device vectors are often specified in octal on minicomputers and in hex on microcomputers.

To accommodate input of this type, the #R directive is used. It specifies that subsequent input values are to be interpreted in the specified radix. The form of the #R directive is:

```
#R n
```

where 'n' is the desired radix. The following example requests the user to enter the address of a device vector as an octal value:

```
PRINT "Enter Device Vector in Octal " , #Z
INPUT #R 8. , DVECTOR
```

The #R directive remains in effect for all numeric variables (except REALs) following the #R specification until another #R is specified or all data has been input.

INPUT Error Processing

The INPUT statement provides great flexibility in processing errors in the input data stream. There are three different methods of handling errors, listed from the simplest to the most sophisticated:

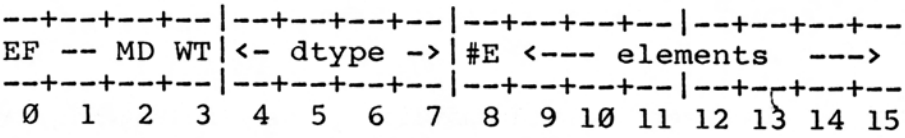
1. Use the default INPUT error handler. This method is the simplest in that it requires no additional programming.

2. Use the #E directive to exit on an error. Then check the input status word, INPSTAT, to determine what additional processing is necessary.

3. Write an error handling routine to be called by the INPUT statement when an error is encountered. This routine will also use the INPSTAT word to determine the nature of the error.

INPUT Error conditions

There are a number of error conditions which can occur during the execution of an INPUT statement. The input error status word INPSTAT contains all of the information necessary to evaluate and handle an INPUT error condition. This 16-bit word has the following form:



EF — End of File.
If set, the RDLINE encountered End-of-File.

MD — Missing Data.
If set, there were less items entered than were requested in the input statement. The total number of successfully converted elements is in the least significant 7 bits of INPSTAT

WT — Wrong Data Type
If set, the INPUT statement was not able to convert the input text into the requested data type. The requested data type is indicated in the 'dtype' field of INPSTAT. The total number of successfully converted elements is in the least significant 7 bits of INPSTAT. The input string which caused the WT error is in TBUF.

dtype — Most recent conversion type
This field represents the data type of the most recently attempted INPUT conversion. If the WT bit is set in INPSTAT, this indicates the requested data type of the item for which the conversion failed. The values in this field have the following meaning:
Ø INTEGER 1 LONG 2 REAL 3 CHAR 4 STRING

#E — The #E directive has been specified
If set, this bit indicates that #E has been specified in this INPUT statement. This field may be used by the INPUT error handling routine to modify its action. The default INPUT error handler will take no action if this bit is set.

A User written error handler may set this bit to indicate that an exit is to be taken from the INPUT statement.

elements — Total converted elements
This 7 bit field indicates the total number of input items that have been successfully converted. It may be used in conjunction with #E and the MD bit to determine the actual number of data items entered (allowing for default value processing).

There is one condition (not necessarily an error) which is not indicated by the INPSTAT word: extra input. If the user responds to a request for two elements with three data items, the INPUT command will ignore the third item. If extra input is of any consequence, it may be detected using the WORD (see chapter 12) function following the INPUT statement:

```
INPUT X , Y
WHILE ( WORD )
  PRINT "Extra Input Detected: " , TBUF
ENDIF
```

The Default Error Handler

The default INPUT error handler attempts to provide a user friendly environment suitable for many applications. If the #E directive has been specified, then the default error handler is not invoked and the INPUT statement terminates immediately. There are three error conditions which are handled by the default error handler:

1. Premature End of File
   If an End of File condition is encountered, it is considered unrecoverable. The error message "Unexpected

End of File" is given and an ABORT is taken. This error can often be avoided by including an EOF marker in the data stream. If that solution is not applicable and if EOF is anticipated, advanced error handling described below must be used.

2. Missing Input Data
   If the user types a carriage return before all of the requested elements are entered, MAGIC/L will respond with:

```
Missing Input
?
```

   The user may now input the additional data. Any previously input data will have already been processed and placed in the appropriate variables.

3. Wrong Data Type
   If the user enters data that cannot be interpreted as the appropriate data type, MAGIC/L will respond with a warning and the offending text:

```
Wrong type: <bad input>
?
```

   The user may now reenter the data. As above, any previously input data will have already been processed and placed in the appropriate variables.

Using the #E Directive

If the automatic error handling is not desired, the facility may be suppressed by placing a #E directive within the INPUT command:

```
INPUT #E , X , Y
```

If an error is detected during the input of X or Y, INPUT will exit immediately without giving an error message. The #E directive need not be the first item in the INPUT command. It will only apply to data items that follow it in the INPUT statement.

The existence of an input error may be determined by inspecting the variable SYSER. If SYSER is -1, the INPUT statement was successful. If there was an error, SYSER will contain the MAGIC/L error code for Input Error. The exact nature of the error is determined by inspecting the variable INPSTAT. Using INPSTAT, the programmer can determine the exact cause of the error and handle it appropriately. If the WT bit is set, indicating a conversion error, the string that could not be converted is in TBUF.

User-Written INPUT Error Handlers

When the INPUT statement encounters an error condition, it sets INPSTAT to indicate the nature of the error and then calls its error handling routine. The INPUT error handling routine, INPERR, is a user-definable function. The INTEGER variable $INPERR contains the execution vector of the error handler to be used.

The default handler, INPERRØ, is as simplistic as possible to save space. The code for INPERRØ is listed at the end of this chapter. If the processing performed by INPERRØ is not sufficient, the programmer is free to write a more complex error handler and install it as follows:

```
$INPERR := BASE MYINPERR
```

The user defined handler will be entered if any of the three errors EF, WT, or MD is encountered. The error handler can use INPSTAT to determine the appropriate action to take. The handler MUST be declared an INTEGER type function.

When the error handler returns to INPUT, the returned value is tested. If it is non-zero (TRUE) INPUT will perform another RDLINE and attempt to complete the requested input. If the returned value is zero (FALSE), INPUT will exit immediately.

Note that the handler must recognize the End of File error as a severe error or INPUT could become stuck in an infinite loop.

INPUT From a File

Thus far, this discussion has assumed that the input is coming from the keyboard. The INPUT facility may also be used to read data from a file, but there a few special considerations:

1. The channel number of the file to be read must be specified. (This of course means that the file must already be open.)

2. The default error handler is of minimal use when reading from a file, since a request for more data will simply read the subsequent line from the file.

3. Third, the End of File error must be anticipated or handled using #E or a user error routine. If EOF is not properly handled, the program will abort when the end of file is reached (Often, input data files will have a data value which indicates the end of data)

One final note: the "keyboard" form of the INPUT statement reads from the standard MAGIC/L input channel (ICH). The

input may therefore be redirected simply by changing ICH or using the LOAD facility (see RDLINE, LOAD, ENDFILE).

The following definition reads a specified data file consisting of one integer per line (up to 100 lines) into the array MYDATA.

```
DEFINE RDFILE
        ADDRESS INFILE
        INTEGER MYDATA ( 1 )              ; declare array
LOCAL
        INTEGER INPCHN
  INPCHN := OPEN ( INFILE , "R" )         ; open data file
  ITER 100
    INPUT #C INPCHN , #E , MYDATA ( I )   ; input one line
    IF ( SYSER <> -1 )  EXIT  ENDIF       ; exit on an error
  LOOP                                    ; else, back for more
  CLOSE ( INPCHN )                        ; close the file
  IF ( INPSTAT AND 0B000X )               ; check for bad data
    PRINT "ERROR ON INPUT"                ; tell somebody
  ENDIF
END
```

Listing of the INPERR0 Routine

The following is a listing of the default error handling routine used by the MAGIC/L INPUT statement. It can serve as an example for more complex user-coded error handlers.

```
DEFINE INPERR0 INTEGER
  IF ( INPSTAT <0 ) ERR ( E$UEOF ) ENDIF ; Unexpected EOF error
  IF ( INPSTAT AND 2000X )
    PRINT "Missing input"
  ELSE
    PRINT "Wrong type: " , TBUF
  ENDIF
  PRINT "? " , #Z
  SET INPERR0             ; Continue by reading more data
END
```

Chapter 8 Record Structures - User Defined Data Types

A record is a collection of one or more variables which have been grouped together for convenience. Each record is given a unique name called the record name. Once a record structure has been defined, the record name becomes a new data-type within MAGIC/L. Variables called record variables may be declared in the same way that intrinsic variables are declared. MAGIC/L does not currently support operators which act on record variables. MAGIC/L does however contain a special construct, SIZE (and SIZEW), to aid in record handling.

Records are a powerful tool in MAGIC/L. They are used to create arrays of mixed data-type, to organize complicated blocks of data (such as file headers), and to create multi-dimensioned arrays. This chapter will describe how records are defined, and how record variables are defined, activated, and used.

Defining Record Structures

Records are defined in a simple way. First a RECORD statement declares the name of the new data-type. Next the components of the record structure are listed, finally an ENDRECORD statement terminates the record structure declaration. For example:

```
RECORD DATE_REC
  INTEGER MONTH
  INTEGER DAY
  INTEGER YEAR
ENDRECORD
```

declares a new data type "DATE_REC" which consists of three integers. No memory is allocated at this time for data storage. The memory space for the INTEGER variables is allocated when a DATE_REC variable is declared.

Record structures may contain record variables as a component. For example, an employee information record may contain the date an employee was hired among its components. Such a record might be defined like this:

```
RECORD EMPL_REC
  INTEGER ID              ; Employee ID number
  INTEGER ENAME ( 20 )    ; Employee name string
  LONG SALARY             ; Current salary
  DATE_REC DAY_HIRED      ; Date hired
ENDRECORD
```

[You may have noticed that we are using a naming convention where record structure names end in "_REC". We have found this to be a useful documentation aid.]

Declaring Record Variables

Record variables are declared the same way as intrinsic data-type variables. Arrays of record variables may also be declared.

```
DATE_REC  TODAY BIRTHDAY
EMPL_REC  EMPLOYEES ( 100 )
```

In the first declaration the variables TODAY and BIRTHDAY are defined to have data-type "DATE_REC". Each of the two variables requires three words of memory. In the second, an array EMPLOYEES of data-type "EMPL_REC" is declared. The array declaration allocates 2600 words of memory (26 words per EMPL_REC variable).

Record variables differ from intrinsic variables in several ways. First, MAGIC/L does not support operators which act on record variables. (The only operator which might be useful would be the assignment operator.) For example it would be nice to be able to copy the data from the TODAY record variable to the BIRTHDAY record variable by coding:

```
BIRTHDAY := TODAY       ; Illegal statement
```

This is NOT a legal MAGIC/L statement. This action may easily be accomplished with the following statement:

```
MVWDS ( TODAY , BIRTHDAY , SIZEW DATE_REC )
```

The MVWDS routine accepts two addresses and a word count as arguments. When record variable names are used in MAGIC/L code, they always represent a pointer to the first word of the record variable storage area. The SIZEW construct is discussed below.

Since record variables are really collections of other variables, it is necessary to activate a record variable in order to access its component parts. A record variable can be activated globally using the WITH command. Global record activation can be over-ridden locally using the colon (:) operator.

WITH - Global Record Activation

The WITH command is used to activate one or more record variables. It is necessary because there is a natural ambiguity in referencing the components of a record variable. The syntax of the WITH statement is:

```
WITH <record-variable>
```

For example, in the statement:

```
YEAR := 1981
```

One may ask whether the variable YEAR is a part of the TODAY variable or a part of the BIRTHDAY variable. The answer is given by the WITH statement which specifies the active variable:

```
WITH TODAY
YEAR := 1981

WITH BIRTHDAY
YEAR := 1947
```

Only one variable of a given record structure may be active at a time. The WITH statement is a global statement overriding any previous WITH statements in other routines. A record variable activated by a WITH command remains activated until another WITH is executed.

When a record variable is a component of another record, the records must be activated in order. Each activation is an independent action. For example:

```
WITH EMPLOYEES ( 1Ø )
WITH DAY_HIRED
```

activates one of the record variables in the EMPLOYEES array. The second WITH statement activates the DATE_REC variable DAY_HIRED in that array element. If a different EMPLOYEES element is then activated:

```
WITH EMPLOYEES ( 11 )
```

the activation of DAY_HIRED is not changed. That is the variables DAY MONTH and YEAR still refer to the DATE_REC variable DAY_HIRED which is contained in the element EMPLOYEES (1Ø). It is necessary to again activate DAY_HIRED to access the one in EMPLOYEES (11). This may be done in a single command:

```
WITH EMPLOYEES ( 11 ) DAY_HIRED
```

In this form, the records are activated in the order in which

they are listed.

As a final example, if we wanted to list the ID number and date hired for all employees with salaries above $30,000 we could write the following routine:

```
ITER 100
  WITH EMPLOYEES ( I )
  IF ( SALARY > 30000L )
    WITH DAY_HIRED
    PRINT ID , SALARY , DAY , MONTH , YEAR
  ENDIF
LOOP
```

Colon (:) Operator - Local Record Activation

The operator ":" may be used to activate a particular record element locally (over-riding the current WITH). The syntax of this operation is:

```
<address> : <record-component>
```

Where <address> is the address of the record to be activated locally, and <record-component> is the name of the variable which is a component of that record. For example, in the DATE_REC record described above, there are three components; MONTH, DAY, and YEAR (all INTEGER variables).

In the previous section, the YEAR component of the TODAY record variable was initialized using the following sequence:

```
WITH TODAY
YEAR := 1981.
```

Using the ":" operator it is possible to initialize the YEAR component in the BIRTHDAY record without deactivating the "WITH TODAY" performed above:

```
BIRTHDAY : YEAR := 1947.
```

The two YEAR variables may be printed as follows:

```
PRINT TODAY : YEAR , BIRTHDAY : YEAR
```

or since TODAY has been activated:

```
PRINT YEAR , BIRTHDAY : YEAR
```

If a record variable is a component of a record structure, the colon operator will yield the address of the nested record variable. This address may be used on the left side of another colon operator to access a component of the nested record. For example the operation:

```
EMPLOYEES ( 5 ) : DAY_HIRED
```

evaluates to the address of the DAY_HIRED record variable contained in the record variable array element EMPLOYEES(5). In order to display the year that this employee was hired, we might use the following code:

```
PRINT  EMPLOYEES ( 5 ) : DAY_HIRED : YEAR
```

Multiple colon operators evaluate from left to right. The operand on the right side MUST be a record component. The operand on the left of the colon MUST be an address, however it need not be the name of a record variable Any address, such as an unsubscripted array name, may be the left-hand operand of a colon. This allows for arrays to be treated as record variables. For instance:

```
INTEGER DATEBUF ( 3 )
DATEBUF : YEAR := 19Ø3
```

will set DATEBUF (2) to 19Ø3.

The DUMMY and OFFSET commands

There are two commands which are used to reserve space with no associated name or type within a record structure. The OFFSET command specifies the offset in 16-bit words of the next data item to be declared within the record. If the OFFSET command is the last line in a record declaration, it specifies the size of the record. The DUMMY command reserves space relative to the current offset. Both commands require a single expression as an argument:

```
DUMMY  <expression>
```

where:

<expression> indicates how many 16-bit words to skip within the record structure.

```
OFFSET <expression>
```

where:

<expression> indicates the offset (in 16-bit words) from the beginning of the record of the next item to be declared.

```
BOFFSET <expression>
```

where:

<expression> indicates the offset (in 8-bit bytes) from the beginning of the record of the next item to be declared.

Suppose for example a file has an 8 word header, but we are only interested in the first word which is the file ID, and the last 3 words which are INTEGER values. We could define

such a record as follows:

```
RECORD HDR_REC
  INTEGER ID      ; First is the ID number
  DUMMY 4.        ; then skip 4 words
  INTEGER X       ; Here is the data we want
  INTEGER Y
  INTEGER Z
ENDRECORD
```

The size of a HDR_REC variable is 8 words but only 4 of them are accessible as named component variables.

The OFFSET command is used within the record structure definition in order to force a specific record length. For example, suppose that HDR_REC (as defined above) is actually 100 words long (instead of 8), but we are only interested in the variables already listed. We can modify the previous declaration like this:

```
RECORD HDR_REC
  INTEGER ID      ; First is the ID number
  DUMMY 4.        ; then skip 4 words
  INTEGER X       ; X is at byte offset 10.
  INTEGER Y
  INTEGER Z
  OFFSET 100.
ENDRECORD
```

Another use of OFFSET is to force a variable to a specific word offset within a record. Suppose that there is another word of interest at byte offset 66 from the beginning of the HDR_REC record. We can simply modify the HDR_REC declaration like this:

```
RECORD HDR_REC
  INTEGER ID
  OFFSET 5.       ; same as DUMMY 4 here
  INTEGER X       ; X is at byte offset 10.
  INTEGER Y
  INTEGER Z
  BOFFSET 66.
  INTEGER EXTRA
  OFFSET 100.
ENDRECORD
```

By using negative values, the DUMMY command may also be used to equivalence variables within a record. Although those who prefer strong type-checking will cringe at this capability, it is nevertheless useful. A major use of the equivalencing capability is for buffers which may receive data in several different formats.

```
RECORD EQUIV_REC
  INTEGER X Y              ; Two INTEGER variables
  DUMMY -2                 ; Back up 2 words
  REAL RX                  ; One REAL variable
  DUMMY -2                 ; Back up 2 words again
  INTEGER XV ( 2 )         ; A two word array
ENDRECORD
```

In this example, the data in a EQUIV_REC variable may be reference as two integers, an integer array, or a REAL variable.

The DUMMY command may also be used to simplify (and clarify) access to adjacent array elements. For example, a graphics application may require a set of line end-points. Such a record structure and array would be declared as follows:

```
RECORD XY_REC
  INTEGER X Y
ENDRECORD

XY_REC XYVEC ( 100 )
```

The array XYVEC is 200 words long (100 coordinate pairs). A little thought reveals that it is tedious to access two adjacent coordinate pairs in order to say draw a line between them. We might do something like this:

```
WITH XYVEC ( N )
LINE ( X , Y , XYVEC ( N + 1 ) : X , XYVEC ( N + 1 ) : Y )
```

Creative use of the DUMMY command significantly simplifies this process. The first two lines of the above code can be eliminated if the XY_REC is defined like this:

```
RECORD XY_REC
  INTEGER X Y
  INTEGER NEXTX NEXTY
  DUMMY -2
ENDRECORD

XY_REC XYVEC ( 100 )
```

Note that the size of XY_REC variables is still 2 words so that the array declaration does not change. This is an example of how proper structuring of data can produce shorter, faster, more readable code.

A final word of warning. Be careful when using the DUMMY command with negative values that the final size of the record structure is positive.

Forcing EVEN offsets

Because many machines have restrictions on odd byte addressing, it is often convenient to force the record offset of an element to be even. It is not legal to access an integer or long on an odd byte boundary on such machines. The .EVEN command may be used to force to next element in a record to fall at an even offset. If the offset is already even, there is no action taken. The following example shows a record with several CHAR variables followed by an INTEGER.

```
RECORD SAMPLE_REC
  CHAR A B C     ; a "random" number of chars
  .EVEN          ; force to even offset
  INTEGER XX     ;  so XX will be on word boundary
RECORD
```

The .EVEN command is also useful at the end of the record declaration to force the total size to be even. This may be desired if an array of odd size records is declared. Note that .EVEN only forces the offset within a record to be even, not the actual address.

It should be noted that MAGIC/L makes no attempt to prevent the user from odd byte word addressing. It is the programmer's responsibility to prevent this from occurring on machines that do not allow it. A special case of this is the LOCAL space of subroutines, which is actually a records structure. On many systems, the LOCAL variables must add up to an even number of bytes.

SIZE and SIZEW - the size of a record structure

MAGIC/L contains two special action words; SIZE and SIZEW. SIZE is used to determine the size in BYTES of a record structure and SIZEW determines the size in WORDS of a record structure. The syntax of the SIZE directive is:

```
SIZE  <record-structure-name>
SIZEW <record-structure-name>
```

The string following the word SIZE must be the name of a record structure, for example:

```
SIZE DATE_REC
```

This two word grouping is treated as an INTEGER value which is the current size of the record structure. Normally, the SIZE construct is used to parameterize a record size in I/O transfer calls such as WRS and RDS, and SIZEW is used to parameterize a record size in data movement operations such as MVWDS.

The Record Base Address

The address of the currently active record variable is maintained in a location called the "record base". Each record structure has a "record base" location. The address of the "record base" location for a given record structure is obtained using the & operator:

```
& <record-structure-name>
```

The value derived from this construct is the address of the memory location which contains the address of the currently active record variable. That is:

```
WITH TODAY
```

is functionally identical to

```
& DATE_REC := TODAY
```

although the former is more efficient.

Pointers to record variables

As we mentioned above, when the name of a record variable is used in an expression, it evaluates to the address of the first word of the variable. Thus record variables can be copied using the MVWDS routine in conjunction with the SIZE construct. For example:

```
MVWDS ( TODAY , BIRTHDAY , SIZEW DATE_REC )
```

Copies SIZEW DATE_REC words from the address of TODAY to the address of BIRTHDAY.

It is possible that you wish to know the address of the currently active record variable (which as we now know is stored in the "base" location). The name of a record structure when used within an expression, evaluates to the address of the currently active record. This property of record structures allows routines to be written without compile-time knowledge of the actual record variable to be processed. For example, suppose we want to write a routine which sets some DATE_REC variable to today's date, by copying the TODAY variable into it. This can be done like this:

```
DEFINE COPYDATE
  MVWDS ( TODAY , DATE_REC , SIZEW DATE_REC )
END
```

Now we can decide which DATE_REC variable will be modified at run-time. For example:

```
WITH BIRTHDAY  ;;  COPYDATE
```

Notice that the WITH command is terminated with the statement delimiter ";;" since COPYDATE appears on the same line.

There are many situations where it is necessary to have initialized data available to a program. In MAGIC/L there are several ways to initialize a data area. The most common form of data initialization is an assignment statement to explicitly initialize a data item following its declaration. MAGIC/L also has a set of initialization commands which allow the programmer to build any desired data structure. We shall describe both forms of initialization in this chapter.

Explicit Data Initialization

The most conventional method is to initialize each data element explicitly. An array may be initialized as follows:

Example 1: An array containing the number of days for each month

```
INTEGER DAMONTH ( 12. )

DAMONTH ( Ø ) := 31.
DAMONTH ( 1 ) := 28.
        .
        .
DAMONTH ( 11. ) := 31.
```

Since the 12 assignment statements in the above example are executed as the module is compiled, the DAMONTH array is apparently initialized at "compile-time". Although it may seem like a foreign concept to "execute" code at "compile-time", remember that all statements in MAGIC/L are compiled as they are entered and then executed when all control structures are matched. Basically since the assignment statements are not within a DEFINE-END structure, they require no space and simply execute as they are encountered - a "compile-time" initialization.

This ability of MAGIC/L to execute code as it is encountered allows for some powerful initialization capabilities. For example if a table is to be initialized with a function, a DO-LOOP can be used to initialize the table (and yet require no storage space in the program).

Example 2: Initialize a table with the function: x * (x + 1) + 1

```
INTEGER TABLE ( 100. )
ITER 100.
  TABLE ( I ) := I * ( I + 1 ) + 1
LOOP
```

Alternatively, if the function already existed in the program, it could have been used for the initialization:

```
DEFINE FX2 INTEGER
        INTEGER X
  FX2 := X * ( X + 1 ) + 1
END

INTEGER TABLE ( 100. )
ITER 100.
  TABLE ( I ) := FX2 ( I )
LOOP
```

Data Initialization Commands

In addition to explicit initialization of data elements, MAGIC/L allows for "unnamed" data structures to be built using a set of data initialization commands. These commands allocate and initialize memory as directed. The following is a list of commands which perform the allocation and initialization function.

```
.BYTE - Initialize data area with CHAR(s)
.WORD - Initialize data area with INTEGER(s)
.LONG - Initialize data area with LONG(s) or REAL(s)
.ADDR - Initialize data area with ADDRESS(es)
.TEXT - Initialize data area with a string
```

A formal description of these commands appears below in this chapter. The function of these commands is to place one or more data elements at the beginning of free memory, and update the pointer to free memory accordingly. The word "." (pronounced dot) returns the address of the next word of available memory. "Dot" is sometimes used to save a pointer to an "unnamed" data area (see example 3 below). A LABEL statement can also be used to locate an otherwise unnamed data area (a LABEL simply gives a name to the current value of "dot").

Since arrays are also allocated at the beginning of free memory, the data initialization commands can be used to "extend" an array. In particular, an array defined to be of size zero can be initialized using these commands. In example 3, the same initialization as example 1 is performed, this time using the .WORD command to allocate and initialize

memory.

Example 3: An array containing the number of days for each month

```
INTEGER DAMONTH ( Ø )
.WORD   31. , 28. , 31. , 3Ø. , 31. , 3Ø.
.WORD   31. , 31. , 3Ø. , 31. , 3Ø. , 31.
```

The DAMONTH array as created in example 3 is identical in all respects to the one created in example 1. In example 1, a 12 word array was allocated and then initialized. In example 3, a zero word array was allocated, then the following 12 words were allocated and initialized using the .WORD command.

One aspect of the method used in example 3, is that the exact length of the array being initialized is specified by the initialization code itself instead of within the array declaration. This feature can be useful when creating terminated lists or initialized string arrays. For example, if a CHAR array is to be allocated and initialized to a message, the method of example 3 frees the programmer from having to count the number of characters in the string. Example 4 shows how such a string might be allocated and initialized using the .TEXT command.

Example 4: Initialize a CHAR array

```
CHAR HIMSG ( Ø )
.TEXT   "Hello, Welcome to MAGIC/L"
```

Another common requirement is to initialize a number of strings where the strings are of variable length. In MAGIC/L it is possible to create such a structure while using only the amount of memory required for each string and without having to give a name to each string. In example 5, four strings are allocated and the ADDRESS array SPTRS contains pointers to each of the four strings. Note that in this example the word "dot" is used to initialize and element of SPTRS with a pointer to where the string allocated by the following .TEXT command is stored.

Example 5: Create an array which contains pointers to variable length strings.

```
ADDRESS SPTRS ( 4 )

SPTRS ( Ø ) := .
        .TEXT   "Shouldn't you be asleep?"
SPTRS ( 1 ) := .
        .TEXT   "Top 'o the mornin' to ya"
SPTRS ( 2 ) := .
        .TEXT   "Good Afternoon"
SPTRS ( 3 ) := .
        .TEXT   "Good Evening"
```

A routine which might access the SPTRS data structure might be:

```
DEFINE TIMEMSG
        INTEGER HOUR
  PRINT STR ( SPTRS ( ( HOUR + 1 ) / 6 ) )
END
```

Which will print an appropriate message for the periods midnight-6AM, 6AM-noon, noon-6PM, and 6PM to midnight.

The LABEL statement

The LABEL statement is used to give a symbolic name to a particular address in memory. The form of the LABEL statement is:

```
LABEL <name>
```

where <name> is any legal MAGIC/L name. The newly defined <name> is assigned the value "dot" and has data-type ADDRESS.

LABELs are most often used in assembly code, but they are also used to give a name to an initialized data structure. Example 4 above initialized a CHAR array with a string. A LABEL could have been used to identify the string in a similar way.

Example 6: Using LABEL to identify a string

```
LABEL HIMSG
.TEXT   "Hello, Welcome to MAGIC/L"
```

The difference between example 4 and example 6 is quite simple. The CHAR array requires two more word of memory than the LABEL in its definition. However, the CHAR array provides the ability to subscript into the string. In addition, the CHAR form can be PRINTed directly, but the LABEL form requires the STR function when printing (or when used with string operators) to specify type STRING. Both versions can be

output using the statement:

```
PRINT STR ( HIMSG )
```

In general, where a data structure is only to be referenced as a whole (i.e. subscripting of individual elements is not required), it is more efficient to use a LABEL than a zero-length array to identify the data structure.

Memory Allocation Commands

In addition to the data initialization commands above, MAGIC/L supports memory allocation commands. These commands allocate memory beginning at "dot" for the specified number of words. The allocated memory is initialized to zeros. The syntax of these commands is:

```
.BLKW   <count>
.BLKB   <count>
.BLKA   <count>
```

where <count> is the number of words, bytes, or addresses for which space is to be allocated. The argument <count> may be any INTEGER expression. Note that .BLKB will allocate an even number of bytes when used on machines that have restrictions on odd byte addressing. .BLKA is equivalent to .BLKW on 16-bit processors, and is equivalent to:

```
.BLKW   nwds * 2
```

on 32-bit processors.

Initialization Commands Description

.BYTE - Initialize data area with CHAR(s)

```
        .CHAR   <iexpr> [ , <iexpr> ... ]

        <iexpr>         is any INTEGER expression
```

The CHAR words are stored in memory in the order that they are specified in the argument list.

.WORD - Initialize data area with INTEGER(s)

```
        .WORD   <iexpr> [ , <iexpr> ... ]

        <iexpr>         is any INTEGER expression
```

The INTEGER words are stored in memory in the order that they are specified in the argument list.

.LONG - Initialize data area with LONG(s) or REAL(s)

```
        .LONG   <lexpr> [ , <lexpr> ... ]

        <lexpr>         is any LONG or REAL expression
```

The values are stored in memory in the order that they are specified in the argument list. LONG and REAL expressions may be mixed in a single .LONG command.

.ADDR - Initialize data area with ADDRESS(es)

```
        .ADDR   <addr> [ , <addr> ... ]

        <addr>          is any ADDRESS
```

The arguments to the .ADDR command are typically LABELs or pointers to data elements.

.TEXT - Initialize data area with a string

```
        .TEXT   <string>

        <string>        is any string
```

The .TEXT command takes only a single argument.

Chapter 10 Creating New Classes of Definitions

MAGIC/L provides the unique facility of defining new classes of words called an "action class." An action class is a set of definitions which execute common code, but have different parameters. The common code is called the "Action Routine" and the parameters are stored in the "Parameter Field." In fact, all definitions in MAGIC/L have an action routine and a parameter field; this chapter describes how to make and use new action routines.

Creating an Action Class

An action class is a set of MAGIC/L words which have the same action routine. The action routine performs some action on the data in the parameter field associated with a given name. For example, all INTEGERs are of the same class. The action routine for INTEGERs in an expression returns the value of the particular INTEGER accessed. The code to do this is the action routine and is common for all INTEGERs.

The action routine is what defines a new class of words. An action routine may be coded either in high level MAGIC/L or in assembly language. In either case, when a word of the new action class is called, the action routine for that action class is executed. A pointer to the parameter field of the word passed in as an argument to the action routine. If the action is in assembly language, the parameter field pointer is passed in the register designated as the "W" register (See the assembler supplement). If the action routine is in high level MAGIC/L, the pointer becomes the last argument to the action routine. As we shall see, other arguments may be passed in to the action routine.

A routine is specified to be an ACTION routine by declaring the routine-type ACTION on the DEFINE or ENTRY statement:

```
DEFINE MYACT ACTION
       or
ENTRY MYACT ACTION        (for assembly language)
```

The name which is declared as an ACTION is treated like a parameter. The code associated with the ACTION class is NOT directly executable. The name of the ACTION class should only be used in a BUILD call or a MAKE command (see below).

In addition, if other routine-type declarations (such as INTEGER or COMMAND) are made when the action routine is defined, those characteristics will be implied for all routines defined in the new action class.

For example, we can define an action class which will change the current radix (by modifying the variable RADIX). The MAGIC/L words DECIMAL and OCTAL are such words.

```
DEFINE SETRAD ACTION
        ADDRESS RAD
  RADIX := PEEK ( RAD )
END
```

Routines which are in the SETRAD action class all have the common action routine SETRAD. The action of SETRAD is transfer the first word of the routine's parameter field into the variable RADIX. Words of the SETRAD class require no arguments, thus the SETRAD definition declares a pointer to the parameter field as its only input argument. Note that Action Routines always take their last argument by "reference," thus requiring special code to handle the extra level of indirection. In this case the extra level is provided by PEEK. An alternate form of coding this is:

```
DEFINE SETRAD ACTION
        INTEGER RAD ( Ø )
  RADIX := RAD ( Ø )
END
```

Here the declaration of the input argument as an array (the size in ignored) triggered the extra level of indirection. This form is usually used when the parameter list contains more than one element.

Making Definitions of the New Class

Once the action routine is defined, any number of words can be created that use it. The routine BUILD is used to create a new definition of a specified action class. The BUILD routine is called as follows:

```
BUILD ( <entry-name> , <action-class> )
```

where <entry-name> is a string which specifies the name of the new entry. The new entry will be a part of <action-class>. Following the BUILD command, the parameter field of the new word should be constructed using the various data structure declarations available in MAGIC/L.

In the case of the SETRAD action class, the parameter field consists of a single INTEGER which specifies the RADIX. The words DECIMAL and OCTAL would be defined as follows:

```
BUILD ( 'DECIMAL , SETRAD )
.WORD   1Ø.

BUILD ( 'OCTAL , SETRAD )
.WORD   8.
```

The MAKE Command

Since most parameter fields consist of one or more INTEGER values. The COMMAND MAKE is used to create new definitions of an action class where the parameter field consists of only a list of INTEGER values. Since this is the most common usage of action classes, it was convenient to provide a facility to build new definitions in a single line of code:

```
MAKE <entry-name> , <action-class> , <ipØ> [, <ipl> ..]
```

The first two arguments are identical to the two arguments of the BUILD routine (in fact MAKE calls BUILD). Following the <action-class> specification, any number of INTEGER parameters can be specified. This is functionally equivalent to placing all of the <ip> arguments in a .WORD declaration on the following line:

```
MAKE <entry-name> , <action-class>
.WORD   <ipØ> [ , <ipl> ... ]
```

The MAKE command is only a convenience. Its function can be provided with the BUILD routine the .WORD declaration. The declaration of DECIMAL and OCTAL using the MAKE command would be:

```
MAKE    'DECIMAL        SETRAD  1Ø.
MAKE    'OCTAL          SETRAD   8.
```

Further Examples of Action Classes

As noted above, if the action routine has additional declarations on the DEFINE line, those attributes are passed on to all of the members of the action class. This next example defines a class of words that will set or display the contents of variables. This example also illustrates a method of using addresses (pointers) as parameters.

```
DEFINE SETARG ACTION COMMAND
        INTEGER VAL
        ADDRESS ARGPTR ( 1 )
  CMDCNT -= ADRSIZE          ; Subtract off the size of
                             ; the parameter field address
  IF ( CMDCNT == 1 )
    POKE ( VAL , ARGPTR ( Ø ) )
  ELSE
    IF ( CMDCNT ==Ø )
      NXTARG ( -1 )          ; Adjust the argument pointer
                             ; for the missing value
      PRINT "Current value:" , PEEK ( ARGPTR ( Ø ) )
    ELSE
      PRINT "This requires Ø or 1 arguments"
    ENDIF
  ENDIF
END
```

Words of this class would be made as follows:

```
INTEGER $ZOOM $SCALE

MAKE    'ZOOM   SETARG
.ADDR   & $ZOOM

MAKE    'SCALE  SETARG
.ADDR   & $SCALE
```

And used as follows:

```
mgl> ZOOM 5
mgl> ZOOM
Current value:       5
mgl> SCALE 3
mgl> SCALE
Current value:       3
mgl>
```

There are a number of interesting points here. First, because SETARG is defined as a COMMAND, all the words made with SETARG as the action will also be COMMANDs. Although when called, these routines will normally have zero or one argument, the action routine will see an extra ADDRESS argument, the pointer to the parameter field. Thus the code subtracts ADRSIZE (the number of words per ADDRESS unit) from CMDCNT and then tests for zero or one argument word.

The final obscure point is the NXTARG (-1). Normally, NXTARG is used to sequence forward through an argument list, that is, it shifts the definition of the arguments so that the second argument word appears to be the first. In the case of SETARG, however, if the first argument is not present we need to shift in the other direction. The rule to follow in such odd cases is: "When there are more arguments passed than were

declared, use a positive NXTARG; when there fewer arguments passed than declared, use a negative NXTARG." This is probably the most complicated example in this manual; don't worry if you don't fully understand it at first.

The examples thus far have used the parameter field to hold a value and a pointer. It is also possible to put the execution vector of a routine into the parameter field. This is very useful when there are similar stretches of code with the difference being the function of the inner loop. In the following example the action routine scans a set of data and performs a function on each element of the data.

```
INTEGER ARR ( 1ØØ )          ; DATA ARRAY

DEFINE SCAN ACTION           ; SCANNING ACTION ROUTINE
        INTEGER FUNC ( 1 )
  ITER 1ØØ
    EXEC ( I , ARR ( I ) , FUNC ( Ø ) )
  LOOP
END
```

The SCAN action routine calls the function whose execution vector is in the parameter field with two arguments, the subscript and the data value. Let's define a pair of functions which might be called from the SCAN action routine (remember these functions are passed two INTEGER arguments):

```
DEFINE QDISP                 ; QUICK DISPLAY FUNCTION
        INTEGER II VAL
  PRINT VAL , #Z
END

DEFINE SDISP                 ; SLOW DISPLAY FUNCTION
        INTEGER II VAL
  PRINT "Element" , II , "  is" , VAL
END

MAKE    'QD     SCAN    BASE QDISP
MAKE    'SD     SCAN    BASE SDISP
```

This example was made rather simplistic, but more complicated functions could have been made. The EXEC call in the action routine will pass the first two arguments into the function being executed. Even though QDISP does not used the counter, II, The argument still must be declared since it is passed in by SCAN.

Efficiency of Action Classes

The efficiency of Action classes must be separated into two categories; space efficiency and time efficiency. In terms of space efficiency, it is usually space efficient to declare an action class even if only two definitions exist within the class. Each member of an action class requires only 1 word overhead plus the size of the parameter field, and the action routine typically requires only a few words overhead.

For example, consider defining the radix routines from above as follows:

```
DEFINE STRAD
        INTEGER RVAL
  RADIX := RVAL
END

DEFINE DECIMAL
  STRAD ( 10. )
END

DEFINE OCTAL
  STRAD ( 8. )
END
```

In this case, STRAD requires 4 words less overhead than does the action routine SETRAD, however the definitions DECIMAL and OCTAL using STRAD require 5 words while the counterparts using MAKE require only 2 words, saving 3 words each time a new member of the action class SETRAD is created. In more typical cases, ACTION classes can save hundreds of words or more by allowing dozens of similar routines to be defined in one page of code.

In terms of execution speed, high level action routines typically require an extra level of indirection (the PEEK in SETRAD) in order to access the data in the parameter field, thus high level ACTIONs may execute sightly slower than a subroutine call counterpart. Assembly level ACTION routines will typically be just as efficient as their ENTRY counterparts since the parameter field pointer is directly passed in a register.

The Vocabulary, or Symbol Table, of MAGIC/L contains entries for every word defined in the language. When a word is compiled, the vocabulary must be searched for the word. In order to separate the vocabulary into groups of words that have common application, MAGIC/L allows for dividing the symbol table into sub-vocabularies, called Branches.

Branches defined in MAGIC/L

Each branch is named, and the name is actually an entry in another branch. Three branches are defined in the basic MAGIC/L. MG/L is the main branch, containing most of the words documented in the MAGIC/L Users Manual, and, by default, will contain the entries created by the user. BTEMP is a special branch that contains dummy entries for the arguments and locals used by a sub-routine while it is being defined. This branch is normally invisible to the user. The assembly mnemonics and directives are in a third branch whose name will depend on the assembler being used. (See the system supplement for the name of the assembler branch.)

Defining a New Branch

Branches are created with the branch statement, as follows:

```
BRANCH ( <branch_name> )
```

where <branch_name> is a string which will name the branch. For example, a new branch called MYBRANCH would be created with the following statement:

```
BRANCH ( 'MYBRANCH )
```

The name MYBRANCH will be used for illustrative purposes throughout this chapter.

Two independent actions may be specified for branches:

- search the branch
- add new names to the branch

Up to eight branches may be active for searching at one time, but only one branch, called the CURRENT branch may receive new

names. Remember that these actions are independent; if the new names are to be found, the CURRENT branch must also be active for searching.

Activating a Branch for Searching

At any time, up to eight branches may be active for searching. A branch is activated simply by invoking the name of the branch. This causes a pointer to the branch to be pushed on a special stack called the vocabulary stack, or V-stack. The branches on the V-stack will be searched in the reverse order of the stack, that is, the last branch pushed will be the first branch searched. The V-stack is popped by the .END statement.

The LOOKUP routine is used by the MAGIC/L compiler (see chapter 12) to search through the active branches until the specified string is found.

The following shows how a branch would be activated and de-activated:

```
MYBRANCH        ; push branch on V-Stack
  .
  .             ; MYBRANCH is the first to be searched
  .             ; in this section
  .
.END            ; pop branch from V-Stack
                ; words in MYBRANCH are no longer found
```

The V-stack is not cleared on an ABORT condition. This means, for instance, that if an error is made while coding assembly language at the keyboard, the assembler will still be active.

The current size of the V-stack is maintained in the INTEGER variable .V which will contain an offset from the base of the V-stack to the last entry pushed (i.e. if .V is zero, only one branch is active). The base of the V-stack is contained in the ADDRESS variable .VØ and the address of the top of the V-stack is given by:

```
<top of V-stack> := ADX ( .VØ , .V )
```

The INTEGER variable .Vl contains the V-stack overflow offset. The value of .V must remain between zero and .Vl or an error occurs.

Searching a Single Branch

The routine $LOOKUPØ is used to scan a single vocabulary branch for an entry. The $LOOKUPØ routine is called by LOOKUP for each branch on the V-stack and is of course user-callable. The calling sequence for $LOOKUPØ is:

```
<?found> := $LOOKUPØ ( <branch> , <entry-name> )
```

where <branch> is searched for <entry-name>. The returned value <?found> is true if <entry-name> is found and false otherwise. If the <entry-name> is found, other global variables are also initialized by the $LOOKUPØ routine. These variables are described with the LOOKUP routine in chapter 12. The following is an example of a routine which returns true if the input string is part of MYBRANCH and false otherwise:

```
DEFINE ?MINE INTEGER
        ADDRESS INAME
  ?MINE := $LOOKUPØ ( MYBRANCH , INAME )
END
```

Note that in the example above MYBRANCH is used within an expression. Branch names within an expression evaluate to an INTEGER pointer to the branch.

Adding New Definitions to a Branch

Although several branches may be active for searches at one time, only one branch may be active for receiving new entries at a given time. This branch is called the CURRENT branch, and a pointer to this branch is maintained in the INTEGER variable CURRENT. When a branch name is used within an expression, a pointer to the branch is returned. Therefore, the following line of code will set the current branch to MYBRANCH:

```
CURRENT := MYBRANCH
```

Definitions created after this line will be entered into the MYBRANCH branch. This will remain until CURRENT is changed, for instance by the line:

```
CURRENT := MG/L
```

which will change CURRENT back the the default branch. If other branches have been created, the user must be careful to restore the proper branch. This is best insured with the APUSH and APOP routines:

```
MYBRANCH                    ; Activate MYBRANCH for searching
APUSH CURRENT               ; save current CURRENT
CURRENT := MYBRANCH         ; set new CURRENT
  .
  code entered into MYBRANCH
  .
APOP                        ; restore old CURRENT
.END                        ; De-activate MYBRANCH for search
```

This method insures that CURRENT will be restore to the proper branch, even if an ABORT occurs while compiling the code.

Note that CURRENT and the V-stack are completely independent (i.e. MYBRANCH was pushed onto the V-stack AND it was specified as the CURRENT branch.) The CURRENT branch need not be activated for searching (but usually is).

Why Use Branches?

There are several reasons to use the branch facility in MAGIC/L. First, compile time is reduced if the LOOKUP is restricted to a smaller number of entries. Also, conflicts in names are reduced by separating the entries into different branches.

A third use of branches is to create a limited environment for the user. This is done by creating a branch that contains only the definitions required for a certain interactive environment and the setting up the V-stack so that only this branch is searched by LOOKUP. This could be done with MYBRANCH as follows:

```
POKE ( MYBRANCH , .VØ )  ;;  CLEAR .V
```

These two statements must be on a single line (once the first statement is executed, the words CLEAR and .V may not be found by LOOKUP). It is often useful when creating such limited environments to place a command into the limited branch which re-activates the default branch. For example:

```
APUSH CURRENT
CURRENT := MYBRANCH

DEFINE M_RESTORE
  POKE ( MG/L , .VØ )
  CLEAR .V
END

APOP
```

The routine M_RESTORE will be in the MYBRANCH branch and will restore the MG/L branch when invoked.

Branches may also be used to prevent the use of words until the environment is set up for them. A data base system, for instance, may have record access routines in a branch that is only active when the data file is opened.

Notes about Branches

If .END is invoked when only one branch is active for searching, a Vocabulary Stack Underflow error will be given.

If .END or the branch names are used within a definition, their action occurs WHEN THAT DEFINITION IS EXECUTED.

A branch need not be active to execute words from the branch if they have been compiled into other words. The branch facility is only used during compilation, when the symbol table is searched.

[This page left blank]

Chapter 12 Advanced Programming Features

This chapter describes a variety of advanced features of MAGIC/L that are useful in writing complex applications. Most of the features listed here will not be needed by the novice MAGIC/L programmer. The features listed here, however, can greatly simplify programming in MAGIC/L. Topics covered are:

- Execution vectors (pointers to routines)
- Execution of routines
- Changing the default radix
- Saving and restoring variables
- Using MAGIC/L compiler routines
- User definable startup and abort handling
- Programming for 16/32 bit compatibility

Several example programs are included at the end of this chapter.

BASE - The execution vector of a MAGIC/L Routine

The BASE construct is used to form the execution vector of a MAGIC/L routine. This two-word construct is declared as follows:

```
BASE <MAGIC/L-routine-name>
```

The BASE construct is treated as an INTEGER value. The value may be passed as an argument to the EXEC routine (see below). The BASE construct is used below (see $RESTART and $ABORT) to specify user-defined routines.

In most 16-bit implementations of MAGIC/L, the execution vector is the address of the beginning of the routine (execution address). In other implementations, the execution vector is an offset into a table of execution addresses. The function VTBL should be used to convert the execution vector into the execution address.

```
<execution-address> := VTBL ( <execution-vector> )
```

The function VTBL returns data-type ADDRESS. The execution address of a MAGIC/L routine is not used in MAGIC/L programming. It can however be useful for example in determining the memory requirements of a particular routine.

EXEC - Execute a MAGIC/L routine

The EXEC routine causes the routine whose execution vector is specified to be executed. The calling sequence is:

```
EXEC ( execution-vector )
```

By using the EXEC routine, it is possible to specify an action at runtime. The following example shows how the EXEC routine might be used:

```
INTEGER $MESSAGE

DEFINE MESSAGE
  EXEC ( $MESSAGE )
END

DEFINE AM-MESS
  PRINT "Welcome to MAGIC/L"
END

DEFINE PM-MESS
  PRINT "Good afternoon"
END

$MESSAGE := BASE AM-MESS
```

Above we have defined a "generic" message routine called MESSAGE. Whenever MESSAGE is executed, the routine whose execution-vector is in the variable $MESSAGE is executed. The last line of the example places the execution-vector of AM-MESS into message. Thus whenever MESSAGE is executed, the string "Welcome to MAGIC/L" is printed. It is possible to change the action of MESSAGE at a later time. All we need to do is define a new routine and place its execution-vector into $MESSAGE:

```
DEFINE MESS1
  PRINT "Do you get the message?"
END

$MESSAGE := BASE MESS1
```

It is useful to note that the routine MESSAGE is defined before either of the actual message routines. In effect, the routine EXEC provides a facility for forward referencing by using back-links.

Sometimes it is useful to initialize the variable the will be executed with a routine that does nothing. The routine NO_OP performs this, as efficiently as possible. NO_OP is also useful to effectively disable a user definable function, such as those listed below.

Inportant Note: EXEC can only be used with functions and subroutines. It will NOT work properly with COMMANDs, PARSED commands, operators, ACTION routines, variables or records.

Recursion

Some problems are best solved by having a routine call itself, or recursion. Although MAGIC/L does not allow a routine to call itself by name, routines may call themselves through the RECURSE routine. RECURSE is a special routine that takes on the properties of the routine currently being defined. A factorial function (the traditional example of recursion) may be defined using RECURSE:

```
DEFINE FACTORIAL INTEGER
       INTEGER VAL
  IF ( VAL > 1 )
    FACTORIAL := RECURSE ( VAL - 1 ) * VAL
  ELSE
    FACTORIAL := 1
  ENDIF
END
```

In this case RECURSE acted as an INTEGER function of one argument.

Several restrictions apply to the usage of RECURSE. First, as with any recursive situation, the routine must be limited to the available memory and stack space. Note that the conditional in the factorial example prevents the recursion from "running away." Since the MAGIC/L stacks are of rather limited size, recursion should be limited to about twenty levels.

The RECURSE routine may not be used with COMMANDS, PARSED commands or ACTION routines. This is essentially the same restriction that applies to the EXEC routine.

One final note: Although recursion is a significant property in the theory of computation, and a popular topic in computer science, it is usually the worst way of solving a problem. The factorial example will run several times faster coded as a DO LOOP.

On the Radix

It is possible to modify the radix (number base) in which MAGIC/L operates by changing the contents of the variable RADIX. The working radix is used by MAGIC/L to control the conversion between binary data and ASCII strings. The words DECIMAL OCTAL and HEX will change the radix to those popular values but any other radix is possible, for example:

```
RADIX := 2                    ; work in binary
```

Sometimes when operating interactively you will not know what the RADIX current is, but you may not want to change it either. Your first impulse will probably be:

```
PRINT RADIX
```

but a little reflection you will realize that the answer will always be 1Ø. (A better way to determine the current radix is left as an exercise to the reader.)

Saving global values - APUSH/APOP

It is often useful to be able to modify the value of an INTEGER value temporarily, and then to restore the original value. The words APUSH and APOP provide a facility for just such a function. This facility is used in the following way:

```
APUSH <address>
  .
  . processing modifying contents of <address>
  .
APOP
```

The <address> argument may be specified by:

1) the name of an INTEGER variable
2) enclosing an ADDRESS expression in parentheses

If an ABORT (see below) should occur, all values which have been saved using the APUSH facility are restored (APOPed).

The APUSH/APOP facility is often used to save the state of the working RADIX when loading a source module. This allows the source module to modify the RADIX variable freely, while restoring the state after the entire module has been loaded. For example:

```
APUSH RADIX       ; Save the current radix value
DECIMAL           ; Operate in decimal for a while
  .
  .
OCTAL             ; Now in octal
  .
APOP              ; Restore incoming RADIX
```

Since APUSH saves a 16-bit value, it cannot be used on 32-bit machines to save ADDRESSes. On 16-bit machines, the APUSH/APOP facility can save ADDRESSes and in particular it is sometimes useful to APUSH the address of the currently active record variable of a record structure.

[Although for compatibility purposes this is not recommended, we recognize that there is occasionally benefit in trading off generality for functionality]

The following is an example of the use of APUSH with record structures. APUSH is used to save the address of the active record variable on entry to a routine, WITH statements are used within the routine and then APOP is used to re-activate the originally active record variable:

```
          ** 16-bit implementations only **
APUSH &) DATE_REC
WITH TODAY
  .
WITH BIRTHDAY
  .
APOP
```

In many cases, an APUSH statement is immediately followed by an assignment to the save variable. This can be accomplished in one statement as follows:

```
APSET ( 1Ø. , & RADIX )
```

which is functionaly equivalent to:

```
APUSH RADIX
RADIX := 1Ø.
```

Note that APSET requires the pointer to the variable to be saved, a rather unique case of "pass by reference" in MAGIC/L.

MAGIC/L Compiler I/O

Many of the routines used by the MAGIC/L compiler are also accessible directly. These built-in routines enhance even further the MAGIC/L string handling package. There are a number of variables that control string I/O in the MAGIC/L environment. These variables are used as:

- I/O channel numbers
- buffer pointers
- line counters

MAGIC/L string I/O routines operate on channel numbers which are stored in the following INTEGER variables:

ICH	MAGIC/L Input channel. If ICH is -1, the default (console) channel number CICH is used.
OCH	MAGIC/L Output channel. If OCH is -1, the default (console) channel number COCH is used.
LICH	Log channel for input. If LICH is -1, nothing is logged otherwise all input through RDLINE is logged to the file opened on channel LICH
LOCH	Log channel for output. If LOCH is -1, nothing is logged otherwise all output via TSTRL (and PRINT) is also logged to the file opened on channel LOCH
CICH	The original system input file
COCH	The original system output file

When MAGIC/L is initiated, an initialization routine opens the system input and output files and places the corresponding channel numbers CICH and COCH respectively. ICH, OCH, LICH, and LOCH are initialized to -1. The user may modify these channel numbers to divert flow to or from any other file. In the event of an error, abort, or console interrupt, ICH and OCH are reset to -1.

In addition to the channel number variables, there are a number of pointers and counters that the user may wish to modify or inspect:

$LBUF	Pointer to the input line buffer for RDLINE.
$TBUF	Pointer to the token or word buffer used by parsing routine WORD

$OBUF Pointer to the formatted output buffer.

LINE# A count of the number of lines read from the current input file

INP Byte pointer to the next byte to be parsed by WORD

Note: $TBUF, $LBUF, and $OBUF initially point to buffers which are large enough to contain the largest line allowed by the operating system (typically 132 characters). To simplify the usage of these three pointers, three CHAR arrays have been defined, TBUF, LBUF, and OBUF, which are automatically based at their pointer counterparts. (Note that this is a very special construct that cannot duplicated by the user.) The pointer form generates faster code and thus should be used when possible. The CHAR array form is provided for sub-scripting needs.

The EXT routine which was described in chapter 5, calls the LOAD routine, which is the basic "open for reading" function in MAGIC/L. The RDLINE function is the MAGIC/L text string reading routine. These routines are described below along with WORD and LOOKUP.

LOAD - Open a text file for reading

```
LOAD ( filename )
```

The LOAD routine saves the current values of ICH and LINE#, sets LINE# to zero, and opens the specified file placing the channel number into ICH. The routine ENDFILE is used to close channel ICH and restore the previous values of ICH and LINE#. [In the normal course of compilation, MAGIC/L executes ENDFILE at the end of each LOADed file]

LOOKUP - Search the dictionary for the name of a routine

```
LOGICAL := LOOKUP ( String )

LOOKUP returns  -1 if the string is the name of a
                   currently defined routine
                 Ø if the string is UNDEFINED
```

If the string is defined, the following variables will contain useful information (These USER variables are considered internal to the MAGIC/L compiler and are subject to change without notice.):

LASTDFP	the address of the Dictionary entry
LASTWORD	The execution vector of the routine (or the PARAMETER or LABEL value)
LASTTYPE	The routine type code
LASTOPM	The data-type of the word
	Ø ==> None associated with the routine
	1 ==> INTEGER
	2 ==> LONG
	3 ==> REAL
	4 ==> CHAR

RDLINE - Read a line of text

RDLINE is a logical function which reads a line of text from the file opened on ICH. RDLINE returns the value -1 if a line was read, and zero if end-of-file was encountered. The data is read into the array designated by the pointer LBUF. The parser pointer INP is initialized to the first data byte of the LBUF array (BP (LBUF)). The parser flag EOL is initialized to zero. The no-line-continuation flag EOC is initialized to "true". The end of line character (either carriage-return, line-feed, or form-feed) is placed into the variable LASTDELIM, and is not included as part of the input string. The RDLINE function is defined schematically as follows:

```
DEFINE RDLINE INTEGER
  CLEAR EOL
  SET EOC
  read the data to where LBUF points
  IF ( any bytes were read )
    INP := BP ( LBUF )
    LASTDELIM := terminating character
    INCREMENT LINE#
    SET RDLINE
  ELSE
    CLEAR RDLINE
  ENDIF
END
```

TSTRL - The MAGIC/L string output routines
TSTR
TYO
ASCII

The routine TSTRL is the primitive string output routine in MAGIC/L. TSTRL outputs a string given a byte-pointer and a byte-count. TSTR outputs a null-terminated string.

```
TSTRL ( byte-pointer , byte-count )
TSTR  ( byte-pointer )
```

TSTRL outputs the specified string on the console output channel COCH. If the output logging channel number LOCH contains a valid channel number, the string is also output to that channel. The TSTR function is provided for convenience and is defined as follows:

```
DEFINE TSTR
        ADDRESS ISTR
  TSTRL ( ISTR , LENGTH ( ISTR ) )
END
```

If only one character is to be output, the TYO routine may be used. TYO takes as an argument the ASCII value of the character and is defined as follows:

```
DEFINE TYO
        CHAR CHR
  TSTRL ( & CHR , 1 )
END
```

As a further convenience, CR is defined as TYO (12k), and will force as carraige return in the output.

A routine often used in conjunction with TYO is ASCII. ASCII has a special syntax that evaluates at compile time to the integer literal that represents the first character of the string that follows. For instance:

```
ASCII A   compiles the same as  65.
ASCII .   compiles the same as  56.
```

A common diagnostic is to place TYO (ASCII .) (or any other character) in various places in code.

These functions are used internally by MAGIC/L but are available to the user. Normally, user I/O is accomplished using the PRINT formatted output facility described in chapter 6.

WORD - The MAGIC/L parsing routine

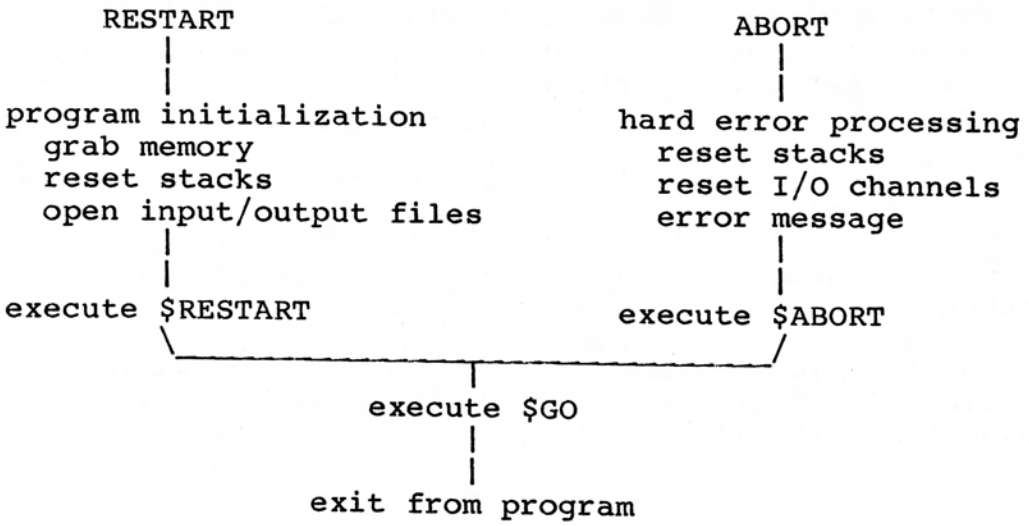
The WORD routine extracts the next token from the input buffer starting with the byte pointed to by INP. Tokens are delimited by spaces, tabs, and nulls. The parsed word is stored as a MAGIC/L string in the token buffer (pointed to by the integer $TBUF). WORD is a logical function that returns TRUE until end of line (a zero length word) is encountered. The variable EOL is set to the value of NOT WORD on exit. If EOL is true on entry to WORD, no parsing is attempted.

Program startup and error handling

MAGIC/L provides a set of user definable startup, error handling, and compiler functions. By setting these functions, the user can create any desired runtime environment. Three variables, $RESTART, $ABORT, and $GO contain the execution vector of the user-defined routines which are to perform these functions:

$RESTART	Routine executed once at program initialization
$ABORT	Routine executed when stack, syntax, keyboard interrupt, or I/O errors are encountered.
$GO	The main program. (By default the MAGIC/L compiler)

The following diagram shows the flow of control during restart or abort situations:



The RESTART Routine

The RESTART routine is a user-definable initialization routine. This routine is used to perform such one-time initialization tasks as opening files, selecting options, or pre-setting variables. MAGIC/L performs all needed initialization (opening channels, etc.) before executing $RESTART. The default restart routine, RSTØ, prints out the copyright message and should be incorporated into any new restart routine.

```
DEFINE MYSTART
  RSTØ              ; Print the standard message
  PRINT "Welcome to my program"
END
```

Any routine may be used as an initialization routine by placing its execution vector into the system variable $RESTART. In the case above:

```
$RESTART := BASE MYSTART
```

The user may place the execution vector of any routine into $RESTART. As shown in the diagram above, after the restart routine is executed, control passes to the user-definable GO routine.

The GO Routine

The routine whose execution vector is stored in the system variable $GO is executed following RESTART or ABORT. By default, $GO normally contains the address of GOØ, the MAGIC/L compiler.

If the interactive MAGIC/L programming environment is desired, there is no need to change $GO. Initialization can be handled by a user-defined restart routine, and error recovery can be handled by a user-defined abort routine.

Note that the automatic APOP on ABORT is performed as part of GOØ and this facility is lost if a new $GO routine is installed.

The ABORT Routine

The ABORT routine is executed whenever an abnormal condition exists in MAGIC/L. There are a variety of paths into the ABORT, including syntax errors, I/O errors, and stack faults. MAGIC/L immediately resets the stacks and reverts the I/O channels to the initial (console) channels to insure a secure, predictable environment.

After the system reset is complete, the function whose execution vector is in the system variable $ABORT is executed. This provides the user the opportunity to perform any error recovery procedure required by his/her program. Typical ABORT routines reset the radix, log errors, activate default records, close temporary channels, or clear special hardware.

If you wish to terminate the program on errors, simply place the execution vector of BYE in $ABORT:

```
$ABORT := BASE BYE
```

Paths Into the ABORT Routine

There are number of routines that may trigger an abort condition. These routines typically output a special message to the terminal and then execute ABORT. The abort condition is similar to a "Fatal Runtime Error" in other languages because the stacks are reset and processing is aborted. In the interactive environment, however, the program is allowed to continue.

These situations trigger an ABORT (Error reporting is treated in more detail in Appendix C):

User programmed ABORT
: The ABORT routine may be entered from a user routine by coding the word ABORT. This allows for user-written error handling routines.

Console Interrupt
: Most operating systems allow for a special keystroke sequence to interrupt processing. The mechanism for this function is different for each operating system, but in MAGIC/L they result in the execution of ABORT.

Stack Error
: MAGIC/L periodically checks the various stacks for underflow or overflow. If either of these conditions occurs, an ABORT is executed.

Compile Error

Syntactical errors encountered during compilation cause an ABORT condition. An error message is generated which describes the syntax error.

Undefined Error

At compile-time, any word which is undefined is noted as such in an error message. At run-time however, the message "UNDEFINED" is printed, and an ABORT is taken on any attempt to execute the undefined word.

Address calculations

The ADDRESS data-type is included to provide source code compatibility between 16-bit and 32-bit versions of MAGIC/L. If ADDRESS is used wherever a pointer to an address is implied then the MAGIC/L code will be directly portable among versions. Two functions have been provided for operations between INTEGERs and ADDRESSes. The functions which are described in Appendix A, are:

```
ADX     - Add ADDRESS and INTEGER
SBX     - Subtract INTEGER from ADDRESS
```

There is no overhead associated with calling these functions. In 16-bit versions, the two conversion routines are ignored by the compiler.

In addition, there are several parameters which may be used to provide source code portability when dealing with ADDRESSes. The parameters are:

```
WDSIZ    - Number of address units per 16-bit word
ADSIZ    - Number of address units per ADDRESS variable
ADRSIZE  - Number of words per ADDRESS variable
```

WDSIZ is 1 for word-addressing machines and 2 for byte-addressing machines. Typically used with the ADX function to index to the next 16-bit word.

ADSIZ is equal to WDSIZ for 16-bit machine, twice WDSIZ for 32-bit machines. Typically used with the ADX function to index to the next ADDRESS.

ADRSIZE is 1 for 16-bit machines and 2 for 32-bit machines. The ADRSIZE parameter is typically used in COMMANDS (see chapter 4) when determining the number of input arguments using CMDCNT.

```
<*
This program solves the "Tower of Hanoi" puzzle.  There are three
pegs and a number of disks that fit on the pegs.  The disks have
different sizes, and it is not legal to place a larger disk on a
smaller disk.  The object is to transfer a stack from one peg
to another, moving one disk at a time.  A recursive solution is
used here, so that a problem of 5 disks is solved by moving 4
disks, etc.  The rule of not placing large disks on small disks
is never violated so it need not be checked.

The tabbing sequences in the display are for ANSI (VT-100)
terminals.
*>

INTEGER NDISKS                      ; number of disks to use

; record structure showing the state of each peg
RECORD PEG_REC
  INTEGER CNT                       ; disks on this peg
  INTEGER DISKS ( 10 )              ; array holding disk numbers
ENDRECORD
PEG_REC PEGS ( 3 )                  ; one record variable for each peg

; initialize the records
DEFINE INIT
        INTEGER COUNT
  NDISKS := COUNT                   ; set up the count
  WITH PEGS ( 0 )                   ; start with peg zero
  CNT := NDISKS                     ; peg zero gets all disks
  DO 1 NDISKS                       ; put them on
    DISKS ( I ) := I'               ;  in reverse order
  LOOP
  CLEAR PEGS ( 1 ) : CNT            ; clear out the other
  CLEAR PEGS ( 2 ) : CNT            ;  two pegs
END

; display pegs on screen
DEFINE SHOW
  PRINT "<33>[1;1H" , #z            ; home cursor
  DO 1 NDISKS                       ; show pegs from top down
    ITER 3                          ; for each peg ...
      WITH PEGS ( I )               ; activate peg record
      IF ( CNT >= J' )              ; if there is a disk this high
        PRINT DISKS ( J' ) , #N     ; show the disk number
      ELSE                          ; otherwise
        PRINT "       ." , #N       ; just a dot
      ENDIF
    LOOP
    PRINT                           ; finish the line
  LOOP
  PRINT "      ---     ---     ---"     ; cute peg bases
END

; move one disk FROM one peg TO another
```

```
DEFINE MOVEDISK
        INTEGER FROM                ; source peg
        INTEGER TO                  ; destination peg
  LOCAL
        INTEGER CURDISK             ; temporary storage
  WITH PEGS ( FROM )                ; activate source peg
   CURDISK := DISKS ( CNT )         ; topmost disk
   DECREMENT CNT                    ; remove it
  WITH PEGS ( TO )                  ; activate dest peg
   INCREMENT CNT                    ; bump the count
   DISKS ( CNT ) := CURDISK         ; and place the disk
  SHOW                              ; show the display
END

; Move a number of disks from one peg to another.
; If the count is greater than one, move count-1 disks to the
; spare (working) peg, move one disk to the destination, then
; move count-1 from the working peg to the destination.
DEFINE MOVESTACK
        INTEGER FROM                ; source peg
        INTEGER TO                  ; dest peg
        INTEGER COUNT               ; count to move
  LOCAL
        INTEGER WORKING             ; the "working" peg
  IF ( COUNT == 1 )                 ; if count is one,
    MOVEDISK ( FROM , TO )          ;  just move it
  ELSE                              ; otherwise ...
    WORKING := 3 - ( TO + FROM ) ; determine the unused peg
    ; solve a simpler problem
    RECURSE ( FROM , WORKING , COUNT - 1 ) ; move smaller stack
    MOVEDISK ( FROM , TO )                 ;  the bottom disk
    RECURSE ( WORKING , TO , COUNT - 1 )   ;  and finish
  ENDIF
END

; high level control
; This sets the count to the given number, clears the screen,
; and move the stack from the first peg to the third.
;
DEFINE HANOI COMMAND
        INTEGER COUNT
  CHKCNT ( 1 )                      ; verify one argument
  INIT ( COUNT )                    ; initialize for that situation
  PRINT "<33>[1;1H<33>[ØJ" , #Z ; home cursor and clear screen
  SHOW                              ; show initial state
  MOVESTACK ( Ø 2 COUNT )           ; move the whole world
END
```

```
<*
The following example program is a general "filter" program that
will scan a text file and process each line of text. Two filter
routines are defined.  One removes trailing spaces and tabs from
a file, the other removes comments. Blank lines are removed. In
an attempt to show examples of usage, the high level control was
made a bit more elaborate than necessary.
*>

CHAR IFNAME ( 40. )      ; global input file name
CHAR OFNAME ( 40. )      ; global output file name

;
; Routine to clear trailing spaces and tabs from a line of text.
; The line is assumed to be in LBUF.  This routine was written
; many times by programmers converting "card image" format to
; normal text.
;
DEFINE UNSPACE
  ITER LENGTH ( $LBUF )              ; scan the string backwards
    IF ( ( LBUF ( I' ) <> 40K ) ^ ; if the char is not a space ..
     AND ( LBUF ( I' ) <> 11K ) ) ;  and not a tab
      CLEAR LBUF ( I' + 1 )          ;  clear the following char
      EXIT                           ;  and exit loop
    ENDIF
  LOOP
END

;
; Routine to clear comments from a line of text.
; Comments are indicated by ";".  After the comment is
; stripped, UNSPACE is called to clear trailing spaces and
; tabs. The line is assumed to be in LBUF.  Because this
; routine scans forward, CINDEX may be used.  Puzzle: This
; routine has a bug which prevents its use on this source
; file!
;
DEFINE UNCOMMENT
  LOCAL
        INTEGER TCNT
  TCNT := CINDEX ( $LBUF , ASCII ; )     ; look for ";"
  IF ( TCNT >0 )                    ; if there is one
    CLEAR LBUF ( TCNT - 1 )         ;  zap it!
  ENDIF
  UNSPACE                  ; now clear out trailing spaces
END

;
; Action routine for processing a file.
; The input file name is assumed in IFNAME, the ouput file
; name is in OFNAME.  Input is handled with
; LOAD...RDLINE...ENDFILE which will place the line in LBUF.
; Output is handled by using the #C print directive to divert
; the output to a file.  A pointer to the filter function is
```

```
; passed as the parameter of the action routine.  The extra
; level of indirection is resolved by declaring the input as
; an array.
;
DEFINE FILTERFILE ACTION
        INTEGER FILTER ( Ø )     ; array of parameters
  LOCAL
        INTEGER PCH              ; channel for output
  LOAD ( IFNAME )                ; open the input file
  PCH := OPEN ( OFNAME , 'WNCL ) ; open the output file
  WHILE ( RDLINE )               ; read a line
    EXEC ( FILTER ( Ø ) )        ; execute filter routine
    IF ( LBUF <>Ø )              ; if not a null line ...
      PRINT #C PCH , LBUF        ;  print to the output file
    ENDIF
  REPEAT                         ; back for more
  CLOSE ( PCH )                  ; close output
  ENDFILE                        ; close input
END

;
; Declare two filtering routines as instances of the
; Action Class FILTERFILE.  The parameters are the
; functions to execute on each line.
;
MAKE    'CLRSPACE       FILTERFILE      BASE UNSPACE
MAKE    'CLRCMNT        FILTERFILE      BASE UNCOMMENT

;
; Define the high level control.  FIXFILE is a PARSED routine
; that takes a single argument:
;       FIXFILE <filename>[/C]
; If the filename has no extension, ".MG" is appended to the
; input filename.  The output filename will have the extension
; ".OUT". The CLRSPACE filter is assumed unless a /C is
; present.  In this case, the filter CLRCMNT is used.
;
DEFINE FIXFILE PARSED
        CHAR INLINE ( Ø )        ; input line
  LOCAL
        INTEGER TCNT             ; temp used for CINDEX
        INTEGER CMNTFLAG         ; flag of which filter
  CHKCNT ( ADRSIZE )             ; verify argument list
  CLEAR CMNTFLAG                 ; assume CLRSPACE
  TCNT := CINDEX ( INLINE , ASCII / )   ; look for a "/"
  IF ( TCNT >Ø )                 ; if there is a switch ..
    IF ( ( INLINE ( TCNT ) AND 137K ) == ASCII C ) ; and a C (or c) ..
      INCREMENT CMNTFLAG         ; setup for CLRCMNT
    ELSE
      PRINT "Unknown Option"     ; Nasty message if not /C
      ABORT                      ;  and get out
    ENDIF
    CLEAR INLINE ( TCNT - 1 )    ; null the "/"
  ENDIF
```

```
  IFNAME := INLINE                  ; set up input
  OFNAME := INLINE                  ;  and output files
  TCNT := CINDEX ( INLINE , ASCII . )   ; look for extension
  IF ( TCNT >Ø )
    CLEAR OFNAME ( TCNT - 1 )       ; remove extension from output
  ELSE
    IFNAME += ".MG"                 ; add extension to input
  ENDIF
  OFNAME += ".OUT"                  ; add extension to output
  CASE ( CMNTFLAG )                 ; execute appropriate filter
    CLRSPACE                        ; if CMNTFLAG is Ø
    CLRCMNT                         ; if CMNTFLAG is 1
  ENDIF
END

;
; Although FIXFILE may be called from the keyboard, sometimes
; its nice to create a "turnkey" program that takes arguments
; from the command line.
; This startup routine is defined for CP/M systems, where the
; command line is left at location 81X.  Other systems would
; require a different setup.  Setting $ABORT to BYE forces
; program termination in case of error.
;
DEFINE STARTUP
  $ABORT := BASE BYE     ; exit on errors
  NOPARSE FIXFILE 81X    ; process the input
  BYE                    ; exit
END

$RESTART := BASE STARTUP
SAVE FIXFILE
```

Appendix A MAGIC/L Library Routines

This chapter describes the library of functions provided in the MAGIC/L environment. These routines can be grouped into the functional categories listed below.

- INTEGER functions
- REAL functions
- Trigonometric Functions
- STRING functions
- INTEGER Shift Functions
- LONG Shift Functions
- Mixed Mode Operations (LONG/INTEGER)
- Mixed Mode Operations (ADDRESS/INTEGER)
- Mode Conversion Routines
- INTEGER Packing and Unpacking Routines
- Bit Handling Routines
- Block Data Movement
- Memory Access Functions

Most of the functions described in this chapter are built-in to MAGIC/L, except that most of the LONG/INTEGER mixed-mode routines may supplied in a separate file named MIXLIB. All routines which would require REAL support are only included when REAL support is supplied.

INTEGER functions

This section describes the library routines which operate on INTEGERs and return INTEGER values. The following functions are described in this section (The asterisk (*) denotes functions which have signed and unsigned versions):

```
        ABS     - INTEGER Absolute Value Function
        CLM     - Range comparison function
        * ISCAL - Extended precision Scaling function
        * MAX   - Maximum of two INTEGERs
        * MIN   - Minimum of two INTEGERs
        MOD     - Modulo (Remainder) function
        UDIV    - Unsigned INTEGER divide
```

ABS - INTEGER Absolute Value Function

```
        <integer> := ABS ( IVAL )
```

The Absolute Value of the signed INTEGER IVAL is returned.

CLM - Range comparison function

```
        <logical> := CLM ( IVAL , LOLIM , HILIM )
```

Returns true if:

```
                LOLIM <= IVAL <= HILIM
```

The signed INTEGER value IVAL is compared to the signed INTEGER values LOLIM and HILIM. If IVAL is within the limits specified by LOLIM and HILIM, the function result is true. If IVAL is outside the limits, the result is false. (The range includes the values LOLIM and HILIM)

ISCAL - Extended precision Scaling function
UISCAL - Extended precision Scaling function (unsigned)

```
        <integer> := ISCAL  ( IVAL1 , IVAL2 , IVAL3 )
        <integer> := UISCAL ( IVAL1 , IVAL2 , IVAL3 )
```

This function returns the 16-bit value:

```
        ( IVAL1 * IVAL2 ) / IVAL3
```

The intermediate product of IVAL1 and IVAL2 has 32-bit precision. In the case of UISCAL, the arguments are treated as unsigned values. In essence the integer IVAL1 is scaled by the fraction IVAL2/IVAL3.

MAX - Return the larger of two integers
UMAX - Return the larger of two integers (unsigned)

```
<integer> := MAX  ( IVAL1 IVAL2 )
<integer> := UMAX ( IVAL1 IVAL2 )
```

The resulting value is the larger of IVAL1 and IVAL2

MIN - Return the smaller of two integers
UMIN - Return the smaller of two integers (unsigned)

```
<integer> := MIN  ( IVAL1 IVAL2 )
<integer> := UMIN ( IVAL1 IVAL2 )
```

The resulting value is the smaller of IVAL1 and IVAL2

MOD - Modulo (Remainder) function

```
<integer> := MOD ( IVAL1 , IVAL2 )
```

The remainder of the signed division IVAL1/IVAL2 is returned. If the quotient is negative, then the remainder is also negative. In order to create a positive modulo, add IVAL2 to the negative remainder.

UDIV - Unsigned INTEGER divide

The unsigned INTEGER divide is similar to the divide operator except that the dividend and divisor are treated as 16-bit unsigned numbers.

```
<integer> := UDIV ( UVAL1 , UVAL2 )
```

The unsigned value UVAL1 is divided by the unsigned value UVAL2.

REAL Functions

This section describes the library routines which operate on REALs and return REAL values. The following functions are described in this section:

```
FABS    - REAL Absolute Value
AINT    - Extract the integer part of a REAL
FRACT   - Extract the fractional part of a REAL
EXP     - Raise 'e' to a REAL power
EXP2    - Raise  2.Ø to a REAL power
EXP1Ø   - Raise 1ø.Ø to a REAL power
LOG     - Logarithm (Base e)
LOG2    - Logarithm (Base 2)
LOG1Ø   - Logarithm (Base 1Ø)
SQRT    - Square Root
```

EXP - Raise 'e' to a REAL power
EXP2 - Raise 2.Ø to a REAL power
EXP1Ø - Raise 1Ø.Ø to a REAL power

```
<real> := EXP   ( RVAL )
<real> := EXP2  ( RVAL )
<real> := EXP1Ø ( RVAL )
```

The functions EXP, EXP2, and EXP1Ø perform exponentiation on the number bases e,2, and 1Ø respectively.

FABS - REAL Absolute Value

```
<real> := FABS ( RVAL )
```

The Absolute Value of the REAL RVAL is returned.

AINT - Extract the integer part of a REAL

```
<real> := AINT ( RVAL )
```

The integer part of the input value RVAL is returned. The sign of the returned value is the same as the sign of RVAL.

RVAL	AINT (RVAL)
5.25	5.ØØ
-9.99	-9.ØØ
-4.ØØ	-4.ØØ

FRACT - Extract the fractional part of a REAL

```
<real> := FRACT ( RVAL )
```

The fractional part of the input value RVAL is returned. The sign of the returned value is the same as the sign of RVAL.

RVAL	FRACT (RVAL)
5.25	Ø.25
-9.99	-Ø.99
-4.ØØ	Ø.ØØ

LOG - Logarithm (Base e)
LOG2 - Logarithm (Base 2)
LOG1Ø - Logarithm (Base 1Ø)

```
<real> := LOG   ( RVAL )
<real> := LOG2  ( RVAL )
<real> := LOG1Ø ( RVAL )
```

The Logarithm of the REAL RVAL is returned. The number base of the logarithm is e,2 or 1Ø for LOG, LOG2, and LOG1Ø respectively.

SQRT - Square Root of a REAL number

```
<real> := SQRT ( RVAL )
```

The Square Root of the REAL RVAL is returned.

Trigonometric Functions

This section describes the trigonometric functions supported by MAGIC/L. REAL support is a prerequisite for the trig routines. On some implementations, hardware floating point support is also required.

```
PI      - A well known REAL constant
RAD     - Convert Degrees to Radians
DEG     - Convert Radians to Degrees
SIN     - Sine of an Angle in Radians
COS     - Cosine of an Angle in Radians
TAN     - Tangent of an Angle in Radians
ASIN    - Arc-sine (in radians) of a REAL number
ACOS    - Arc-cosine (in radians) of a REAL number
ATAN    - Arc-tangent (in radians) of a REAL number
ATAN2   - Polar angle of X and Y coordinates
```

DEG - Convert Radians to Degrees

```
<degrees> := DEG ( <radians>  )
```

The input value is converted from radians to degrees.

RAD - Convert Degrees to Radians

```
<radians> := RAD ( <degrees> )
```

The input value is converted from degrees to radians.

```
SIN     - Sine of an Angle in Radians
COS     - Cosine of an Angle in Radians
TAN     - Tangent of an Angle in Radians

<real> := SIN ( <radians> )
<real> := COS ( <radians> )
<real> := TAN ( <radians> )
```

The value of trigonometric function is returned as a REAL.

```
ASIN    - Arc-sine (in radians) of a REAL number
ACOS    - Arc-cosine (in radians) of a REAL number
ATAN    - Arc-tangent (in radians) of a REAL number

<radians> := ASIN ( <real> )
<radians> := ACOS ( <real> )
<radians> := ATAN ( <real> )
```

The angle in radians is returned as a REAL.

ATAN2 - Polar angle of X and Y coordinates

```
<radians> := ATAN2 ( Y , X )
```

ATAN2 returns four quadrant arc-tangent of (Y/X). This can also be thought of as the polar angle (in radians) measured counter-clockwise from the X-axis of the cartesian coordinate (X,Y).

String Manipulation Routines

The routines described in this section are used in string manipulation. Included are the following routines which perform concatenation, string movement, and string sizing.

```
CINDEX  - Locate a character within a string
CLUC    - Case-Conversion (Lower case to Upper case)
CONCAT  - String concatenation
$CONCAT - String concatenation (primitive)
LENGTH  - Determine the length of a null-terminated string
MVSTR   - move a string into a destination array
STCMP   - string comparison
STCOM   - string comparison for equality
```

CINDEX - Locate a character within a string

```
<offset> := CINDEX ( string , char )

string          pointer to a null-terminated string
char            byte-value of the character to be
                located
```

The input string is scanned for the specified character. The returned value <offset> reflects whether the character was located, and if so, where in the string it was located.

If found, <offset> is the number of characters in the substring including the located character. Note that this is one greater than the "subscript" number of the located character.

If not found, <offset> is the negative of the length of the string.

If the string is "null," <offset> will be zero.

Examples:

```
CINDEX ( "ABCDE" , 103K )       returns:  3
CINDEX ( "ABCDE" ,  77K )       returns: -5
```

A common usage of CINDEX is checking for the existence of extensions on filenames. In the following example, a default extension is added to a filename if no extension is present.

```
CHAR FNAME ( 20. )
IF ( CINDEX ( FNAME , ASCII . ) <0 )    ; no extension
  FNAME += ".DAT"                       ; add one
ENDIF
```

CLUC - Case-Conversion (Lower case to Upper case)

```
CLUC ( string )
```

All lower-case alphabetic characters in the specified string are converted to upper-case. The string is modified in place. CLUC should not be used directly on string literals, or when the argument might reside in read-only memory.

CONCAT and $CONCAT - String concatenation

```
CONCAT ( dest-string , source-string )
$CONCAT ( dest-string , source-BP , source-len )
```

The first form, CONCAT, is normally used. $CONCAT is supplied so that pieces of a string may be extracted. Concatenation to a null string is the same as MVSTR. String literals should never be concatenated to, since no extra space is provided. Care should be taken that the destination array is large enough for the new string.

LENGTH - Determine the length of a null-terminated string

```
<integer> := LENGTH ( string-ptr )
```

LENGTH counts the number of bytes in the null-terminated string whose string-pointer is specified. (The null is NOT included in the count.)

MVSTR - move a string into a destination array

```
MVSTR ( Source-string , destination-array )
```

Strings are often initialized by moving a string literal to an array:

```
MVSTR ( "This is a string literal" , dest-string )
```

Note that id dest-string is a CHAR array, this is the same as:

```
dest-string := "This is a string literal"
```

STCMP - string comparison

```
        <relation> := STCMP ( String1 String2 )
```

STCMP compares the two input strings byte by byte. If the strings differ, the relationship between the ASCII codes of the differing bytes is reflected in the returned value. (This routine is sensitive to case e.g. "ABC" is less than "ABc".) In addition, the magnitude of the returned valued indicates the offset into the strings where the difference occurred. STCMP is the primitive used by the string relational operators.

```
        STCMP returns   negative if String1 <  String2
                            zero if String1 == String2
                        positive if String1 >  String2
```

For example:

```
        STCMP ( "ABC" , "ABCD" )   returns -4
        STCMP ( "ABC" , "XYZ" )    returns -1
        STCMP ( "ABCXY" , "ABCD" ) returns 4
```

STCOM - string comparison for equality

```
        LOGICAL := STCOM ( String1 String2 )

        STCOM returns   -1 if the strings are identical
                         Ø if they differ
```

STCOM is implemented as follows:

```
        DEFINE STCOM INTEGER
                ADDRESS STR1 STR2
          STCOM := STCMP ( STR1 , STR2 ) ==Ø
        END
```

String conversion routines

The routines which are used by the MAGIC/L compiler to convert input strings into numeric values are also available to the user. The routine NLITERAL is the primitive ASCII to integer conversion routine. NLITERAL is called by two other routines. ILITERAL and LLITERAL which are used to determine whether the string is a legal INTEGER or LONG respectively. In systems which contain REAL support, the routine RLITERAL performs the ASCII to REAL conversion.

NLITERAL - Convert ASCII to 32-bit integer

```
<logical> := NLITERAL ( string-pointer )
```

NLITERAL will return true if the string is a valid INTEGER or LONG according to the MAGIC/L literal conventions described in Chapter 1.

The result of the conversion is placed in the LONG variable LLVAL. The INTEGER variable ILVAL is equivalenced to the least-significant 16-bits of LLVAL.

NLITERAL will also set bits in the flag word LITFLAG, so that the user may determine the nature of the string converted. The most significant byte of LITFLAG is used to describe the converted number, the least significant byte of LITFLAG contains the number of digits in the converted string.

LITFLAG Bit Mask	Meaning
8000x	Set if number is an implicit LONG (value overflowed 65536.)
4000x	Set if number is an explicit LONG (a trailing "L" was specified)

ILITERAL - Convert ASCII to INTEGER

```
<logical> := ILITERAL ( string )
```

The ILITERAL routine is used by the MAGIC/L compiler to determine whether a string is a legal INTEGER literal. If so, ILITERAL returns <true>, and the converted value is placed in the INTEGER variable ILVAL. The actual definition of ILITERAL is as follows:

```
; Integer literal processor
;  Returns TRUE if valid integer, else FALSE
;  Converted value will be in ILVAL
DEFINE ILITERAL INTEGER
        ADDRESS ISTR
  ILITERAL := NLITERAL ( ISTR ) AND ^
     ( ( LITFLAG AND ØCØØØX ) ==Ø )
END
```

LLITERAL - Convert ASCII to LONG

```
<logical> := LLITERAL ( string )
```

The LLITERAL routine is used by the MAGIC/L compiler to determine whether a string is a legal INTEGER literal. If so, LLITERAL returns <true>, and the converted value is placed in the LONG variable LLVAL. This routine is identical to the routine NLITERAL.

RLITERAL - ASCII to REAL conversion

Most implementations of MAGIC/L support 32 bit REAL numbers. These versions of MAGIC/L will have the function RLITERAL.

```
logical := RLITERAL ( string-pointer )
```

The result of the conversion will be placed in the variable RLVAL (which occupies the same memory location as LLVAL). RLITERAL does not modify the variable LITFLAG.

INTEGER Shift Functions

This section describes the INTEGER shifting functions provided in MAGIC/L. The functions described in this section are:

LSHIFT - INTEGER shift left
RSHIFT - Arithmetic INTEGER shift right
URSHIFT - Logical INTEGER shift right

LSHIFT - INTEGER shift left

<integer> := LSHIFT (IVAL , shift-count)

The bits in the 16-bit integer IVAL are shifted to the left the specified number of times. On each shift, the LSB is set to zero, and the MSB is lost.

RSHIFT - Arithmetic INTEGER shift right

<integer> := RSHIFT (IVAL , shift-count)

The bits in the 16-bit integer IVAL are shifted to the right the specified number of times. On each shift, the MSB (sign bit) is unchanged, and is propagated towards the right. The LSB is lost).

URSHIFT - Logical INTEGER shift right

<integer> := URSHIFT (IVAL , shift-count)

The bits in the 16-bit integer IVAL are shifted to the right the specified number of times. On each shift, the MSB is set to zero, and the LSB is lost.

LONG Shift Functions

This section describes the LONG shifting functions provided in MAGIC/L. The functions described in this section are:

LLSHIFT - LONG shift left
LRSHIFT - Arithmetic LONG shift right
LURSHIFT - Logical LONG shift right

LLSHIFT - LONG shift left

```
<long> := LLSHIFT ( LVAL , shift-count )
```

The bits in the 32-bit integer LVAL are shifted to the left the specified number of times. On each shift, the LSB is set to zero, and the MSB is lost.

LRSHIFT - Arithmetic LONG shift right

```
<long> := LRSHIFT ( LVAL , shift-count )
```

The bits in the 32-bit integer LVAL are shifted to the right the specified number of times. On each shift, the MSB (sign bit) is unchanged, and is propagated towards the right. The LSB is lost).

LURSHIFT - Logical LONG shift right

```
<long> := LURSHIFT ( LVAL , shift-count )
```

The bits in the 32-bit integer LVAL are shifted to the right the specified number of times. On each shift, the MSB is set to zero, and the LSB is lost.

Mixed Mode Operations (LONG/INTEGER)

This section describes a set of mixed-mode functions which operate on LONGs and INTEGERs typically returning a LONG value. These functions are supplied in source form in the module MIXLIB.MG The following functions are described in this section (The asterisk (*) denotes functions which have signed and unsigned versions):

```
        LIADD   - Add LONG and INTEGER
        * LIDIV - Divide LONG by INTEGER with INTEGER quotient
        * LIMUL - Multiply two INTEGERs with LONG product
        LISUB   - Subtract INTEGER from LONG
```

LIADD - Add LONG and INTEGER

```
        <long> := LIADD ( LVAL , IVAL )
```

The 16-bit integer IVAL is sign-extended to 32-bits and added to the 32-bit integer LVAL.

LIDIV - Divide LONG by INTEGER with INTEGER quotient
ULIDIV - Divide LONG by INTEGER with INTEGER quotient(unsigned)

```
        <integer> := LIDIV  ( LVAL , IVAL )
        <integer> := ULIDIV ( LVAL , IVAL )
```

The 32-bit integer LVAL is divided by the 16-bit integer IVAL. The result is a 16-bit integer value. In the case of ULIDIV the arguments are treated as unsigned values.

LIMUL - Multiply two INTEGERs with LONG product
ULIMUL - Multiply two INTEGERs with LONG product (unsigned)

```
        <long> := LIMUL  ( IVAL1 , IVAL2 )
        <long> := ULIMUL ( IVAL1 , IVAL2 )
```

The 16-bit integers IVAL1 and IVAL2 are multiplied. The result is a 32-bit product. In the case of ULIMUL the two arguments are treated as unsigned integers.

LISUB - Subtract INTEGER from LONG

```
        <long> := LISUB ( LVAL , IVAL )
```

The 16-bit integer IVAL is sign-extended to 32-bits and subtracted from the 32-bit integer LVAL.

Mixed Mode Operations (ADDRESS/INTEGER)

The functions in this section are included to provide source level portability for MAGIC/L code. The functions ADX and SBX are alternate names for other functions included in MAGIC/L but the actual alternate depends on whether the MAGIC/L is a 16-bit or a 32-bit implementation.

function	16-bit	32-bit
ADX	INTEGER "+"	LIADD
SBX	INTEGER "-"	LISUB

ADX - Add ADDRESS and INTEGER

```
<address> := ADX ( <address> , IVAL )
```

The 16-bit integer IVAL is added to the specified ADDRESS. If addresses are 32-bits, IVAL is sign-extended to 32-bits.

SBX - Subtract INTEGER from ADDRESS

```
<address> := SBX ( <address> , IVAL )
```

The 16-bit integer IVAL is subtracted from the specified ADDRESS. If addresses are 32-bits, IVAL is sign-extended to 32-bits.

Mode Conversion Routines

The routines described in this section are used to convert between different data-types. Conversions are supported among INTEGER, LONG, ADDRESS, and REAL data-types as follows:

INTEGER-LONG conversion

- LSWORD
- MSWORD
- UXTND
- WDLONG
- XTND

INTEGER-REAL conversion

- FIX
- FLOAT

INTEGER-ADDRESS conversion

- AD2INT
- INT2AD

LONG-REAL conversion

- LFIX
- LFLOAT

AD2INT - Convert ADDRESS to INTEGER

```
<integer> := AD2INT ( AVAL )
```

The ADDRESS AVAL is converted to an INTEGER. AD2INT is commonly used to convert the difference between two ADDRESS variables to an INTEGER value.

FIX - Convert REAL to INTEGER

```
<integer> := FIX ( RVAL )
```

The REAL value RVAL is truncated and converted to an INTEGER. The range of the result is -32768. to +32767. No indication of overflow is given

FLOAT - Convert INTEGER to REAL

```
<real> := FLOAT ( IVAL )
```

The INTEGER IVAL is converted to REAL.

INT2AD - Convert INTEGER to ADDRESS

```
        <address> := INT2AD ( IVAL )

The INTEGER IVAL is converted to an ADDRESS.
-------------------------------------------------------------
LFIX    - Convert REAL to LONG

        <long> := LFIX ( RVAL )

The REAL value RVAL is truncated and converted to an LONG.  No
indication of overflow is given.
-------------------------------------------------------------
LFLOAT  - Convert LONG to REAL

        <real> := LFLOAT ( LVAL )

The LONG LVAL is converted to REAL.
-------------------------------------------------------------
LSWORD  - Extract the least significant 16-bits from a LONG

        <integer> := LSWORD ( LVAL )

The least significant 16-bits of the argument LVAL are
returned LSWORD is used to convert a LONG value into an
INTEGER value by truncation.
-------------------------------------------------------------
MSWORD  - Extract the most significant 16-bits from a LONG

        <integer> := MSWORD ( LVAL )

The most significant 16-bits of the argument LVAL are returned
-------------------------------------------------------------
WDLONG  - Form a LONG from 2 INTEGERs

        <long> := WDLONG ( IVAL1 , IVAL2 )

A 32-bit LONG integer is formed with IVAL1 as the most
significant 16 bits and IVAL2 as the least significant 16 bits
-------------------------------------------------------------
XTND    - Convert INTEGER to LONG (signed)
UXTND   - Convert INTEGER to LONG (unsigned)

        <long> := XTND  ( IVAL )
        <long> := UXTND ( IVAL )

XTND:   The 16-bit INTEGER value IVAL is sign-extended to form a
        signed 32-bit LONG value.

UXTND:  The 16-bit INTEGER value IVAL is extended with 16 zeros to
        form an unsigned 32-bit LONG value.
-------------------------------------------------------------
```

INTEGER Packing and Unpacking Routines

The routines in this section are provided are used to pack and unpack byte and nibble fields within INTEGER values. The routines included in this section are:

```
Packing Routines
----------------
BYTEWD
NIBBYTE

Unpacking Routines
------------------
LSBYTE
LSNIB
MSBYTE
MSNIB
```

BYTEWD - form an INTEGER from two 8-bit values

```
<integer> := BYTEWD ( IVAL1 , IVAL2 )
```

The two values IVAL1 and IVAL2 are masked to 8 bits. The result is then formed by shifting the masked IVAL1 8 bits to the left and adding IVAL2.

LSBYTE - Extract the least significant byte from an INTEGER

```
<integer> := LSBYTE ( IVAL )
```

The INTEGER argument value is masked to 8-bits

LSNIB - Extract the least significant nibble from a byte

```
<integer> := LSNIB ( IVAL )
```

The INTEGER argument value is masked to 4 bits

MSBYTE - Extract the most significant byte from an INTEGER

```
<integer> := MSBYTE ( IVAL )
```

The most significant 8-bits are extracted from the 16-bit argument value. (The value is byte-swapped and masked to 8 bits)

MSNIB - Extract the most significant nibble from a byte

```
<integer> := MSNIB ( IVAL )
```

The most significant 4 bits are extracted from the least significant byte of the INTEGER argument value. (The argument

is shifted right 4 bits and then masked to 4 bits)

NIBBYTE - Form a byte from two 4-bit values

```
<integer> := NIBBYTE ( IVAL1 , IVAL2 )
```

The two values IVAL1 and IVAL2 are masked to 4 bits. The result is then formed by shifting the masked IVAL1 4 bits to the left and adding IVAL2.

Bit Handling Routines

Bit addressing in MAGIC/L requires two 16-bit words. The first word is a 16-bit unsigned bit number within an array. The second word is the address of the bit array. The bit offset allows for a 65536 bit or 8192 byte array. Bit offsets Ø-15 address bits Ø-15 of word Ø of the array. Bit offsets 16-31 address bits Ø-15 of word 1 of the array, etc. Note that the ordering of bits within a word is implementation dependent. The bit handling routines are:

```
CLRBIT  - Clear a bit
GETBIT  - Test a bit
SETBIT  - Set a bit
```

CLRBIT - Clear a bit in a bit array

```
CLRBIT ( bit-offset , array )
```

The bit addressed by the specified bit-offset and array address is set to Ø.

GETBIT - Test a bit in a bit array

```
<integer> := GETBIT ( bit-offset , array )
```

The bit addressed by the specified bit-offset and array address is tested.
If the bit is zero, Ø is returned If the bit is one, -1 is returned

SETBIT - Set a bit in a bit array

```
SETBIT ( bit-offset , array )
```

The bit addressed by the specified bit-offset and array address is set to 1.

Block Data Manipulation

The following routines are provided to allow for manipulation of contiguous blocks of memory. These routines can be further divided into several categories:

Block Display

DISP
BDISP
CDISP

Short Integer Print

=
?

Block Initialization

FILL
MVZER

Block Movement

MVBYTES
MVWDS
RMVBYTES
RMVWDS

Block Comparison

VEQ

The block movement routines are time efficient even if only three words are moved. The routines RMVBYTES and RMVWDS transfer the last word of the data area first.

DISP - Display the contents of an array

```
DISP  ( array-addr , wcnt )
BDISP ( array-addr , bcnt )
CDISP ( array-addr , bcnt )
```

array-addr	is the address of the array
wcnt	is the number of words to be displayed starting from the specified address
bcnt	is the number of bytes to be displayed starting from the specified address

The three display routines are similar in form but are used in slightly different situations. DISP is used when displaying INTEGER arrays. The count and labeling is in words. BDISP is used for displaying areas of memory. The display is still in words, but the labeling and count are in bytes. CDISP is for displaying arrays of characters or byte values and the labeling and count are in bytes.

As an example of the use of DISP, consider an array called SQUARES. We could initialize the array as follows:

```
INTEGER SQUARES ( 35 )

ITER 33
  SQUARES ( I ) := I * I
LOOP
```

We could then display the SQUARES array:

```
DISP ( SQUARES , 35 )
```

18819	Ø	1	2	3	4	5	6	7	8	9
Ø	Ø	1	4	9	16	25	36	49	64	81
1Ø	1ØØ	121	144	169	196	225	256	289	324	361
2Ø	4ØØ	441	484	529	576	625	676	729	784	841
3Ø	9ØØ	961	1Ø24	Ø	Ø					

The number 18819 is the address of the specified array SQUARES. The subscript of any of the displayed values is the sum of the number at the top of the column and the number at the left of the row.

= - Integer variable display command
? - Integer memory display command

```
= <variable> [<variable> ... ]
? <address>  [<address> ... ]
```

These two routines are a quick means of displaying variables or the contents of memory locations at the keyboard. All values are displayed as unsigned integers separated by a space. The "=" command is used to display the contents of variables or the returned values from functions. The "?" command adds an extra of indirection so that it displays the contents of a memory location or the value that an integer points to (in 16-bit systems).

These commands do not attempt to resolve the data type of the arguments, thus a LONG or a REAL will be displayed as its two INTEGER parts. The order of the two parts will be implementation dependent. In addition, a comma is not required between arguments unless needed to separate expressions. These routines are not recommended in formal coding, they are intended only for keyboard use.

Examples:

```
INTEGER X Y LONG Z X := 3 ;; Y := 5

= X Y
displays as:
 3 5

Z := 1ØL
= Z
displays as:
 1Ø Ø
```

FILL - Fill an array with a value

```
FILL ( array-addr , nwords , value )
```

"Nwords" words of the specified array are set to the specified value. The routine MVZER below is equivalent to:

```
FILL ( array-addr , nwords , Ø )
```

MVBYTES - Move a string of bytes (bottom to top)

```
MVBYTES ( source-bptr , dest-bptr , nbytes )
```

source-bptr	is a byte-pointer to the first byte in the source string
dest-bptr	is a byte-pointer to the first byte in the destination string
nbytes	is the number of bytes to be moved

Note: if the source string and the destination string overlap, the source string will be duplicated properly only if source-bptr is higher than dest-bptr. (see RMVBYTES)

MVWDS - Move a block of data (bottom to top)

```
MVWDS ( source-addr , dest-addr , nwords )
```

source-addr	is the address of the first word in the source array
dest-addr	is the address of the first word in the destination array
nwords	is the number of words to be moved

Note: if the source array and the destination array overlap, the source array will be duplicated properly only if source-addr is higher than dest-addr. (see RMVWDS)

MVZER - Clear an array

```
MVZER ( array-addr , nwords )
```

"Nwords" starting at the specified array address are cleared. Note: The second argument refers to words of memory, not array elements. This means that for LONG and REAL arrays, "nwords" must be twice the number of array elements to be cleared.

RMVBYTES - Move a string of bytes (top to bottom)

RMVBYTES (source-bptr , dest-bptr , nbytes)

source-bptr	is a byte-pointer to the first byte in the source string
dest-bptr	is a byte-pointer to the first byte in the destination string
nbytes	is the number of bytes to be moved

Note: if the source string and the destination string overlap, the source string will be duplicated properly only if source-bptr is lower than dest-bptr. (see MVBYTES)

RMVWDS - Move a block of data (top to bottom)

RMVWDS (source-addr , dest-addr , nwords)

source-addr	is the address of the first word in the source array
dest-addr	is the address of the first word in the destination array
nwords	is the number of words to be moved

Note: if the source array and the destination array overlap, the source array will be duplicated properly only if source-addr is lower than dest-addr. (see MVWDS)

VEQ - Compare two integer arrays

<logical> := VEQ (array1 , array2 , nwords)

array1	is the address of the first array
array2	is the address of the second array
nwords	is the number of words to be compared

VEQ returns TRUE (-1) if the two arrays are indentical, otherwise it returns FALSE (Ø).

Memory Access Functions

The memory access routines are used when it is desired to access a specific address in memory. The "peek" routines return data from memory, the poke routines store data into memory. The following routines are described in this section:

```
BPEEK   - Single Byte access
BPOKE
LPEEK   - LONG integer access
LPOKE
PEEK    - INTEGER access
POKE
RPEEK   - REAL access
RPOKE
APEEK   - ADDRESS access
APOKE
```

BPEEK - Return the contents of a byte from memory

```
<integer> := BPEEK ( byte-ptr )
```

The byte addressed by the specified byte-ptr is returned

BPOKE - Store a value into a memory location

```
BPOKE ( IVAL , byte-ptr )
```

The value IVAL is masked to 8 bits and is stored into the byte addressed by the specified byte-ptr.

LPEEK - Return the contents of a 32-bit memory location

```
<long> := LPEEK ( address )
```

The 32-bit contents of the specified address (and address+1) are returned

---

LPOKE - Store a value into a memory location

```
LPOKE ( LVAL , address )
```

The 32-bit value LVAL is stored in the specified address (and address+1)

PEEK - Return the contents of a 16-bit memory location

<integer> := PEEK (address)

The 16-bit contents of the specified address are returned

POKE - Store a value into a memory location

POKE (IVAL , address)

The 16-bit value IVAL is stored in the specified location

RPEEK - Return a REAL value from a memory location

<real> := RPEEK (address)

The 32-bit contents of the specified address (and address+1) are returned as a REAL value

---

RPOKE - Store a value into a memory location

RPOKE (RVAL , address)

The 32-bit REAL value RVAL is stored in the specified address (and address+1)

APEEK - Return a ADDRESS value from a memory location

<address> := APEEK (address)

This routine is the same as PEEK or LPEEK, depending on the machine used.

APOKE - Store an ADDRESS value into a memory location

APOKE (AVAL , address)

This routine is the same as POKE or LPOKE, depending on the machine used.

A set of high-level file I/O functions has been defined in MAGIC/L which allow the user to create, open and close files, as well as perform various types of I/O. Although I/O is an operating system dependent function, the MAGIC/L I/O package exists in the same form on all implementations of MAGIC/L providing source level compatibility across operating system and processor boundaries. ALL of the calls described in this appendix exist on ALL implementations of MAGIC/L. You should refer to the system supplement for any information concerning system dependent considerations.

All access to a file (or device) is made by reference to an "I/O channel." The association between a filename and a channel is made by the OPEN function, which returns the INTEGER channel number associated with the OPENed file. In the OPEN call the user may specify various options pertaining to file creation and access. In most implementations, the MAGIC/L channel number (as returned by the OPEN function) is the same as the system channel number.

MAGIC/L supports three type of I/O: block, text, and sequential. Block I/O is the most primitive form of I/O, normally corresponding to the operating system's direct, unbuffered disk I/O. Block I/O is random access, and transfers data in multiples of 512 bytes. In some systems, notably CP/M, disk I/O is based on "sectors" which are smaller than 512 bytes. MAGIC/L will cluster several sectors together to from a "disk block."

Text I/O and Sequential I/O are buffered, and thus may transfer an arbitrary number of bytes on each call.

Within MAGIC/L the line-feed (12 octal,ØA hex) is treated as the new-line character. RDTXT converts the system dependent line delimiter into a line-feed internally to MAGIC/L. WRTXT converts all line-feeds into the system dependent line delimiter. This means that inserting a line-feed character into a string which MAGIC/L prints will generate the proper output data in an operating system independent way.

Sequential I/O is sometimes called "Stream I/O" because it treats the file as a continuous stream of data bytes.

Normally, when a file is read or written using sequential or text I/O, the user does not need to know the exact position of

the data in the file. MAGIC/L however allows the user to determine and/or specify the file position of the next buffered I/O transfer with the GPOS and SPOS calls. The file position is specified by a LONG (32-bit) pointer. The file position pointer is not used or affected by the block I/O calls RDB and WRB.

The read functions return the number of bytes transferred. On read operations, End-of-file is signalled by the returned byte count being zero. Exceptional conditions such as "physical I/O error" or "illegal channel" will cause an ABORT.

MAGIC/L, of course, uses these I/O calls internally as part of its compiler; RDTXT, for instance is called by the MAGIC/L compiler RDLINE function.

The following routines are implemented in all versions of MAGIC/L. Additional related routines may be implemented in specific versions. The error messages are "generic" and the exact text will vary from system to system.

Opening and Closing files

OPEN -- Open a disk file - and return the channel number

```
chan := OPEN ( filename , options )
```

filename	is a string specifying the name of the file to be opened.
options	is an ASCII string which selects various options at OPEN time. Each option is selected by a character (which may be in upper or lower case). Options may be specified in any order. The action taken is determined by the combination of all option characters.

OPTIONS	action
A	OPEN the file for appending
N	New. Create FILENAME, then open it.
C	Create FILENAME if it doesn't exist then open (Correct "file doesn't exist" error)
L	Local open. Close automatically on ABORT.
R	Read. Open for input
W	Write. Open for output

The OPEN routine opens the a file with the specified filename and returns the channel number on which the file is open.

IMPORTANT: Note that MAGIC/L assigns a channel dynamically; the user does not specify the channel number. This channel number is used in all other I/O calls which access this file. The number of channels which can simultaneously be active depends on the implementation, but will be at least eight.

MAGIC/L automatically performs an IOERR on OPEN calls (and on most of the calls that follow). If the user wishes to manually check the error status, the IOERR function should be temporarily replaced with a NO_OP. The following example shows how a test for existence of a file can be implemented:

```
; <logical> := ?EXISTS ( FILENAME )
; returns TRUE if the file exists
DEFINE ?EXISTS INTEGER
        ADDRESS FNAME
  LOCAL
        INTEGER TCHN
  APUSH $IOERR                      ; save current error handler
  $IOERR := BASE NO_OP              ; install NO_OP as handler
  TCHN := OPEN ( FNAME , 'RL )      ; open a file
  APOP                              ; restore error handler
  IF ( SYSER <> -1 )                ; check error code
    CLEAR ?EXISTS                   ; error, return FALSE
  ELSE
    SET ?EXISTS                     ; no error, return TRUE
    CLOSE ( TCHN )                  ; and CLOSE file
  ENDIF
END
```

The OPEN code characters defined here are the basic set common to all implementations. Certain implementations might include other option characters to allow for system dependent options. (N.B. Use of system dependent options renders a MAGIC/L program non-portable. In some cases, however, it is unavoidable. The special options have been chosen so that they are ignored by other systems, usually providing compatibility.)

MAGIC/L has a special facility for closing files automatically on an ABORT. If an "L" (for Local) is added to the OPEN options, a flag is set that indicates that this channel should be closed if an ABORT is taken. The option should always be used when an ABORT prevents the normal CLOSE from executing. The CLOSE is actually performed in GOØ, when the compiler initializes after the ABORT, so the feature will not work if the $GO function is changed. If this feature is desired when a special GO function is used, the routine GOINIT may be invoked. This routine performs the automatic closes and automatically restores any values that may have been APUSHed.

Errors:

```
File does not exist
File already exists
Illegal filename
No More Channels
```

Example:

```
INTEGER CHAN CHAN := OPEN ( 'MYFILE , 'NW )
```

This call sets CHAN to the channel number to be used when accessing the file MYFILE. The "N" option will force a new version of the file to be created. The "W" option specifies that the file is opened for writing. Note that both of the arguments are strings, so that the quote is needed if the strings are specified explicitly in the call.

CLOSE - Close a file

```
CLOSE ( channel )
```

The file which is opened on the specified channel number is CLOSEd. If there is data in a buffer, it will be flushed to the file. The channel number will be reused for later OPENs. On some systems, if the file was just created, CLOSE will cause the file to become "permanent" instead of "tentative." When the BYE command is used, all open files are closed automatically. If a file is open and MAGIC/L is terminated abnormally (e.g. by a trap or system abort), the file buffers may not be flushed, and on those systems where a new file is "tentative," it will be deleted.

Errors:

```
Illegal Channel Number
No file open on Channel number
```

Text Oriented I/O

WRTXT - Write a line of text to a file

WRTXT (chan , buffer , count)

chan	channel number
buffer	Array containing text to be written.
count	The number of bytes to be written.

The text in the BUFFER is output to the file opened on CHAN. 'count' bytes from 'buffer' are output. All line-feed characters are converted to the system dependent line termination sequence.

Notes:

In systems which store both CR and LF in text files (e.g. CP/M, RT-11), more bytes may be written to the output file than were specified by 'count'.

The buffer may include nulls.

Errors:

Illegal Channel Number
No file open on Channel number
File Space Exhausted (disk full)
Line too Long (more than 132 characters on most systems)

RDTXT - Read a line of text from a file

count := RDTXT (chan , buffer)

count	Number of bytes returned into 'buffer'
chan	system channel number
buffer	Array to receive the text

Bytes are read from the file up to and including a line terminating character. The system dependent line terminating character is converted into a line-feed (12 octal, ØA hex) character. The returned 'count' includes the line feed. This means that a "blank line" will return a 'count' of one.

The destination 'buffer' is null terminated. This allows the data in 'buffer' to be treated as a MAGIC/L string.

An end of file is signified by a returned length of zero. Not

all systems have a well defined EOF for text files. Care should be taken that the EOF is properly marked.

Example - Type a file

```
DEFINE TYP
        CHAR FNAME ( 1 )
LOCAL
        INTEGER CHAN
        INTEGER CNT
        CHAR    BUF ( 134. )
  CHAN := OPEN ( FNAME , 'RL )  ; Open FNAME for reading
  BEGIN
    CNT := RDTXT ( CHAN , BUF )
  IF ( CNT )                    ; Read until End of File
    WRTXT ( COCH , BUF , CNT )  ; Print the buffer contents
                                ; COCH is the console channel
  REPEAT
  CLOSE ( CHAN )                ; close the file when done
END
```

Errors:

Illegal Channel Number
No file open on Channel number
Line too Long (more than 132 characters)

Sequential I/O

RDS - Read a specified number of bytes from a file

NREAD := RDS (chan , buffer , nbytes)

nread	actual number of bytes read
chan	system channel number
buffer	array to receive the data
nbytes	maximum number of bytes to be read (less bytes are read if EOF is reached)

Up to NBYTES bytes are read into BUFFER from the file opened on CHAN. The actual number of bytes read is returned in NREAD. If NREAD is zero, end-of-file (EOF) has been encountered. If NREAD is less then NBYTES, the EOF was encountered part way through the record, and only NREAD bytes are valid. On some systems, EOF is only recognized on physical disk-block boundaries. In these systems, a user defined EOF mark should be used.

Errors:

Illegal Channel Number
No file open on Channel number

WRS -- Write a specified number of bytes

WRS (chan , buffer , nbytes)

chan	system channel number
buffer	array containing data to be written
nbytes	number of bytes to be written

The first NBYTES bytes in the BUFFER are written to the file opened on CHAN.

File Positioning Calls

SPOS -- Set the file position pointer

```
SPOS ( chan , lpos )
```

chan	system channel number
lpos	a LONG file pointer specifying the byte in the file to be transferred next

The file position pointer is set to the specified absolute offset into the file. The action taken if LPOS is beyond EOF is system dependent. On some systems, if LPOS is -1L, the file position will be set to EOF.

GPOS -- Get the file position pointer

```
lpos := GPOS ( chan )
```

chan	system channel number
lpos	a LONG file pointer specifying the byte in the file to be transferred next

The file position pointer is returned as a LONG value. Usually this is saved for a later SPOS call.

Block I/O

RDB - Block read

```
NBYTES := RDB ( chan , fblk , buffer , nblocks )
```

nbytes	actual number of bytes read
chan	system channel number
fblk	block number of the first block to be read
buffer	array to receive the data
nblocks	number of 512 byte disk blocks to be read

This routine provides random file read access at the disk block level. Blocks FBLK through (FBLK + NBLOCKS - 1) are read into BUFFER. Blocks are numbered starting from zero and are 512 bytes long. The actual number of bytes read is returned in NBYTES. The file position pointer is not used or affected by RDB.

WRB - Block write

```
WRB ( chan , fblk , buffer , nblocks )
```

chan	system channel number
fblk	block number of the first block to be written
buffer	array to containing the data to be written
nblocks	number of 512 byte disk blocks to be written

This routine provides random file write access at the disk block level. Blocks FBLK through (FBLK + NBLOCKS - 1) are written from BUFFER. Blocks are numbered starting from zero and are 512 bytes long. The file position pointer is not used or affected by WRB.

[This page left blank]

Compiler Error Handling

MAGIC/L has a comprehensive error detection and reporting facility that aids program development. When an error is encountered during compilation, the type of error, line number, the input line, and the token on which the error was encountered are all displayed. For instance, the line:

```
IF ( 1 ) END
```

will generate the following message:

```
Error: Syntax - IF matched by END
 Token: END
 ln 1:   IF ( 1 ) END
```

An error will terminate compilation on the following conditions:

1. Syntax errors such as mismatched keywords.
2. I/O errors, such as "File does not exist."

On many types of errors, the compilation can continue, but a special word is compiled to prevent execution of the erroneous code. In these cases the message will indicate "Warning" instead of "Error" at the time that the code is compiled. An "Error" message will be given on any attempt to execute the code.

Definitions that do not contain an error (or call definitions containing an error) will be compiled properly and may be executed, even if other parts of the program contain errors. Thus, some errors can be ignored while other parts of the program are tested.

Certain assemblers in MAGIC/L (and possibly future versions of the compiler) will attempt to generate the "best guess" code, but Warnings should normally be considered "delayed errors."

Note: Since a "Warning" will eventually result in an "Error" and an ABORT, if a "Warning" is detected on input from the keyboard an ABORT is taken IMMEDIATELY.

Redefining Words

Whenever a word is defined with a name that conflicts with a previous definition, the diagnostic message:

```
Redefined: aaaaa
```

is issued (aaaaa stands for the name being redefined). The previous definition continues to be used in the all prior definitions that called it, but the new definition will be used for all future references.

While it is legal to redefine any word, including the "Key Words" that make up the MAGIC/L syntax, it is undesirable to do so, since it diminishes program readability. Sometimes, a word is redefined with enhanced functionality, using the old definition as part of the new definition:

```
DEFINE BYE
  PRINT "Terminating program at " , #Z
  TIME
  BYE   ; this is the old BYE, not recursion!
END
```

Variable names which appear within the argument or LOCAL definition area of a MAGIC/L routine may be reused as local names within other routines without conflict. If however a local name is in conflict with a previously defined global name, then the global definition may not be referenced by the routine using the local name.

Undefined Words

When a word is found that is not previously defined and is not a legal numeric or string literal a warning will be displayed on the terminal (and the bell will ring), but compilation will continue with the next definition.

In addition to the warning message being given, a fatal error routine is compiled. When the routine containing the undefined word is executed, a fatal error is taken when (and if) the undefined word is encountered:

```
Error: Attempt to execute undefined word
```

This two stage process allows compilation of source modules to continue even though the code has errors. If the undefined word is not within a DEFINE structure, then of course the compiled code is executed immediately making it appear as though there are two errors when an undefined word is entered.

Undefined words are assumed to be of type INTEGER. This may cause additional mixed-mode warnings if it is encountered in

an expression that uses other data-types.

Mixed Mode Warnings

A Mixed Mode warning is issued whenever an operator or assignment statement encounters operands of different types. The warning message includes a display of the line of code in question. Because of the potentially dangerous possibilities of executing mixed mode code, a fatal error is compiled as in the case of undefined words.

Syntax Errors

Most syntactical structures in MAGIC/L require keyword pairs; one keyword to open the structure, another to close it. When a closing keyword (such as ENDIF or LOOP) is encountered, a check is made to see if the most recent opening keyword is the proper mate. If it is not, a fatal compile error is given which states that the closing keyword was not properly matched. For example in the structure:

```
DO 5 , 1Ø
   .
  IF ( ... )
    .
LOOP
```

the error message "Syntax - IF matched by LOOP" is given since the opening keyword immediately preceding the LOOP was not a DO (in this case it was an IF). If the closing keyword takes arguments, such as UNTIL (...), the erroneous token will be displayed as the close parenthesis, because it is at that time that the keywords are checked.

I/O Errors

I/O errors are normally associated with user written code that calls various system resources. MAGIC/L, however, uses the same calls that are available to the user, so I/O errors may occur during compilation. The messages displayed for I/O errors are similar to syntax errors, except the line and token are not displayed. I/O errors will cause an ABORT and halt processing.

I/O errors are checked for by the routine IOERR. Whenever a system call is performed, the integer variable SYSER will be set to -1 if there is no error, or to the appropriate system error code, which will always be positive. IOERR checks SYSER and triggers the error message if it is not -1. Most high level calls in the MAGIC/L I/O package execute IOERR immediately following the system call, and the user should

follow this convention when appropriate.

IOERR is a user definable routine, like RESTART and ABORT (see chapter 12). The execution vector is maintained in the variable $IOERR and the default routine is IOERRØ. This allows the user to intercept errors and process them in a special way. One way to do this is to set $IOERR to NO_OP, and then manually check SYSER:

```
APUSH $IOERR                ; save current error handler
$IOERR := BASE NO_OP        ; install NO_OP as handler
CHAN := OPEN ( 'MYFILE , 'RW )  ; open a file
APOP                        ; restore error handler
IF ( SYSER <> -1 )          ; check error code
   ... error handling code ...
```

The error handling may also be in a separate definition that is installed in $IOERR. In either case, it is important that the normal handler, IOERRØ, be restored during normal operations, because many internal routines rely on IOERR to ABORT on errors.

The Error Message File

The error messages for MAGIC/L are contained in a file named MGL.ER. If the file is not present when MAGIC/L is initialized, the messages will be displayed as code numbers. The pathname of the error message file is maintained in the string ERPATH. If for some reason the name (or directory, disk, extension, etc.) must be changed, s SAVE must be performed. The file will be opened properly when the new program is run.

The error messages may be displayed be the use of the Get Error Message routine, GERR:

```
GERR ( <code> , <flag> , <buffer> )
```

where <code> is either the systen error code or MAGIC/L error code; <flag> is -1 to indicate system code, Ø to indicate MAGIC/L code; and <buffer> is an array of at least 8Ø bytes to receive the message. If the code is unknown, a string containing the number is returned. The following code will display all the MAGIC/L error message:

```
CHAR EBUF ( 8Ø. )
ITER 2Ø5k          ; highest code is 2Ø4k (see below)
  GERR ( I , Ø , EBUF )
  PRINT "Error code: " , #I 4 , I , "  " , EBUF
LOOP
```

Invoking an Error Condition

Users may invoke the error handling system by calling ERR or WARN with an error code as an argument. These are the routines used by the compiler when errors are reported. Both routines call GERR to report the error message, and then display the line and token currently active. The difference between ERR and WARN is that ERR always calls ABORT, while WARN only calls ABORT if ICH is -1 (input from console). Thus, ERR should be used to report fatal conditions, while WARN should be used to report possibly recoverable errors.

Runtime Stack Errors

MAGIC/L maintains a variety of internals stacks, buffers and pointers. These are normally invisible to the user until an unfamiliar error message appears. The message should be informative enough to indicate the cause of the error. It should be noted that all of the stacks are of "finite" size, and while they are large enough for most normal applications, they are not suited for deeply nested recursion.

It is important to note that these stacks and buffers are not checked every time they are used. Checks are performed as each line is read at compile time and while working at the keyboard, but are suppressed at runtime to increase throughput. The user may force a check of all the stacks at runtime by executing the word ERRCHK. This word should placed in inner loops when debugging obscure problems.

Listing of MAGIC/L Error Messages

Following is a list of all the error messages generating by MAGIC/L. (Not all codes are generated by all versions.) The code is listed in octal. This list does not include system error messages. The system error messages are normally the same as the standard error messages for a given system.

Ø	Syntax - Missing BEGIN for UNTIL/FOREVER
1	Syntax - Missing IF for REPEAT
2	Syntax - Missing BEGIN for BEGIN-IF-REPEAT
3	Syntax - Missing IF for ELSE
4	Syntax - Missing IF for ENDIF
5	Syntax - Missing DO/ITER for LOOP
6	Syntax - Missing DEFINE for END
2Ø	Syntax - Unmatched parentheses
21	Unexpected EOF
22	Insufficient memory
23	Syntax - Assignment within expression
24	Unsupported operator for data-type
25	Unknown data-type
26	Mixed-mode operation
27	Syntax - Not within RECORD definition
3Ø	PTR Function requires argument(s)
31	Syntax - Missing name field
32	Undefined word
33	Attempt to execute undefined word
34	Attempt to execute illegal operator
35	Attempt to execute a mixed mode expression
41	Stack overflow
4Ø	Stack underflow
42	V-stack overflow
43	V-stack underflow
44	A-stack overflow
45	A-stack underflow
46	L-stack overflow
47	L-stack underflow
51	R-stack overflow
5Ø	R-stack underflow
52	Compile buffer overflow
53	Compile buffer underflow
6Ø	Illegal colon specification
61	Not a RECORD structure name
7Ø	Illegal #F format
71	Error on Input
72	Numeric OPEN codes not supported
73	Operator not within expression
74	Inconsistent Revision

100	Address error
101	Missing operand
102	Too many operands
103	Wrong number of operands
104	Illegal addressing mode
105	Multiply defined label
106	Illegal register number
107	Questionable syntax
110	Displacement Overflow
111	Illegal Reference Class
120	Syntax - IF matched by UNTIL/FOREVER
122	Syntax - IF matched by IF-REPEAT
125	Syntax - IF matched by LOOP
126	Syntax - IF matched by END
130	Syntax - ELSE matched by UNTIL/FOREVER
131	Syntax - ELSE matched by REPEAT
132	Syntax - ELSE matched by IF-REPEAT
133	Syntax - ELSE matched by ELSE
135	Syntax - ELSE matched by LOOP
136	Syntax - ELSE matched by END
141	Syntax - BEGIN matched by REPEAT
143	Syntax - BEGIN matched by ELSE
144	Syntax - BEGIN matched by ENDIF
145	Syntax - BEGIN matched by LOOP
146	Syntax - BEGIN matched by END
150	Syntax - DO/ITER matched by UNTIL/FOREVER
151	Syntax - DO/ITER matched by REPEAT
152	Syntax - DO/ITER matched by IF-REPEAT
153	Syntax - DO/ITER matched by ELSE
154	Syntax - DO/ITER matched by ENDIF
156	Syntax - DO/ITER matched by END
170	Floating Point Underflow
171	Floating Point Overflow
172	Division by zero
173	Square root of negative number
174	Log of negative number
175	Illegal inverse trig value
200	No more channels available
201	Illegal file name
202	Channel not open
203	Illegal channel number
204	Illegal I/O for open

[This page left blank]

Appendix D Hints on Debugging MAGIC/L

This informal appendix is an attempt to clarify certain obscure points in the MAGIC/L syntax and to describe various debugging techniques. Many of the comments stem from questions asked by novice users, others from our own experience. It is recommended that this be reviewed at moments of frustration (it may even save a phone call!).

Compile time bugs

MAGIC/L does a rather good job of reporting errors detected at compile time. The latest revisions (Rev 2.2Ø and later) remove most of the ambiguities of the error reporting.

Remember that many compile time errors are initially reported as warnings, and compilation continues. These bugs must be fixed eventually, but it is possible to test part of a program while avoided to the code that contains errors. "Redefining" errors are often benign, but they should not be ignored, since redefining a keyword like "define" is inconvenient.

The most common compile time problem is running out of compile-buffer space (definition too large) or memory space (program too large). Usually the appropriate messages are given, but not always. This problem is discussed below in the section on Stack and Buffer Errors.

It is possible that compile time failures are actually caused by damage to the compiler code by some previously executed function. Problems of this type are discussed below.

On rare occasions, VERY bad syntax will cause a fatal error error at compile time. MAGIC/L may be so damaged that the error message does not appear. To isolate the code that caused the problem the line

```
LICH := COCH
```

placed before the suspect code will cause all compiler input to be echoed on the terminal. The logging may be suppressed later by:

```
SET LICH
```

Debugging Runtime Errors

Runtime errors may be divided into two general types, those that simply give the wrong answer, and those that cause serious damage to MAGIC/L. The first type are called "Algorithm Errors" because they are usually caused by syntactically correct code that does the wrong thing.

Algorithm errors are very easy to debug in MAGIC/L. Each subroutine may be executed individually and the global variables inspected for anomalies. In fact, LOCAL variables should be initially coded as global variables to enhance this capability. DO LOOPs may be set up to execute functions with differing input while the output is printed. Since the source module may be recompiled almost instantly, it is rather painless to insert PRINT statements in the code.

If you are writing diagnostic routines at the keyboard, remember to use LOGON or LOGINP to save these routines, so that you don't have to reenter them repeatedly.

The important thing to remember is that MAGIC/L provides a virtually unlimited number of ways to debug errors. The hardest thing for a novice MAGIC/L programmer to remember is that MAGIC/L is an extremely powerful debugger.

A much more insidious type of bug is caused by a routine that modifies code or an internal data base. Often the symptoms of these bugs are delayed, so there is no obvious cause of the problem. MAGIC/L may be crippled to the point where it cannot serve as a debugger. This type of bug is almost always caused by subscripting errors, block data movement errors, or misuse of pointers.

Subscript Errors

There is no subscript checking in MAGIC/L. One of the more common mistakes for beginners (especially ex-Fortran programmers) is that arrays always begin subscipting at zero. The following sequence has been tried often (but it still doesn't work!):

```
INTEGER ARR ( 10 )
DO 1 10
  ARR ( I ) := ...
```

The DO statement should be either DO 0 9 or ITER 10.

Subscript errors of this type usually destroy the definition or variable following the array. If execution of a word causes a failure, an array preceding it would be suspect.

Block Data Movement

The easiest way to destroy memory is with block data movement calls. The following routines are always suspect:

MVWDS	RMVWDS
MVBYTES	RMVBYTES
MVSTR	CONCAT

Remember that the reverse moves and the forward moves take the same arguments (the pointers always point to the low end of the arrays). CONCAT should not be used to concatenate to a string literal. When MVZER or MVWDS are used on CHAR arrays, remember to divide the declared size by two.

Various I/O calls such as RDB and RDS are effectively block data movement routines. The byte count for reading in records should be parameterized in terms of the SIZE of the record.

Argument Passing

Passing the correct number of arguments is the user's responsibility. No attempt is made to count arguments except in COMMANDs. Passing too many arguments usually causes stack full errors if the code is executed repeatedly, but passing too few will feed random values into a subroutine on the first call. Framed routines (those that declare input, output, or locals, and all COMMANDs) will force the stack pointer to the correct value upon exit. This makes MAGIC/L more stable, but can hide bugs that will reappear when code is modified. Although this type of bug might seem to be very common, they are virtually eliminated by testing each section of code as it is developed. The routine ERRCHK can be inserted in the code to check for stack faults. ERRCHK should be used whenever bizarre symptoms appear, since a run-away stack may damage other internal databases.

Recursion

Recursion can cause certain problems in MAGIC/L. Although it is quite possible to generate recursive programs in MAGIC/L, care should be taken that the nesting depth is not too deep, because all of the various stacks are of rather finite size. If deep nesting is expected, the ERRCHK routine should be used. In most cases, recursion is the least efficient way to solve a problem.

A special case of recursion occurs in the user definable functions such as $RESTART and $ABORT. It is easy to see that a $ABORT function should not execute ABORT, but it is equally important not to execute any routine that might cause an ABORT, such as ERRCHK or I/O. The "Bug of the Month" award

has been won several times by $ABORT routines that insisted on calling themselves.

Instant Commands

Some confusion has been caused by the usage of the "Instant Commands," namely:

```
BASE    SIZE    SIZEW   $O
ASCII   NOPARSE
```

These words are compile-time routines that "intercept" the following word from the input stream and processes it in a special way. Thus, none of these routines can handle syntax such as parenthesis or operators. For instance:

```
ASCII ( A )
```

will compile a 50k ("ASCII ("), the "A" will likely be undefined, and the ")" will cause an error. Except for NOPARSE, all of the Instant Commands result in the compilation of an integer, and are functionally equivalent to an integer literal or parameter.

BASE and &

The Instant Command BASE and the operator & might seem to have the same function, but they are quite different. BASE is ALWAYS used in conjunction with functions and routines when the "Execution Vector" is desired. This value usually becomes an argument to EXEC. The & operator is ALWAYS used with variables, so that they will be evaluated as addresses, rather then values. Two exceptions arise, BASE is often used as a keyboard diagnostic to determine the existence or memory location of a routine, as in:

```
PRINT BASE FOO , BASE GOO
```

& has the special purpose of returning the address where a record type stores the address of the currently active record.

Commas in MAGIC/L

The usage of commas in MAGIC/L is slightly inconsistent. Commas MUST be used to separate expressions in the PRINT and INPUT commands. This has to do with the manner in which PRINT and INPUT determine the type of the expressions. Commas are ignored in variable declaration lines and in PARSED commands. Commas are not required in argument lists unless one or more of the arguments is an expression. Lack of commas to separate expressions may cause anomalous results. It is strongly recommended that commas always be used in "formal coding" both to enhance readability and to allow a planned Syntax Checking Program (ala LINT in C) to fully analyze argument lists. There is one place in MAGIC/L where a comma MUST NOT be inserted: the PRINT directive #F (for specifying the field for floating point output) takes two arguments which cannot be separated by a comma.

Meaningless Statements

There are virtually limitless number of meaningless MAGIC/L statements. Most of these fall into the category of placing numbers on the stack and never using them. The compiler has no way of detecting this. If X is a variable, the line:

```
mgl> X
```

has no meaning, since MAGIC/L provides no easy means for referencing the number (actually the address of X) that this line will leave on the stack. If done repeatedly, there will be a stack fault. Because of the automatic stack adjustments, this sort of bug is usually quite passive.

The WITH Statement

The WITH statement is used to activate record variables. The action of WITH is entirely at runtime, not at compile time as with some other languages. The action is also GLOBAL and not LOCAL to a subroutine. IF a "LOCAL WITH" is desired, either the APUSH facility should be used, or the colon structure.

The action of WITH at compile time is curious. It merely sets a flag so that references to record variables on that line will be compiled in the WITH form. Other words on the WITH line be compiled in the normal manner. This includes the "meaningless syntax" described above. Thus, the line:

```
WITH X
```

will have no effect if X is an intrinsic variable (i.e. INTEGER or LONG etc.) instead of a record variable.

Compile-time versus Run-time

The difference between compile-time and run-time can be confusing to beginners. These modes are determined be the context of each line, not whether the line is in a file or entered at the keyboard. For instance, the following sequence the ITER structure is run immediately because it is not in a definition:

```
INTEGER X ( 1Ø. )
ITER 1Ø.
  X ( I ) := I * I      ; fill array with squares
LOOP
```

Another example of this concerns changing the radix. In the lines:

```
OCTAL
X := 25
```

X will be set to 25 octal. But if the same lines are in a definition:

```
DEFINE FOO
  OCTAL
  X := 25
  ...
```

X will be set to 25 decimal (assuming decimal is the current radix at compile-time). The OCTAL command will only affect number conversion following the EXECUTION of FOO. To avoid this problem, it is wise to always be explicit when specifying numeric literals (i.e. use 25. or 25k).

Stack and Buffer Errors

MAGIC/L maintains several internal stacks that may underflow or overflow due to errors and memory limitations. Although these stacks are normally invisible to the user (except the Parameter Stack, or "The Stack," which is the stack which the user references from assembly language) knowledge of their usage is helpful in debugging. Note that the usage of these stacks is subject to change in future revisions of MAGIC/L.

RETURN Stack

Used for return addresses and threaded code stacking. Underflow is unlikely, overflow caused by run-away recursion.

LOOP Stack

Used for DO-LOOP nesting and channel number nesting (by LOAD or EXT). Underflow is unlikely, overflow is caused by nesting

LOOPs and LOADs too deeply.

AUX Stack

Used for APUSH and APOP. Since this stack is used explicitly, stack faults are caused by user errors. This stack is automatically "unwound" on ABORTs.

Parameter Stack

The main argument passing stack. Stack faults are normally caused by improper subroutine linkage. The user is responsible for passing the proper number of arguments. Note that routines that declare arguments or LOCAL variables will always set the stack according to its specifications, thus coding errors may be unnoticed until code is modified.

The DICTIONARY (free memory space)

Although not really a stack, the dictionary space grows upward though memory. This area contains both code and data for the program and is allowed to grow until memory is exhausted, at which time the message "INSUFFICIENT MEMORY" is issued. The symbol table also occupies this space, growing from high memory down.

Two system variables .D and MEMORY specify the boundaries of dictionary space. The variable .D contains the address of the lowest free memory location. It is modified whenever a new routine or variable is defined. The variable MEMORY contains the address of the highest free memory location. It is modified whenever more space is required for the symbol table or I/O and module buffers.

In spite of all our efforts, it is still possible for a program to crash while trying to report "Insufficient Memory."

Compile Buffer

This space is used to hold the compiled code before it is transferred to the dictionary space or executed. If a definition is too long, a fault occurs. In the current implementation, the compile buffer is large enough to hold about forty lines of code. If a compile buffer full error occurs, or compilation dies in a large definition, break the definition into several parts and try again. Note that string literals always pass through the compile buffer, so definitions with long strings are the most likely to cause problems.

[This page left blank]

Space and speed will not be of major concern to the novice MAGIC/L user. MAGIC/L code will run as fast as code generated by many compilers and is much faster than BASIC code. MAGIC/L code is also quite compact so that most applications can be performed in less than a thousand bytes.

Eventually, however, each programmer will want a program to run faster or take less space. The most important factor, of course, is proper programmer style. (There is no limit to how poorly a program can be written.) We will not attempt to teach proper style here, we can give the information needed to optimize a MAGIC/L program.

Space Efficiency

MAGIC/L is inherently very space efficient. Although initially a lot of space is taken by MAGIC/L, this space consists largely of library routines that would be used by most large programs. Thus the primary advice for saving space is to take advantage of all of the built in functions!

Since MAGIC/L code is compact, most space problems are caused by large data arrays. Often large amounts of space may be saved by using a single "work array" for many purposes. If it is defined as a RECORD, the same space can be given several different names or broken down into several sub-arrays.

Large arrays may be declared as LOCALs for dynamic allocation. Remember, however, that when calls to routines are nested the LOCAL space is also nested. Care should be taken that the nested LOCAL space does not exceed available memory.

After the large array space has been minimized, there is usually no obvious way of gaining thousands of bytes of space. There are, however, a large number of ways to gain small amounts of space, which can easily add up to several thousand bytes.

The following table shows the space usage of various type of variables or routines. Each symbol name will also take four words in the symbol table, but this can be recovered using the symbol table editing program.

INTEGER and CHAR	3 words
LONG and REAL	4 words
Variables in records	three words plus one ADDRESS
Record Types	2 words plus one ADDRESS
Record Variables	2 words plus one ADDRESS plus SIZEW
Simple Routines	2 words plus code
Routines with Arguments or LOCALS	4 words plus code
MAKEs	1 word plus parameter list
COMMANDS and PARSED	3 words plus code

The code generated for a routine is easy to calculate. Almost all invocations of variables, routines or operators take one word with the following exceptions:

- Punctuation (commas, parens) use no space.
- Numeric literals require one extra word beyond the size of the literal.
- String literals require two extra words beyond the size of the literal.
- PARAMETERs take two words (same as INTEGER literals)
- Instant commands (BASE, SIZE, etc.) take two words.
- The numbers -1, Ø, 1 and 2 only take one word.
- Unsubscripted arrays take two words.
- Most control words take two words, some take none (ENDIF and BEGIN).
- COMMANDs use two words (one more than normal)
- NXTARG uses two extra words if there is no argument
- & uses no extra space.
- WITH uses no extra space.

It is easy to determine the optimal size for a routine, depending on how many times it is called. For instance, the routine:

```
DEFINE CLEARX
  CLEAR X
END
```

requires only four words to create (two words for a simple definition, two words of code). CLEARX uses one word each time its invoked, as opposed to two words to explicitly say "CLEAR X". Thus CLEARX will begin to save space on its fifth usage. Routines that are more complex will usually pay if they are used two or three times. Conversely, nothing is gained by making a routine longer than 1Ø to 4Ø lines (except difficulty in debugging). Breaking an unwieldy routine into two or three parts costs very little, and can aid debugging considerably.

A study of the table produces a few other results:

Use INTEGERs rather than PARAMETERs
PARAMETERs are very useful for documentation. However, if one

parameter is used often, it should be coded as an INTEGER. INTEGERs cost a few extra words to define but save a word on each usage.

Use MAKEs

The ACTION ... MAKE construct can save large amounts of code space and requires very little overhead. Chapter 10 shows how even trivial case can be coded more efficiently with this facility.

Use the Special Assignment Operators

INCREMENT and DECREMENT save three words each time they are used; += and -= save two words; and CLEAR and SET save one one word when used. They were provided mainly to save space. The savings are greater when subscripting or the ":" operator are involved.

Use Logical Expressions

Novice MAGIC/L programmers tend to forget that logical expressions can be used in assignment statements as well as conditionals. For example:

```
DEFINE XGTY INTEGER
  IF ( X > Y )
    SET XGTY
  ELSE
    CLEAR XGTY
  ENDIF
END
```

can be coded with a savings of six words:

```
DEFINE XGTY INTEGER
  XGTY := X > Y
END
```

Use LOCAL Variables

Although global variables simplify debugging, they waste space, both in code and in symbol table space. Each global integer takes three words of code space, while locals usually require no additional space. The code space for locals is shared by all instances of the same type of variable in the same position in the LOCAL definition. About 30 such locals are predefined and their usage is therefore free. You should avoid a local list with an "odd" configuration such as a large array followed by a long list of integers; placing the integers first will save space.

Use Global Variables to Pass Arguments

Although global variables cost space, passing the same argument repeatedly to a subroutine can cost more. If a primitive routine is called with the same argument list more then three or four times, it often pays both in space and speed to use globals to pass arguments.

Use Initialized Data Bases

As explained in chapter 9, initializing data bases uses no space (or time) at runtime. In particular, strings that are used repeatedly should be created with .TEXT rather than being used as inline literals.

Remember that inline literals always take an extra word of overhead (two for strings). The following stretch of code sends a series of line positions to a vector display screen:

```
VECTOR ( 100 , 100 , 100 , 200 )
VECTOR ( 100 , 200 , 200 , 200 )
VECTOR ( 200 , 200 , 200 , 100 )
VECTOR ( 200 , 100 , 100 , 100 )
```

This can be coded as:

```
INTEGER VECS ( 0 )
        .WORD 100 , 100 , 100 , 200
        .WORD 100 , 200 , 200 , 200
        .WORD 200 , 200 , 200 , 100
        .WORD 200 , 100 , 100 , 100
ITER 16
  VECTOR ( VECS ( I ) , VEC ( I + 1 ) , ^
      VECS ( I + 2 ) , VECS ( I + 3 ) )
LOOP ( 4 )
```

Even with the added subscripting and looping overhead, this saves 10 words. Savings are greater if there are more items, if the items are referenced by other routines, or if RECORD structures are used to simplify the addressing. In addition, this form is easier to debug since the table may be accessed from the keyboard at runtime.

Use "Formatting" Functions

The PRINT routine can be costly, especially if complicated formatting is involved. Several consecutive PRINTs should be combined, if possible. Some programs will use the same format repeatedly. In these cases a large amount of space can be gained with a routine such as:

```
DEFINE OUTVAL
        ADDRESS MESS
        INTEGER VAL
  PRINT STR ( MESS ) , ":" , #R 10. , #UI 5 , VAL
END
```

This can save about six words for each usage.

Speed Efficiency

As with the space considerations, it is easy to determine the approximate time of a routine in MAGIC/L. Each "low level" word, such as operators, variables, entering and exiting a high level routine, or any simple assembly code entry, takes approximately the same amount of time. This may seem rather simplistic, but an experienced MAGIC/L programmer can usually determine the execution time of a routine to within a factor of two by inspection. This "cycle time" will be determined by the hardware used, but is in the range of five to twenty microseconds for most machines.

There are several exceptions to this rule. Multiplies and divides are costly on some machines, and software floating point is time consuming. Block data and string routines take time proportional to the data size. I/O routines and other systems calls can be very costly.

A non-obvious speed cost involves subroutines that pass arguments or declare locals. The time involved in entering and leaving such routines is about eight cycles, as opposed to two cycles for the simplest routines. The means that the optimal size for a routine for speed considerations is not as small as would be implied by space considerations. One line routines that pass arguments should be avoided in time critical code.

If a program runs too slowly the first thing to do is to isolate the "bottleneck." Most programs spend 95% of their time in 5% of the code. Informal benchmarks should be performed as the program is developed so that bottlenecks are isolated before they become too integrated into the program to be fixed.

The bottleneck will usually occur within an iteration loop (or some other looping construct). Within such an inner loop any time-consuming routine should be avoided, particularly passing arguments to high level routines. Time is saved here by coding explicitly, rather than calling routines.

When all else fails, assembly language can be used to gain speed. Certain routines, such as block data manipulations, are particularly well suited to assembly language coding. The original high level code should be commented out to serve as documentation and to aid in porting. Most of the MAGIC/L operators and library routines are written in this manner.

[This page left blank]

BRIEF LISTING of MAGIC/L SYNTAX and FUNCTIONS

All values are signed 16-bit integers unless otherwise stated.
Type ADDRESS is the same as INTEGER for 16-bit addressing machines, LONG for 32-bit addressing machines.

[...] implies optional arguments.

"..." inplies string literal (with quotes) or an array that contains a string. A string literal with no spaces requires only a leading single quote ('no_space).

<condition> implies logical value (-1 is TRUE, Ø is FALSE).
Logical output is TRUE if <condition> is met.
Non-zero input to a conditional is taken as TRUE.

MAGIC/L is case insensitive except for string comparisions.

--- PUNCTUATION ---

^	line continuation
;;	line separater
(	begin expression or argument list
)	end expression or argument list
,	separate arguments (required in PRINT and INPUT)
:	explicit record base specifier
;	ignore rest of line (comment)
<*	begin comment field
*>	end comment field (should be at end of line)

--- NUMERIC LITERALS and the RADIX ---

The default input and output radix is contained in RADIX. This variable may be set by: OCTAL, HEX or DECIMAL. For literal input the current RADIX may be overidden by appending "." for decimal, "K" for octal, "X" for hex. Hex literals must begin with a digit (i.e. ØFFFFX). Longs are specified either by a trailing "L" (must follow ".", "K", or "X") or a value greater than 65535. Reals are always input and output in decimal.

--- OPERATORS ---

Assignment Operators

:=	+=	-=

Binary Operators

+	-	*	/	OR	XOR	AND

Binary Relational Operators

==	<>	<	>	<=	>=

Unary Relational Operators

==Ø	<>Ø	<Ø	>Ø	<=Ø	>=Ø

Unary Operators

-	NOT	&

INCREMENT variable
DECREMENT variable
CLEAR variable
SET variable

```
        --- CONTROL STRUCTURES ---
BEGIN   ...   UNTIL ( <condition_is_true> )
BEGIN   ...   IF ( <condition_is_true> )     ...   REPEAT
BEGIN   ...   FOREVER
WHILE ( <condition_is_true> )  ...   REPEAT
IF ( <condition_is_true> )     ...   ENDIF
IF ( <condition_is_true> )     ...   ELSE   ...   ENDIF

CASE ( function_number )
  function0 , function1 , ...
ENDIF

        --- DO LOOP ROUTINES ---
DO    low_value , high_value
UDO   unsigned_low_value , unsigned_high_value
ITER  iteration_count
UITER unsigned_iteration_count
LOOP  [( increment_value )]

I       innermost counter
J       second innermost counter
K       third innermost counter
I'      innermost reverse counter
J'      second innermost reverse counter
K'      third innermost reverse counter
EXIT    force last iteration of innermost LOOP
        (I, I' are undefined after EXIT)

        --- INTRINSIC DATA TYPE DECLARATION ---
CHAR    variable_name [( array_size )] ...
INTEGER variable_name [( array_size )] ...
LONG    variable_name [( array_size )] ...
REAL    variable_name [( array_size )] ...
ADDRESS variable_name [( array_size )] ...

        --- RECORD DEFINITIONS ---
RECORD record_type_name
.EVEN
DUMMY   word_count
OFFSET  word_offset
BOFFSET byte_offset
ENDRECORD

        --- RECORD DECLARATION ---
RECORD_TYPE_NAME variable_name [( array_size )] ...

        --- RECORD ACTIVATION ---
WITH variable_name ...
& record_type_name := address
```

```
        --- PARAMETERS, LABELS ---
PARAMETER name := integer
LABEL name

        --- GLOBAL MEMORY ALLOCATION ---
.BYTE byte ...
.WORD integer ...
.LONG long  ...
.ADDR address ...
.BLKB byte_count
.BLKW word_count
.BLKA address_count
.TEXT "string"

        --- DEFINITION TYPES ---
ACTION          action type for MAKEs
COMMAND         call does not require parens; allows argument counting
PARSED          same as COMMAND, but for string input
INTEGER         integer function
LONG            long function
REAL            real function
ADDRESS         equivalent to address type of implementation

        --- DEFINITION STRUCTURE ---
DEFINE routine_name [definition_type]
     [input_argument_specification]
  [LOCAL
     local_variable_specification]
  ... code ...
END

RETURN          force immediate exit from within a definition

        --- SPECIAL WORDS FOR COMMANDS or PARSED COMMANDS ---
NXTARG [( word_count )]         iterate through arguments
CMDCNT                          word count of arguments
CHKCNT ( word_count )           check input count
NOPARSE parsed_command "arguments" ...  allows array input to PARSED

        --- MAKE-ACTION ---
BUILD ( "routine_name" , action_type )  parameters are added
                                        with .WORD etc.
MAKE "routine_name" action_type integer_parameter ...
High level action routines receive a pointer to the parameters
as the LAST argument.  Assembly action routine are passed a
pointer in register "W".

        --- INSTANT COMMANDS ---
Arguments to Instant Commands must be a single word.
$O record_element       compiles integer offset of element
SIZE record_type        compiles record size in bytes
SIZEW record_type       compiles record size in words
BASE entry_name         compiles execution vector
ASCII character         compiles integer value of character
INSTANT entry_name      delays execution of keywords
```

```
        --- BLOCK MOVE ROUTINES ---
FILL      ( address , word_count , value )
MVZER     ( address , word_count )
MVWDS     ( source_address , dest_address , word_count )
RMVWDS    ( source_address , dest_address , word_count )
MVBYTES   ( source_address , dest_address , byte_count )
RMVBYTES  ( source_address , dest_address , byte_count )

        --- SHIFTING FUNCTIONS ---
left_shifted_integer           := LSHIFT ( integer , shift_count )
right_shifted_integer          := RSHIFT ( integer , shift_count )
unsigned_right_shifted_integer := URSHIFT ( integer , shift_count )
left_shifted_long              := LLSHIFT ( long , shift_count )
right_shifted_long             := LRSHIFT ( long , shift_count )
unsigned_right_shifted_long    := LURSHIFT ( long , shift_count )

        --- BIT FUNCTIONS ---
Notes: Bit numbering is system dependent. The offset can be 65535.
<bit_set> := GETBIT ( bit_offset , address )
SETBIT ( bit_offset , address )
CLRBIT ( bit_offset , address )

        --- INTEGER LIBRARY ---
<within_limits> := CLM  ( value , low_limit , High_limit )
<vectors_equal> := VEQ  ( address , address , word_count )
minimum_value   := MIN  ( integer , integer )
maximum_value   := MAX  ( integer , integer )
minimum_value   := UMIN ( unsigned_integer , unsigned_integer )
maximum_value   := UMAX ( unsigned_integer , unsigned_integer )
remainder       := MOD  ( dividend , divisor )
absolute_value  := ABS  ( integer )
<v1_gt_v2>      := UGT  ( v1 , v2 )     unsigned comparison

        --- PACKING and UNPACKING FUNCTIONS ---
packed_byte     := NIBBYTE ( ms_nibble , ls_nibble )
packed_integer  := BYTEWD ( ms_byte , ls_byte )
packed_long     := WDLONG ( ms_integer , ls_integer )
least_significant_byte    := LSBYTE ( integer )
most_significant_byte     := MSBYTE ( integer )
least_significant_nibble  := LSNIB ( byte )
most_significant_nibble   := MSNIB ( byte )
least_significant_integer := LSWORD ( long )
most_significant_integer  := MSWORD ( long )

        --- TYPE CONVERSION ---
long    := UXTND ( unsigned_integer )
long    := XTND ( integer )
long    := LFIX ( real )
integer := FIX ( real )
real    := LFLOAT ( long )
real    := FLOAT ( integer )
string  := STR ( address )      force string type
address := & variable           force pointer mode
```

```
        --- ADDRESS OFFSET ROUTINES ---
address := ADX ( address , integer_offset )
address := SBX ( address , integer_offset )

        --- PEEK and POKE ROUTINES ---
byte    := BPEEK ( address )
integer := PEEK  ( address )
long    := LPEEK ( address )
real    := RPEEK ( address )
address := APEEK ( address )
BPOKE ( byte , address )
POKE  ( integer , address )
LPOKE ( long , address )
RPOKE ( real , address )
APOKE ( address , address )

        --- STRING LIBRARY ---
String literals must not be the destination.
byte_pointer    := BP ( address )  nop on byte addressing machines
byte_count      := LENGTH ( "string" )
relation        := STCMP ( "string" , "string" )  case sensitive
<strings_same>  := STCOM ( "string" , "string" )  case sensitive
offset          := CINDEX ( "string" , char )     case sensitive
MVSTR   ( "string" , address )
$CONCAT ( "dest_string" , address , byte_count )
CONCAT  ( "target_string" , "source_string" )
CLUC    ( "string" )            Convert to upper case in place

        --- PRINT ROUTINES ---
PRINT directive_or_variable , ...
#TYO    ( ascii_value )
#FIELD  ( address , field_specifier , byte_count )
IFPRINT ( column )      output CR if beyond column
ENCODE  ( address )     transfer output buffer to array

        --- PRINT DIRECTIVES ---
#I integer_width                (negative width means left justify,
#UI unsigned_integer_width      positive width means right justify,
#S string_width                 zero means significant chars only)
#R radix
#P pad_character_value
#F field_width fractional_width
#T tab_column                   negative implies relative tab
#A ascii_value                  put single character
#C channel_number               PRINT to a file
#Z              suppress CR
#N              suppress CR and delay output

        --- INPUT ROUTINES ---
INPUT [#C channel ,] variable , variable , ...
#E              suppress error processing
#R radix
```

```
        --- SYSTEM I/O DEFINITIONS ---
channel := OPEN ( "filename" , "option_string" )
         Standard Option Codes
    A   Append. Open the file for appending.
    N   New.    Create FILENAME, then open it.
    C   Create FILENAME if it doesn't exist.
    L   Local.  Close automatically on ABORT.
    R   Read.   Open for input.
    W   Write.  Open for output.
CLOSE ( channel )
byte_count := RDTXT ( channel , address )  count includes LF
byte_count := RDS ( channel , address , byte_count )
byte_count := RDB ( channel , block_number , address , block_count )
WRTXT ( channel , address , count )  LF converted to system CR string
WRS ( channel , address , byte_count )
WRB ( channel , block_number , address , block_count )
long_byte_position := GPOS ( channel )
SPOS ( channel , long_byte_position )
BYE

        --- ADDITIONAL HIGH LEVEL MAGIC/L I/O ROUTINES ---
resolved_channel := SYSOCH      (OCH if not -1, else COCH)
resolved_channel := SYSICH      (ICH if not -1, else CICH)
TSTR ( "string" )
MSG ( "string" )
TYO ( char )
CR

        --- USER DEFINABLE FUNCTIONS ---
These routines are USER definable functions.  They execute the
function contained in $NAME.
<not_end_of_file> := RDLINE     Read a line from ICH into LBUF
<not_end_of_line> := WORD       Parse a word from LBUF to TBUF
<in_symbol_table> := LOOKUP ( "string" )
                                Lookup in active branches
<valid_literal>   := LITERAL    Literal test and translate
ENTER ( "entry_name" )          Enter into current branch
TSTRL ( "string" , length )     Type a string
PROMPT                          Prompt function
ERR  ( error_code )             Error report function
WARN ( error_code )             Warning report function
IOERR                           System error report function
<continue_input> := INPERR      Input error function
UNDEFINED                       Execute for undefined words
LOAD ( "file_name" )            Open file for compilation
ENDFILE                         Close compiled file
GO                              Compiler Main Loop
ABORT                           Force abort processing (stacks
                                  are reset before $ABORT)
$RESTART        Restart function variable (has no high level name)
```

```
        --- SYSTEM VARIABLES ---
ICH             Current Input Channel   (-1 implies CICH)
OCH             Current Output Channel  (-1 implies COCH)
LICH            Log Input Channel       (-1 implies not used)
LOCH            Log Output Channel      (-1 implies not used)
CICH            Console Input Channel   (Standard Input)
COCH            Console Output Channel  (Standard Output)
LINE#           Line in current file
.D              Low limit of free memory
.DØ             Low limit for permanent allocation
MEMORY          Highest address available
CURRENT         Currently active branch
SYSRØ           Returned value from system call
SYSR1             (Usage is System Dependent)
SYSR2             (Usage is System Dependent)
SYSER           Returned error code (-1 is no error)
ERCHN           Error file channel (-1 implies not used)
CBUF            Address of compile buffer
$LBUF           Line buffer pointer   (filled by RDLINE)
$TBUF           Token buffer pointer  (filled by WORD)
$OBUF           Output buffer pointer (filled by PRINT)
INP             Pointer to next word to be parsed
EOC             Up-arrow NOT detected
EOL             Nothing more to parse
CHECK           Count of nesting level during compilation
LASTDELIM       Last delimiter found
COLUMN          ON during RDLINE
#COLUMN         Number of chars in OBUF
INPSTAT         INPUT status
RADIX           Current RADIX
LITFLAG         Result of last literal conversion
MSLVAL          Most significant word of literal
ILVAL           Least significant word
LLVAL           Both words (MSLVAL+ILVAL)
RLVAL           REAL alias of LLVAL
STATBITS        Global MAGIC/L status (cleared at coldstart)
ABOCHAN         Bitmap of "local" files

        --- BUFFERS ---
TBUF            CHAR array at $TBUF
LBUF            CHAR array at $LBUF
OBUF            CHAR array at $OBUF
ERPATH          Error message pathname
XUTIL           Utility directory or extension
PROMSTR         String issued in keyboard prompt
ENAME           Last word defined
```

```
        --- BRANCHES ---
BRANCH ( "branch_name" )          Create a branch
MG/L              Main MAGIC/L branch
USER              Users' branch
BTEMP             Argument and local branch
.END              Close last branch opened

        --- MISC REAL and TRIG ---
fractional_part := FRACT ( real )
absolute_value  := FABS  ( real )
absolute_value  := LABS ( long )
natural_exponential := EXP ( real )
base_2_exponential  := EXP2 ( real )
base_10_exponential := EXP10 ( real )
natural_log := LOG ( real )
base_2_log  := LOG2 ( real )
base_10_log := LOG10 ( real )
square_root := SQRT ( real )
3.1415927   := PI
radians := RAD ( degrees )
degrees := DEG ( radians )
sine    := SIN ( angle_in_radians )
cosine  := COS ( angle_in_radians )
tangent := TAN ( angle_in_radians )
angle_in_radians := ASIN ( sine )
angle_in_radians := ACOS ( cosine )
angle_in_radians := ATAN ( tangent )
angle_in_radians := ATAN2 ( Y , X )  Polar angle of X and Y coordinates

        --- MISCELLANEOUS ROUTINES ---
APUSH variable  Save an integer variable and its address.
APOP            Restore an APUSHed variable.  This must be
                  matched with APUSHes.
APSET ( new_value , & integer )     APUSH and set
ERRCHK          Check all stacks for under/overflow.
EXEC ( arguments , ... , routine_execution_vector )
                Execute a routine.  COMMANDs and PARSED
                  commands should not be used with EXEC.
EXT filename    Compile a file. ".mg" is appended to filename,
                  directory name contained in XUTIL will be
                  searched if the file is not found locally.
                  On some systems, XUTIL is the utility
                  extension.
GOINIT          Clear compiler variables, close Local files,
                  and APOP all APUSHed variables.  This should
                  be part of the ABORT function if $GO is
                  changed.
OERM            Open the error message file.
NO_OP           No operation
REVMSG          Output copyright and revision message.
RECURSE ( ...)  Call the routine currently being defined.
                This take on the properties of the current
                definition, but may not be used in COMMANDs
                or ACTIONs.
RST0            The default startup message.
SAVE filename   Save executable program on disk.  The
                  appropriate file extension is added to
                  the saved file.
```

How to Report Documentation and Software Problems

At Loki Engineering we are constantly striving to improve the quality of our documentation. If you have any trouble locating the information you need, or have found parts of this manual to be unclear or inaccurate, we would appreciate hearing from you.

In the event you encounter a problem in your use of MAGIC/L, please report the difficulty to us by filling in and mailing this form. If possible, the problem should be isolated to a simple case that fails on a standard release version of MAGIC/L. We cannot analyze problems that occur only during execution of large programs.

Product:_____ Version:_____

Reported by:_____ Date:_____

Company:_____ Phone:_____

Address:_____

Computer/Operating system:_____

Comments:_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please mail this form, along with any additional information, to:

Loki Engineering, Inc.
Customer Service Dept.
55 Wheeler Street
Cambridge, MA Ø2138