MAGIC/L[tm]

HIGH LEVEL

PROGRAMMING LANGUAGE

FOR THE

DATAMOVER/68000 BOARD

MTU SYSTEM MANUAL

February, 1984

520-004-0 Rev. A

COPYRIGHT NOTICE

Micro Technology Unlimited
2806 Hillsborough Street
P.O. Box 12106
Raleigh, NC 27605 USA
(919) 833-1458

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

No warranties, either express or implied, are made by Micro Technology Unlimited with respect to this manual or the software described herein, its quality, performance, merchantability, or fitness for any particular application. This product is sold "as is". The buyer assumes all risk as to quality and performance. Under no circumstances will MTU be liable for direct, indirect, incidental or consequential damages resulting from any defect in this software, even if MTU has been advised of the possibility of such damages. Should the software prove defective following the purchase, the buyer assumes the entire cost of all necessary servicing, repair or correction and any incidental, indirect, or consequential damages. Additional rights vary from state to state so some of the above exclusions and limitations may not apply.

NOTICE

Micro Technology Unlimited reserves the right to make changes to the product and specifications described in this manual at any time without notice.

MAGIC/L is a trademark of Loki Engineering, Inc.

TABLE OF CONTENTS

MTU MAGIC/L INTRODUCTION

MAGIC/L is a higher level language which runs on the MTU DATAMOVER board with its 68000 processor. It is interfaced to the MTU system through DMXMON. This means that the MAGIC/L Language is running in the CODOS environment, so learning new operating system principles and commands is not necessary.

MAGIC/L offers three main advantages as a programming language on the MTU system. First, it is a structured language, but still operates interactively. This offers much easier program development and debugging than typical compiled languages. Second, by running on the 68000 processor, the language offers more performance than any other language on the system. Last of all, MAGIC/L offers a much larger amount of free memory for programs and data. Because of these three reasons, MAGIC/L is a very suitable language for serious applications development.

There is more to MAGIC/L on the MTU system than just the language. Included are a number of utility routines and programs as well a library of Integer Graphics routines. The following is a list of the Distribution files which are part of the MTU MAGIC/L Language:

```
FOLD_CMD.C - Patch to fold CODOS commands to upper case. (See Appendix 17.3)
DMXMON.C   - DATAMOVER Cross Monitor program
MGL.E      - MAGIC/L Error Message File
MGL.6      - standard MAGIC/L Language
RMGL.6     - MAGIC/L Language with floating point
BED.M      - MAGIC/L Buffer Editor source file
DICT.6     - Dictionary Utility
SORTREF.6  - Cross Reference Data Output Utility
XREF.M     - Cross Reference Utility source file
IGL.M      - Integer Graphics Library source file
```

Some of those programs above are provided as source files so that you may optionally add them to the MAGIC/L language.

When copying versions of the MAGIC/L to other disks, the MGL.E error file must accompany the version. All of the ".6" files in the list above are versions of MAGIC/L. MGL.6 and RMGL.6 are fully executable versions of the language. DICT.6 and SORTREF.6 are also versions of MAGIC/L, though they have been fixed so that they can only execute their intended application. Thus they all require the presence of the MGL.E file. In addition, if you plan to access the MTU Editor from MAGIC/L (see chapter 7 on USING THE CODOS EDITOR), the presence of DMXMON on the disk is also required.

2. STARTUP

One feature of MAGIC/L is that virtually everything may be changed by the user. The following discussion assumes that the standard startup routines and messages have not been modified by the user, and numbers will be printed in decimal. For details about these routines and what modifications can be made by the user, refer to to chapter 12 in the MAGIC/L Users Manual.

2.1 THE STARTUP COMMAND

The command line processing of CODOS and DMXMON make is possible to startup MAGIC/L from either CODOS or DMXMON. The syntax of the command to startup MAGIC/L from CODOS would be:

```
DMXMON ‹magic/l› [ ‹source file› ]
```

where ‹magic/l› is a SAVEd version of MAGIC/L. This includes MGL.6, RMGL.6, or any SAVEd version to which you may have added some of your own definitions. If the ‹source file› argument is specifed, that file will be automatically compiled before MAGIC/L enters the keyboard input mode. If DMXMON is already running, simply enter the command line above, omitting the "DMXMON".

2.2 STARTUP SIGNON MESSAGES

When MAGIC/L first begins running, it will display a signon message similar to the following:

```
MAGIC/L   Rev 2.50.88    Copyright (c) 1983 Loki Engineering, Inc.
  211052 bytes available.
```

The number of bytes initially available will vary depending on what routines have already been compiled into the language. For example, the program RMGL will show fewer bytes free since a package of floating point routines are included. The revision number displayed will also be different if later versions have been released. After this signon message is printed, MAGIC/L will compile the file specified on the command line, assuming a file name is present.

If a file is specified, MAGIC/L will display the file name, starting location for the code, and the number of bytes free prior the compilation. For example, this message might appear as:

```
MYPROG.M          36596  211052
```

assuming "MYPROG.M" was specified as a source file on the command line.

Once the compilation, if any, is complete, MAGIC/L will display the prompt:

```
mgl›
```

and enter the keyboard input mode. At this point, MAGIC/L is waiting for a complete line to be entered on the keyboard. Since MAGIC/L is using CODOS to input the line, the standard CODOS line editing functions (CNTL-B, etc.) are available. With the MAGIC/L Language now running, operation will be as described in the MAGIC/L Users Manual.

3. EXITING AND REENTERING MAGIC/L

There are two ways to exit MAGIC/L, and two ways to reenter MAGIC/L. Reentering MAGIC/L refers to restarting execution of the MAGIC/L code already present in the DATAMOVER RAM. This assumes that the RAM used by MAGIC/L is still same as it was when MAGIC/L was exited.

3.1 EXITING MAGIC/L

The proper way to terminate MAGIC/L is to execute the BYE command. This will free all MAGIC/L assigned channels and return to DMXMON. This type of exit will also occur automatically if an End-of-File is received while inputting from the console input channel, CICH. This can occur if CICH is assigned to a disk file, or a CNTL-Z is entered on the keyboard as the first character of a new line.

A second way to leave MAGIC/L is to execute an appropriate DMXMON SVC which returns execution to DMXMON or CODOS. It is important to note that exiting MAGIC/L in this manner will leave the MAGIC/L assigned channels still assigned. Thus this is not a proper way to terminate a session with MAGIC/L. However, this type of exit can be useful in certain situations relating to job files. For example, it permits MAGIC/L to startup and return from the MTU EDITor, using a job file.

3.2 REENTERING MAGIC/L

If MAGIC/L is exited via either method above, it may be "cold" started again by executing one of the following commands:

```
GO ‹cold entry point›
or
DMXMON GO ‹cold entry point›
```

The first command is used when you are already in DMXMON. If you are in the CODOS Monitor instead, use the second command. Reloading DMXMON does not affect the MAGIC/L code in the DATAMOVER RAM.

The cold entry point may be found in several ways. It is the address following the "=" in the display from a GETLOC command performed on the MAGIC/L file. This address is also found in the first four bytes of the object and may be displayed with a DUMP command once the object is loaded into memory. And finally, the cold entry point may be displayed from MAGIC/L using the following command:

```
PRINT #R 16. , VTBL ( BASE .ENTER )
```

If MAGIC/L was exited via the second method above, leaving the MAGIC/L channels still assigned, it may be "warm" started again using one of the following commands:

```
GO ‹warm entry point›
or
DMXMON GO ‹warm entry point›
```

The first command is used when you are already in DMXMON, and the second if you are in the CODOS Monitor instead. As with the cold start entry, reloading DMXMON does not affect the MAGIC/L code in the DATAMOVER RAM.

The warm entry point is found the easiest via the MAGIC/L command:

```
PRINT #R 16. , VTBL ( BASE REENTER )
```

This entry point enters MAGIC/L at a point where the MAGIC/L channels still assigned will not be disturbed. This means that any programs or commands executed outside of MAGIC/L must not disturb any of the MAGIC/L assigned channels.

A very minor point concerning both cold and warm entry is that MAGIC/L does not precede its signon message or prompt with a carriage return. This means that the message is printed at the current text cursor position, wherever it may be. On cold entry, the text cursor will almost always be at the beginning of a new line, since it takes a carriage return to execute the startup command. The warm entry is another matter, since it can occur at any point due to an error or user abort. Occasionally with a warm entry, you may find the "mgl>" prompt printed at the end of a line, rather than an the beginning of a new line. No harm is caused by this situation, which may be corrected by simply entering a carriage return.

4. SAVING A MAGIC/L PROGRAM

Saving a MAGIC/L program involves writing to disk a complete copy of the MAGIC/L language, including any new words you may have added or redefined. Thus, a SAVEd program may be considered just another version of the MAGIC/L language. It may be started up and used just like any other version. When a particular version of MAGIC/L is started up, it will initially be in the same state it was in when the program was saved.

4.1 THE SAVE COMMAND

Saving a compiled MAGIC/L program to disk is accomplished with the MAGIC/L SAVE command. For example:

```
SAVE MYPROGRAM
```

will write the current program to the file called MYPROGRAM.6. Once the file is written, MAGIC/L automatically exits back to DMXMON.

MAGIC/L writes the program file disk by constructing and executing an appropriate DMXMON RESAVE command. This has a couple of important implications. First, it means that the SAVE command will unconditionally overwrite an existing file which has the same name. Second, if a file is overwritten, that file will still contain the original date on which the file was first created. It may seem a bit risky overwrite files in this manner, but program development work should not be saved primarily with the SAVE command. You should always create a source file. This can be done using the MTU EDITor, or by working interactively in MAGIC/L while a log file is active. With source code available, recompiling a program is a simple matter.

4.2 MAKING SAVED PROGRAMS TRANSPORTABLE

The MAGIC/L language is protected to run on a single machine (assuming standard User Number protection). As you might assume, all programs saved from a particular version will contain the same protection as the original. Normally this would mean that application programs can be run only on the machine on which they were compiled. However, the Dictionary Utility (DICT) has a command (KILL EXTENDER) which disables the EXTENDER in a SAVEd version of MAGIC/L. Along with disabling the EXTENDER, the protection will be disabled as well. Such a version would then be transportable to other machines.

Since the EXTENDER is the routine which allows new symbols, variables, definitions, etc. to be added, the unprotected MAGIC/L is no longer extensible. However, existing variables, definitions, etc. may still be used, so application programs can run even though the EXTENDER is disabled.

5. THE MAGIC/L INTERFACE TO DMXMON

MAGIC/L is interfaced to the CODOS operating system through DMXMON. Because a number of enhancements were made to DMXMON Version 1.0 to allow for a better interface, it will be necessary to use DMXMON Version 1.1 or later to run MAGIC/L. The latest version of DMXMON is included on the MAGIC/L Distribution Disk to insure you have the required version.

The interface routines accomplish their task employing only DMXMON SVCs. This means that the standard CODOS environment, with its associated commands and features, is in effect while running MAGIC/L. The standard line editing capabilities (CNTL-B, etc.) available in CODOS are available while entering MAGIC/L statements. Also, you can execute any CODOS command or program, provided it does not conflict with the execution or memory requirements of MAGIC/L and DMXMON.

5.1 I/O CHANNELS

The use of only DMXMON SVCs implies that MAGIC/L must perform its I/O via CODOS channels. As it turns out, MAGIC/L's I/O commands work via channels as well. As you might expect, the interface routines were implemented such that the MAGIC/L channel number will be the same as the CODOS channel it uses. Thus, MAGIC/L provides nine I/O channels, numbered 1 through 9. CODOS channel 0 is reserved for operating system use, so MAGIC/L avoids using that one.

To provide a more stable environment, and support job files in the most flexible way, MAGIC/L performs some special I/O initialization during cold startup. During cold startup, MAGIC/L will unconditionally mark channel 1 as open for reading and channel 2 as open for writing. It then unconditionally FREEs CODOS channels 3 through 9. Next, MAGIC/L opens a channel to the console (for reading and writing) which it assigns to both the console I/O variables CICH and COCH. Finally, MAGIC/L assigns a second channel to its error message file MGL.E. Though the I/O interface automatically chooses which channel is used, the two channels above will invariably be channel 3 for console and channel 4 for error message file. The I/O interface will always choose the lowest available channel number.

By assigning its own console I/O channel, MAGIC/L insures that CICH and COCH are assigned to the console. MAGIC/L is also able to leave channels 1 and 2 almost undisturbed. The one exception is that MAGIC/L will make use of channel 1 to invoke a job file which runs and returns from the MTU Editor. Although MAGIC/L does not make much use of channels 1 and 2, application programs may make full use of these channels if desired. They may be accessed directly, or they may be assigned to the user definable built in channel variables to achieve some desired redirection. Though reading from a channel opened for writing is permitted, writing to a channel opened just for reading is not. Thus, writing to channel 1 will cause a "wrong I/O for open" error.

If you wish, you may close either channels 1 or 2 and reassign them to something else. However, you should be aware that MAGIC/L will not automatically close either channel when exiting MAGIC/L. When CODOS or DMXMON assumes control, these channels will be in the same state as they were when last in MAGIC/L. If either channel has been left unassigned, both CODOS and DMXMON will automatically reassign these channels to the console.

When using channels and redirection, it is recommended that the console channel variables, CICH and COCH not be disturbed. MAGIC/L ultimately relies on these channels to communicate with the user. A mistake in changing either could cause MAGIC/L to hang or crash.

5.2 PROGRAM ABORTS

There are three basic events which can cause a 68000 program, and thus a MAGIC/L program, to abort. The first event is a CODOS or DMXMON error. The second is when the user presses the INT key. The third is when the user presses CNTL-C, which is only effective when the keyboard is being tested for this abort. Most often this is used when output is being printed on the display. In all of these cases, the currently executing MAGIC/L program is aborted and a warm entry into MAGIC/L performed. This means that after one of these aborts, execution always returns to MAGIC/L. In the case of CODOS and DMXMON errors, the standard error message will be displayed before the warm entry occurs.

6. DMXMON MONITOR COMMANDS

To make CODOS and DMXMON easy to access from MAGIC/L, a number of MAGIC/L commands have been implemented which execute a corresponding CODOS or DMXMON command. Where possible, the command names are identical. However, some names conflict with certain MAGIC/L routine names. It these cases, the CODOS or DMXMON command name is changed slightly. The following is a list of these commands and their function:

MAGIC/L COMMAND	DESCRIPTION
CALC	Same as DMXMON CALC command.
CODOS	Same as DMXMON command which exits to CODOS. Won't free channels.
DCLOSE	Same as CODOS CLOSE command.
DOPEN	Same as CODOS OPEN command.
COPYF	Same as executing COPYF Utility.
DELETE	Same as CODOS DELETE command.
DIR	Same as executing DIR Utility.
DISK	Same as CODOS DISK command.
DRIVE	Same as CODOS DRIVE command.
FILES	Same as CODOS FILES command.
JOB	Same as CODOS DO command.
MON	Passes remainder of command line to DMXMON for execution.
REN	Same as CODOS RENAME command.
TY	Same as CODOS TYPE command.

In the case of the JOB (i.e. CODOS DO) command, execution of the job file will not take place until MAGIC/L is exited.

7. USING THE CODOS EDITOR FROM MAGIC/L

Since a powerful screen editor, EDIT, comes standard with your MTU computer, it would be desirable to have direct access to this editor from MAGIC/L. Because EDIT and DMXMON conflict in memory usage, access to the editor can can be accomplished only via a job file. The required process has been implemented in the ED command, which executes and returns from EDIT. In addition, the MED and EX commands have been implemented to save retyping the file name when you wish to edit and re-compile a program under development.

7.1 THE ED COMMAND

The ED command is used when you are in MAGIC/L and wish to edit a text file. The syntax of this command is as follows:

ED ‹edit file› [‹backup file›]

This command will write an appropriate job file, called MED.J, on the disk in the default drive. The ED command then assigns MED.J to channel 1 and returns execution back to CODOS. CODOS then begins executing the job file which will startup the EDIT program. A backup of the file will be made if the backup file name is specified. If the backup already exists, it will be overwritten. At this point you can use the EDIT program just as you would normally. When you QUIT the editor, the job file will still be assigned to channel 1. The final command in the job file reloads DMXMON and executes a GO command to MAGIC/L's warm entry point to return you to MAGIC/L. MAGIC/L will be in the same state as when you executed the ED command.

7.2 THE MED AND EX COMMANDS

To further simplify program development, there is also a MED command which saves the file name so that it need not be specified each time you wish to edit the file. The syntax for the MED command is:

MED [‹file name›]

You must specify the desired file name the first time you edit the file. Thereafter, you need only execute "MED" to edit the file again. Also, the command:

EX

will automatically compile the file specified for the MED command.

8. USING THE MAGIC/L BLOCK EDITOR

To complement the file editing capabilities of the MTU Editor, the BED.M program is supplied to provide editing capability while entering MAGIC/L routines interactively. When loaded, compound statements (definitions, loops, etc.) entered on the keyboard will be saved as a block in a 1000 character buffer. You can tell when you are entering a compound statement by the presence of dashes ("-") or extra angle brackets (">") in the prompt. After the compound statement has been executed you will be able to list, edit, and re-execute that compound statement as many times as you like, until you enter a new compound statement.

8.1 LOADING THE BED ROUTINES

This program is supplied in source code form so that it may be added to the language only when needed. To load the BED program, execute the command:

```
EXT BED
```

Once loaded, the saving of compound statements to a buffer automatically begins. Also, three routines are defined for manipulating that buffer. These routines are:

```
LIST [ <file name> ]  - List the buffer to the console or file.
RETRY                 - Re-compile and execute the buffer.
BED                   - Edit and RETRY the buffer.
```

For the following descriptions of these commands, assume the following compound statement has been entered on the keyboard:

```
mgl> ITER 10
mgl>> PRINT I
mgl>> LOOP
```

When entered, this compound statement would have printed the number from zero to nine, as well as have been saved in the buffer.

8.2 THE LIST COMMAND

The LIST command is used for listing the buffer contents to the console or a file. For example, executing LIST would display:

```
mgl> LIST
ITER 10
PRINT I
LOOP
mgl>
```

Specifying a file name would cause the buffer to be listed to the file. If the file already exists, the listing will be appended to the file. The default extension for the file name will be ".M".

8.3 THE RETRY COMMAND

The RETRY command is used to re-compile the buffer contents. This is accomplished by simply redirecting the RDLINE function to take its input from the buffer instead of the ICH channel. The text in the buffer will be displayed and compiled as though it had been entered on the keyboard. For example, executing RETRY would display:

```
mgl> RETRY
mgl> ITER 10
mgl>> PRINT I
mgl>> LOOP
```

along with the numbers from zero to nine again.

If the text contains a variable declaration or a function definition, the word will be redefined. Also, if an error or abort occurs while the compound statement is being entered, MAGIC/L will ABORT. The buffer will contain all lines entered up to the ABORT. Executing RETRY in this case will re-compile all lines in the buffer except the last one. The last line will be displayed on the console for editing, and control returned to the keyboard. You may edit this last line and enter any additional lines needed to complete the compound statement.

8.4 THE BED COMMAND

The BED command is used to edit the buffer contents and then RETRY the buffer. To execute the command, enter:

```
BED
```

The BED routine will then present each line in the buffer in sequence. For each line, you have four options:

1. Accept the line
2. Modify the line
3. Delete the line
4. Insert a new line before the current line

To accept the line as displayed, simply enter a carriage return. To modify the line, make your changes and enter a carriage return. The line editing is accomplished with the CODOS line editor routine, so the standard line editing features are available. To delete the line, simply clear the line with CNTL-X and enter a carriage return. Finally, to insert a new line before the one displayed, place a "~" character at the beginning of the line, followed by the text you wish to insert. Then entering a carriage return will insert the line and redisplay the current line again.

Once all you have sequenced through all of the lines in the buffer, an automatic RETRY is executed. Execution of the BED command will continue as described for the RETRY command, except that the modified buffer contents will be re-compiled.

9. MAGIC/L LOGGING FACILITIES

MAGIC/L contains provision for logging (i.e. duplicating) the terminal dialog to a specified file or device. The entire dialog may be logged, or just the input or output. The logging is accomplished via log channels LICH and LOCH. Use of these channels are simplified by the routines, LOGON, LOGINP, LOGOUTP, LOGOFF, and ENDLOG which are already built into the MAGIC/L Language.

9.1 THE LOG CHANNELS

Built into MAGIC/L are two channel variables, LICH and LOCH, which may contain a channel number or -1. These variables are used exclusively for logging terminal dialog. If LICH or LOCH contain -1, they are considered "deactivated", and no logging will be performed on that channel. LICH is used to log input that goes through the RDLINE function. This will include all keyboard input, as well as compilation of files. LOCH is used to log output that passes through the TSTRL routine. This includes all PRINT statements and prompts.

The two log channels may be used independently, or they may both be set to the same channel, so that both input and output are duplicated into the same log file. to avoid deadlocks, the logging procedure uses a special I/O call that ignores errors (i.e. log files may be incomplete if the disk becomes full).

9.2 THE LOGON COMMAND

The LOGON command is used to activate a log file. The syntax for the command is:

LOGON [‹file name›]

where ‹file name› specifies the desired log file, or device. If the file name is specified, a channel is opened to the file and logging is activated. If the file already exists, the log data will be appended to the end of the file. The file will be created if it doesn't already exist. The channel number will be placed in both LICH and LOCH, so both input and output will be logged to the specified file. If a log file was previously opened by LOGON, it will be closed.

If a log file has be previously opened, but temporarily deactivated with the LOGOFF command (described below), it may be re-activated by executing the LOGON command without a file name. If LOGON is executed without a file name when no file had previously been activated, the error message "Missing Name Field" is reported.

9.3 THE LOGINP AND LOGOUTP COMMANDS

The LOGINP and LOGOUTP commands are two alternate forms of the LOGON command. They have the same calling sequence as LOGON and differ only in the logging option they specify. The syntax for these commands in:

```
LOGINP  [<file name>]
LOGOUTP [<file name>]
```

where the <file name> argument operates the same as in the LOGON command.

The LOGINP command will activate logging of only the input portion of the terminal dialog. A log file created this way may be directly used as a source file, though presumably some editing would be desirable.

The LOGOUTP command will activate logging of only the output portion of the terminal dialog. This normally includes prompts, output from PRINT statements, and error messages. It is often called prior to calling a routine that will display data.

9.4 THE LOGOFF COMMAND

The LOGOFF command is used to temporarily suspend or deactivate the logging process. Both input and output logging are deactivated. The file remains open and the channel number is saved. It is desirable to use LOGOFF when compiling a file, otherwise the entire file will be duplicated in the log file. The LOGON, LOGINP, and LOGOUTP commands may be used to re-activate a suspended log file, or to change the current logging function.

9.5 THE ENDLOG COMMAND

The ENDLOG command is used to terminate the logging process. This will close the log channel, set LICH and LOCH to -1, and reset the other variables involved the the logging process. It will be necessary to specify a file name with the LOGON, LOGINP, and LOGOUTP, if you wish to resume logging.

10. THE SYMBOL TABLE UTILITY ROUTINES

Included standard in both MGL and RMGL are five symbol table utility routines. These routines allow you to manipulate MAGIC/L's symbol table and dictionary in various ways. The following is a list of the routines and a brief description of their use:

RENAME	- rename an existing entry in the symbol table
ALIAS	- create an additional entry for an existing one
REPLACE	- replace an existing function or command with a new one
FORGET	- remove routines and symbols from memory
DLIST	- list symbols to console
FLIST	- list symbols to a file (in SORTREF format)

10.1 THE RENAME COMMAND

The RENAME command is used to change the name of an existing entry in the symbol table to a new name. The change is made only to the name in the symbol table, and has no effect on the properties of the entry. Any entry may be renamed, including those for the "keywords." The syntax for the command is:

RENAME ‹old name› ‹new name›

where ‹old name› is the name of the existing entry and ‹new name› is the new name for the entry. For example:

RENAME INTEGER INT

will cause the name INT to be used in place of INTEGER.

If the new name specified matches an existing WORD, the original WORD remains unchanged. There will be, however, two entries in the symbol table with the same name. Because of the way MAGIC/L searches the symbol table, the more recently defined entry will be one used. Also it is recommended that the RENAME command be used sparingly, so that each user does not end up with a different dialect of MAGIC/L.

10.2 THE ALIAS COMMAND

The ALIAS command is used to create a new entry in the symbol table to refer to an existing WORD. Once the additional name is added, either name may be used to refer to the WORD.
The syntax for the command is:

ALIAS ‹new name› ‹old name›

where ‹new name› is the name for the new entry to add to the symbol table and ‹old name› is the name of the existing WORD. For example:

ALIAS INT INTEGER

will create a new entry, INT, with the same properties as INTEGER. The entry for INTEGER will be unchanged. If the new name specified exists as well, that

WORD will be redefined (i.e. renamed). Also, you should note that the sequence of arguments for ALIAS is reversed from that used by RENAME.

10.3 THE REPLACE COMMAND

The REPLACE command is used to replace an existing routine with a new one. Unlike redefining a routine, REPLACE will change the action of the routine for all previous uses. It is essential that the new version have the same calling sequence as the routine it replaces. This does not, however, prevent you from replacing a high level routine with an assembly routine (or vice versa), or replacing a function with a command, etc. The syntax for the command is:

```
REPLACE ‹old routine› ‹new routine›
```

where ‹old routine› is the name of the existing routine to be replaced, and ‹new routine› is the name of the new routine to substitute. For example:

```
REPLACE CHK FASTCHK
```

will cause FASTCHK to be executed wherever CHK had previously been used. Assuming the old routine was defined prior to the new one, a RENAME could be used after the REPLACE to change the new name to the old name.

REPLACE works by patching the old routine to jump execution to the new routine. As a result, the old routine is destroyed. If the new routine calls the old routine, the old action will NOT be performed. Rather, a recursive definition will be created.

10.4 THE FORGET COMMAND

The FORGET command is used to release memory back to a specified definition or label. The memory may then be reused for other applications. The symbol table space associated with the released definitions is also freed. The syntax of the command is:

```
FORGET ( ‹name› )
```

where ‹name› is a string. Memory will be released back to the definition or label named by the string.

A typical dialog using FORGET might look like this:

```
mgl› EXT FORGET
mgl› LABEL HERE
mgl› EXT MYFILE
   ... test MYFILE ...
mgl› ED MYFILE
mgl› FORGET ( 'HERE )
   ... back to second line ...
```

If HERE is declared as the first routine in MYFILE, it does not have to be defined here. This sequence allows MYFILE to be repeatedly compiled, without accumulating old versions in memory. If FORGET is loaded after MYFILE, the space for it is also recovered, since it will be forgotten also.

There are a number of limitations in the usage of FORGET. These are as follows:

1. Only the most recently defined words may be forgotten. FORGET operates from the current value of "." (the top of code space) back to the specified location.

2. All of the released routines MUST reside in the same branch, which must also be the current branch.

3. The WORD used to specify the FORGET range must not be a PARAMETER, since PARAMETERs have no associated address.

4. The symbol table space released by FORGET may only be reused as symbol table space. Thus, if FORGET is used over a large range of entries, some of the space will not be recovered immediately.

5. FORGET may not be called from within a "framed" definition, that is, one that declares arguments or locals. It is possible to use FORGET in a simple definition.

FORGET is typically used in three ways. First, routines which are used to fill in data structures, or to modify structures (such as the ALIAS routine) may be forgotten after their use. Second, utilities may be loaded, used, and forgotten. Third, programs under development may be forgotten, edited, and recompiled without leaving the MAGIC/L environment, though if the routine is small it is usually easier to simply "redefine" everything.

You may wish to note that the symbol table routines described in this chapter were added last in both MGL and RMGL. This means that these routines may be forgotten, if you wish to recover the memory space (about 990 bytes). To forget the symbol table routines, execute the command:

```
FORGET ( 'ALIAS )
```

10.5 THE DLIST AND FLIST COMMANDS

The DLIST and FLIST commands are used for listing the entries in the symbol tables. The DLIST command will display all of the entries in the most recently activated branch, five to a line. The FLIST command will list the entries to a file. The syntax for these commands is:

```
DLIST
```

and

```
FLIST ‹file name›
```

where ‹file name› is the file to which the list will be written. The FLIST command writes a cross reference type file which can be displayed by the SORTREF program (see chapter 12 describing the Cross Reference Utility). This file will contain only entry type records with no file information. In the SORTREF output the /B option gives all of the information. In addition, since only the first five characters and the length each name is maintained in the symbol table, the trailing characters of long names are displayed as underbars.

11. THE DICTIONARY UTILITY

The DICT program is a dictionary utility used to modify a MAGIC/L program after it has be SAVEd. The primary purpose of the utility is to compress the symbol table and to eliminate unneeded entries from the symbol table. In addition, DICT can be used to display, rename, and alias entries in the symbol table.

11.1 THE MAGIC/L SYMBOL TABLE

To make the actions of the DICT program a little more understandable, the following is a brief description of the structure and operation of the MAGIC/L symbol table. The symbol table in MAGIC/L contains a four word entry for each word defined in MAGIC/L. This includes all definitions, variables, labels, parameters, and assembler opcodes. Symbol table entries are used at compile time to determine the characteristics and addresses of the words. The symbol table is usually not referenced at runtime, so a "turnkey" program does not need a symbol table at all.

The symbol table is divided into several sections called BRANCHes (see the chapter on Vocabulary Branches in the MAGIC/L User's Manual). Since branches are often comprised entirely of entries used only in one phase of building a program, DICT has the facility to entirely eliminate such a branch.

During compilation the symbol table is built as a linked list of small buffers. This fragmentation optimizes memory usage during development, but adds overhead in the final SAVE file. DICT compresses the symbol table into one buffer per branch. Thus, simply running DICT on a SAVE file will typically reduce the size of the symbol table by about 10%.

11.2 RUNNING THE DICT UTILITY

The DICT program is a MAGIC/L SAVEd program itself. Thus it is executed from DMXMON just like any other executable object file. The syntax for starting up the DICT program is as follows:

```
DICT [<SAVE file> [<command file>]]
```

where <SAVE file> is the SAVEd MAGIC/L file you wish to access, and <command file> is the name of a file containing a list of DICT commands. If neither argument is specified, DICT will display with:

```
Input file:
```

and accept arguments the same as those accepted on the command line.

Once the SAVE file is specified, its symbol table will be read into memory. The SAVE file will not be accessed again unless the EXIT command is executed. If the command file is specified, the contents of the file will be used as the initial command stream for DICT. Once these commands have been executed, commands will be input from the keyboard (unless the command file performs an exit before end-of-file is encountered).

After the SAVE file has been opened and its symbol table read, DICT will enter the command mode. If the command file was specifed the commands will be read from the file. If no command file was specified, or when EOF is reached in the command file, or an error occurs with a command in the file, DICT will enter the interactive mode. At this point, DICT will prompt with:

```
dic>
```

and the user may enter any of the following DICT commands.

Some of the commands, described below, will act on a single branch called the Active branch. This branch will default on startup to the last branch displayed by the SHOW command. The Active branch may be respecified by the USE command.

11.3.1 THE HELP COMMAND

The HELP command gives a listing of the available commands with the calling sequence of each command. Arguments enclosed in square brackets ("[...]") are optional. The HELP message is:

```
Available commands:
  ALIAS   <new-name> <old-name>
  CLEAN
  EXIT
  HELP    [<topic>]
  KILL    <branch-name> or ALL
  LIST    [<entry_name> [<entry_name>]]
  PURGE   <entry_name> [<entry_name>]
  RENAME  <old-name> <new-name>
  QUERY   [ON or OFF]
  QUIT
  SHOW
  UNPURGE <entry_name> [<entry_name>]
  USE     [<branch_name>]
```

11.3.2 THE ALIAS COMMAND

The ALIAS command is used to create a new entry that takes on the same characteristics as some other entry. The syntax of the command is:

```
ALIAS <new-name> <old-name>
```

where <new-name> is the desired name of the new entry, and <old-name> is the name of the existing entry. The new entry will always be placed in the last branch listed by the SHOW command. The old entry will come from the Active branch. Both entries will be functionally equivalent. The ALIAS command may be used to transfer entries to a new branch.

11.3.3 THE CLEAN COMMAND

The CLEAN command is used to remove all unneeded symbols from a selected branch. The syntax of the command is:

CLEAN [<branch_name> or ALL]

where <branch_name> is the name of a branch to be "cleaned." If ALL is specified, all branches will be cleaned. If no argument is specified, only the Active branch will be cleaned. The CLEAN command will scan the appropriate branch or branches, eliminating those entries which have their purge bit set. The corresponding symbol table space will be freed.

11.3.4 THE EXIT COMMAND

The EXIT command is used to write the new symbol table back into the SAVE file. It will also modify certain internal variables such as MEMORY to reflect the memory space gained. After updating the file, the EXIT command will return execution back to DMXMON. Since this command modifies the SAVE file, it may be desirable to make a backup copy before using the DICT utility.

11.3.5 THE KILL COMMAND

The KILL command is used to eliminate an entire branch. It is equivalent to setting the purge flag for all entries in a branch and then CLEANing that branch. The syntax for the command is:

KILL [<branch> or ALL or EXTENDER]

where <branch> is a branch to kill. If no argument is specified, the Active branch is killed. If a branch is specified, it is made the Active branch and then killed. If "ALL" is specified, all branches are killed.

Specifying the "EXTENDER" argument results in a special version of the KILL command. It eliminates those entries which allow for the extension of the symbol table. It also clears the corresponding code space and disables the serial number checking. This results in version of MAGIC/L which can no longer add new entries to the symbol table, making it non-extensible. This non-extensible version may be executed on any MTU system which can run DMXMON.

If you plan to kill the MG/L branch along with the EXTENDER, it will be necessary to kill the EXTENDER first. This allows certain symbols to be found before they are killed in the MG/L branch.

11.3.6 THE LIST COMMAND

The LIST command is used to display one or more entries in the Active branch. The syntax for the command is:

LIST [<entry_name> [<entry_name>]]

where <entry_name> is the name of an entry in the symbol table. If no entry names are specified, all entries in the Active branch are listed. If one entry is specified, only that entry is listed. If two entries are specified, all entries in the range from the first entry to the second entry are listed.

11.3.6 THE PURGE COMMAND

The PURGE command is used to flag one entry or a range of entries in the Active branch for purging. The syntax of the command is:

PURGE ‹entry_name› [‹entry_name›]

where ‹entry_name› is the name of an entry in the symbol table. If one entry is specified, only that entry is purged. If two entries are specified, all entries in the range from the first entry to the second entry are purged. Once flagged, removal the purged entries is accomplished with the CLEAN command.

11.3.7 THE QUERY COMMAND

The QUERY command is used to control the mode of the Query facility in DICT. The Query facility provides for a verification before the execution of potentially destructive commands. If Query mode is ON, the commands QUIT, EXIT, CLEAN, and KILL will ask if the user really wants to perform that function. If the response is any string beginning with "Y", the action will be performed. The syntax for the command is:

QUERY [ON or OFF]

If no argument is specified, the current query mode is displayed. The default state of the Query mode is on. Typically you will want to turn Query mode OFF is you plan to input commands from a file.

11.3.8 THE QUIT COMMAND

The QUIT command is used to exit DICT without updating the SAVE file. The syntax for the command is:

QUIT

This command should be used if a mistake was made, or if DICT was used for inspection only. The SAVE file will not be modified if QUIT is used to terminate DICT.

11.3.9 THE RENAME COMMAND

The RENAME command is used to change the name of an entry in the Active branch. The syntax for the command is:

RENAME ‹old-name› ‹new-name›

where ‹old-name› is the current name of the entry, and ‹new-name› is the desired new name for the entry.

11.3.10 THE SHOW COMMAND

The SHOW command is used to list the branches that have been read into memory, and the number of entries in each branch. The Active branch will be displayed with an asterisk beside the name. The syntax for the command is:

SHOW

Since the name of a branch is actually an entry that may have been purged previously, unnamed branches are given names starting with "BRO." Also, the number of words saved so far is displayed.

11.3.11 THE UNPURGE COMMAND

The UNPURGE command is used to clear the purge flag for one entry or a range of entries. The syntax for the command is:

UNPURGE ‹entry_name› [‹entry_name›]

where ‹entry_name› is the name of an entry in the symbol table. If one entry is specified, only that entry is unpurged. If two entries are specified, all entries in the range from the first entry to the second entry are unpurged. This will prevent those entries from being removed by the CLEAN command.

11.3.12 THE USE COMMAND

The USE command is used for selecting the Active branch. The syntax for the command is:

USE ‹branch_name›

where ‹branch_name› is the name of the desired branch.

11.4 TYPICAL USES OF DICT

The major purpose of DICT is to recover space from the symbol table. Several hundred words may be recovered simply by the compression of buffers in the symbol table. DICT is used for this purpose on the distribution versions of MAGIC/L.

Most programs contain entries that are only used internally, and therefore may be purged. Note that this process may be performed repeatedly as a program is developed, thus insuring that maximum space is available at any time. If the internal routines are placed in a separate branch, it is easy to eliminate the unneeded entries.

A common usage of DICT is to purge the assembler branch after the assembly code for any application is written. The resulting file will have as much as 500 words of additional space and may serves as the basis for future development.

The BTEMP branch is used internally be MAGIC/L to support local variables and input arguments. The names of the entries in the BTEMP branch will be meaningless because they are modified after the END statement of a DEFINE-END structure. BTEMP should not be PURGEd or KILLed if the resulting file is to be capable of DEFINE structures.

A special use for DICT is to kill the EXTENDER. Primarily, this will make an application transportable as well as free up some memory. It should be noted that although the program will no longer be extensible, all of the interactive compiler functions (DO-LOOP, WHILE-REPEAT etc) will still work. However, the following commonly used words will be disabled:

ACTION	ADDRESS	BOFFSET	BRANCH	BUILD
CHAR	COMMAND	DEFINE	DUMMY	ENDRECORD
ENTRY	.EVEN	INTEGER	LABEL	LOCAL
LONG	MAKE	OFFSET	PARAMETER	PARSED
REAL	RECORD			

In addition, several words indirectly reference the extender and thus should also be purged if the are present:

ALIAS ENTER

An example use of the DICT program is on the DICT program itself. The EXTENDER was killed, so it can be run on any MTU system which can run DMXMON. Also, since all of the DICT entries were placed in two special branches (one for commands, the other for the HELP functions), the MG/L, .MAC, and BTEMP branches were all KILLed. This will release about 3000 words for program use.

12. THE MAGIC/L CROSS REFERENCE UTILITY

The MAGIC/L cross reference utility is used to create a sorted cross reference map of a MAGIC/L program. All entries created by the program can be listed in alphabetical order, and optionally, all references to those entries listed as well. The cross reference listings will show the file name in which the reference occurred, the entry making the reference, and the line number in the file.

The cross reference utility comes in two parts. The first part is the XREF.M program. This is supplied in source code form so that it may be optionally added to MAGIC/L when desired. XREF.M must be loaded prior to compiling any files that are to be cross referenced. Once loaded and activated, it will generate an "XREF" file as other source files are compiled.

The second part of the cross reference utility is the SORTREF.6 program. This is a utility program which is executed directly from DMXMON. Its function is to display an "XREF" file previously generated by XREF.M or FLIST command.

12.1 GENERATING THE CROSS REFERENCE FILE

The first step in generating a cross reference map of a program is to generate the cross reference, or "XREF", file. This is accomplished by first loading the XREF.M program, using the command:

```
EXT XREF
```

Once loaded, four utility commands are available for controlling the cross reference file. These commands are as follows:

```
XREF [<file name>]
XRENTRY [<file name>]
XRLOOKUP [<file name>]
ENDXREF
```

where <file name> is the name of the desired cross reference file. A default extension of ".r" will be added to the name if an extension is not already present.

XREF, XRENTRY, and XRLOOKUP all activate some form of the cross referencing process. The file name must be specified the first time cross referencing is activated. After that, switching between different forms of cross referencing is accomplished by executing the commands without the file name. Cross referencing to a given file continues until an ENDXREF command is executed, or a new cross reference file is specified with one of the other commands. With a cross reference file active, a particular program is cross referenced by loading it with the EXT or EX command.

The cross reference file can contain two types of records, "Entry" records, and "Lookup" (i.e. reference) records. The declaration of definitions or variables can generate "Entry" records, thus logging the creation of new entries. When an "Entry" record is written to the cross reference file, a flag is also set in the symbol table for that entry. Whenever there is a successful LOOKUP of a definition or variable which has this flag set, the second type of record (i.e. "Lookup" record) can be generated. This means that the "Lookup"

record logs references to those definitions and variables that have "Entry" records. The only difference between the XREF, XRENTRY, and XRLOOKUP commands is which of the two types of records they will write to the cross reference file.

12.1.1 THE XREF COMMAND

The XREF command is used to create a full cross reference of the programs which are subsequently compiled. With this form of cross reference active, both types of records will be written to the cross reference file. The following shows the dialog for making a full cross reference of a file called MYFILE.M:

```
mgl> XREF MYFILE.R     ; activate cross referencing, entries and references
mgl> EXT MYFILE        ; load user program file
mgl> ENDXREF           ; terminate cross reference
```

The cross reference records will be written to a file called MYFILE.R and will contain records for both entries and references.

12.1.2 THE XRENTRY COMMAND

The XRENTRY command is used to just list the definitions and variables in a program. With this form of cross reference active, only "Entry" records will be written to the cross reference file. Typically the XRENTRY command will be used in conjunction with the XREF command so that references to certain routines are ignored. A typical dialog might look like:

```
mgl> XRENTRY MYFILE.R   ; activate cross referencing, just entries
mgl> EXT MYLIB          ; load library file
mgl> XREF               ; generate entries and references
mgl> EXT MAINPROG       ; load main program
mgl> ENDXREF            ; terminate cross reference
```

This dialog would result in the cross reference file, MYFILE.R, containing only entries for the routines in MYLIB.M, and entries with references for the routines in MAINPROG.M.

12.1.3 THE XRLOOKUP COMMAND

The XRLOOKUP command is used just list the references to previously recorded entries. With this form of cross reference active, only "Lookup" records will be written to the cross reference file. If the XRLOOKUP command is used, it must follow one of the other commands. Otherwise, there would be no logged entries that could be "looked up". A typical dialog might look like:

```
mgl> XREF MYFILE.R      ; generate both types
mgl> EXT PRIMITIVES     ; load primitive routines
mgl> XRLOOKUP           ; change to references only
mgl> EXT FILE1          ; generate references to
mgl> EXT FILE2          ;    the primitive routines
mgl> ENDXREF            ; terminate cross reference
```

This would generate a cross reference file called MYFILE.R which would contain entries and references for the PRIMITIVES.M file, and only references to the "primitives" for the FILE1.M and FILE2.M files.

12.2 LISTING THE CROSS REFERENCE FILE

Once the cross reference file has been generated, it can be listed using the SORTREF program. This listing may be written either to the display or a disk file. The SORTREF utility is a program which is executed directly from DMXMON. The command syntax for executing SORTREF is:

```
SORTREF [<input file>[/A/B/E/N/P/T] [<output file>]]
```

where <input file> is a cross reference file, previously generated using XREF.M or by the FLIST command. A default file extension of ".R" will be used if an extension is not present in the name. The optional <output file> is the destination for the generated listing. A default file extension of ".L" will be used if an extension is not present on the name. If no output file is specified, the listing will be written to the display.

The "/A/B/E/N/P/T" above are all option switches which control the display listing. These switches function as follows:

SWITCH	USAGE
A	Display addresses in Decimal (default is hex)
B	Brief Entry Only display.
E	Display Entries only (no references)
N	Do not sort. Implies display Entries only.
P	Display parameter values in Decimal (default is hex)
T	Echo results on terminal

If no arguments are specified on the command line, SORTREF will prompt:

```
SORTREF requires one or two arguments:
 <input-filename>[/A/B/E/N/P/T] [<output-filename>]
SORTREF>
```

and wait for arguments as would be specified on the command line. A typical command might be:

```
SORTREF  MYFILE/E  MYFILE.L
```

which would write a sorted list of the entries logged in the cross reference file MYFILE.R to the output file MYFILE.L.

12.3 THE CROSS REFERENCE LIST OUTPUT

The listing generated by SORTREF can contain output from both types of records (i.e. entry and reference records) found in the cross reference file. The entry records can be displayed in two formats; a full format requiring two lines:

```
<entry name>  <type>                              <# of arguments>
        <file>/<line #>       [<address>]
```

and the following brief format (/B option) requiring only one line:

```
<entry name>  <type>                    <# of arguments>   [<address>]
```

Where:

entry name	is the full name of the entry.
type	is the type of entry. This is as descriptive as possible. Parameters will show the value.
# of arguments	is displayed for functions and commands that declare input arguments. It is not displayed for assembly routines.
file	is the file in which the entry is defined.
line #	is the line of the file in which the entry is defined.
address	is the VTBL entry which contains the address where the definition is compiled.

The reference records are displayed in the following format:

```
            <file>          <definition>     <line #>
```

Where:

file	is the file in which the entry is referenced. If the reference was at the console, "console" is displayed.
definition	is the definition in which the entry is referenced. If the reference was not within a definition, "....." is displayed.
line #	is the line of the file in which the entry is referenced.

If two records in a row have the same file, the filename is not redisplayed. Similarly, if the definition is the same, it is not repeated and the line numbers are placed on the same line. Thus, the entire display (including entry record) could have the following format:

```
<entry name>  <type>                              <# of arguments>
        <file>/<line #>       [<address>]
        <file>          <definition>     <line#> <line#> <line#>
                        <definition>     <line#>
        <file>          <definition>     <line#> <line#>
```

To illustrate what the output might look like, here is an example program, and what the various cross reference listings would be. Assume the following program is MYFILE.M:

```
PARAMETER YSIZE := 3
INTEGER X Y ( YSIZE )

DEFINE PX
        INTEGER VAL
  PRINT X * VAL
  ITER YSIZE
    PRINT X , Y ( I )
  LOOP
END
```

and a cross reference file was generated with:

```
XREF MYFILE.R
EXT MYFILE.M
ENDXREF
```

The full cross reference will look like this:

```
PX              Subroutine                          1 Input Value
        MYFILE.mg/3                     [6024]

X               Integer
        MYFILE.mg/1                     [6004]
        MYFILE.mg       PX                      5     7

Y               Integer Array
        MYFILE.mg/1                     [6012]
        MYFILE.mg       PX                      7

YSIZE           Parameter = 3
        MYFILE.mg/0
        MYFILE.mg       .....                   1
                        PX                      6
```

A cross reference of entries only looks like this:

```
PX              Subroutine                          1 Input Value
        MYFILE.mg/3                     [6024]

X               Integer
        MYFILE.mg/1                     [6004]

Y               Integer Array
        MYFILE.mg/1                     [6012]

YSIZE           Parameter = 3
```

A brief cross reference of entries only looks like this:

```
PX              Subroutine                          1 Input Value  [6024]
X               Integer                                            [6004]
Y               Integer Array                                      [6012]
YSIZE           Parameter = 3
```

Note that the input argument, VAL, is not included, because it is only temporary, and the name may be reused.

If words are redefined, two entry records will be displayed, each with the reference records that are appropriate.

Naturally there is an upper limit on the number of records that can be handled by SORTREF. The cross reference file may have up to 64K records. SORTREF, however, currently uses a core-based sort that only has room for about 8192 records. For sorting entries only, this will be sufficient for a very large program. If the full cross reference is desired, this will be enough for a fairly large sized program (50 to 60 pages). If necessary this limitation may be overcome by breaking the cross reference process into several parts, making use of the XRENTRY and XRLOOKUP commands. When the number of records exceeds several hundred, the sorting process may take a while; please be patient.

13. MAGIC/L LOW-LEVEL SVC ROUTINES

All DMXMON SVCs (except SVC 2) have a MAGIC/L routine which may be called to execute that SVC. This is accomplished using two "action" routines, called SYSCAL and SYSCALM. Each SVC routine uses one of these action routines to make the necessary SVC call to DMXMON. On call, the register values are passed as arguments. On return, the return register values are placed in the system registers, SYSR0, SYSR1, and SYSR2 for accessing by the running program.

All SVC names in MAGIC/L begin with a "?". This is done to minimize the chance of name conflicts with these SVCs. The remainder of the name will match at least part of the SVC name given in the DMXMON manual. If you wish to delete these SVCs from the symbol table, you can use the DICT Utility and purge symbols from ?RTS to ?CWS and ?TOM to ?KYBOFF.

The MAGIC/L SVCs should be considered low-level routines. All passed arguments must be long values. Also, the sequence of the arguments may not be to most logical. Typically, higher-level routines will be written to better interface these SVCs to an application program. An example of this is the Integer Graphics Library (IGL) which is supplied on your distribution disk. The routines in the IGL have a logical syntax, similar to BASIC's IGL commands. Also, the IGL routines accept integer arguments, and some are functions which return a value.

In the syntax shown below, the proper order of the register arguments is given. In an MAGIC/L statement, the register values may be specified by any long expression. Some SVC's may require additional arguments to be stored in memory at some location. You should refer to the DMXMON Manual for details about extra arguments and the function of the values in the passed arguments.

On return from the SVC routines, the return values may be found in the system registers, SYSR0, SYSR1, and SYSR2. These are all long variables already defined in MAGIC/L. Shown in the tables below are which system registers will contain return values and which 68000 register returned the value. You should refer to the DMXMON Manual for details on what is represented by these values.

13.1 CODOS RELATED SVC ROUTINES

The following are the CODOS related low-level SVC routines. All names in this group consist of the last three characters of the name in the DMXMON manual, preceded by a "?".

SVC#	MAGIC/L SYNTAX	RETURN VALUES (TO MAGIC/L)
0	?RTS	none
1	?RET	none
2	not implemented	
3	?INB (D1)	SYSR0 = D0, SYSR2 = CY
4	?OUB (D1 , D0)	SYSR2 = CY
5	?INL (D1)	SYSR0 = D0, SYSR2 = CY
6	?OUL (D1 , D0)	none
7	?OUS (D1 , D0)	none
12	?DFB	none
13	?MON (A0)	none
14	?QCA (D1)	SYSR0 = D0, SYSR2 = CY
15	?INR (D1 , D0 , A0)	SYSR0 = D0, SYSR2 = CY, (A0 is ignored)

16	?OUR (D1 , D0 , A0)	SYSR2 = CY
17	?BOF (D1)	none
18	?EOF (D1)	none
19	?PSF (D1 , D0)	none
20	?QFP (D1)	SYSR0 = D0
21	?ASS (D1 , D0 , A0)	SYSR0 = D0
22	?FRE (D1)	none
23	?TNC (D1)	none
28	?QVN	SYSR0 = D0
29	?QFS (A0)	SYSR0 = D0, SYSR2 = CY, (A0 is ignored)
30	?DAT (A0)	none
31	?TIM (A0)	none
32	?TOM	SYSR0 = A0
33	?CEH (D0 , A0)	none
34	?IWS (D0 , A0)	none
35	?CWS (D0 , A0)	none

13.2 TEXT INPUT AND DISPLAY SVC ROUTINES

The following are text input and display related SVC routines. Each of these, where possible, match the name in the DMXMON manual, preceded by a "?".

SVC#	MAGIC/L SYNTAX	RETURN VALUES (TO MAGIC/L)
64	?GETKEY	SYSR0 = D0
65	?TSTKEY	SYSR0 = D0, SYSR2 = CY
66	?IFKEY	SYSR0 = D0, SYSR2 = CY
67	?INLINE (A0)	SYSR0 = D0, SYSR2 = CY
68	?EDLINE (D0 , A0)	SYSR0 = D0, SYSR2 = CY
69	?CLRDSP	none
70	?INITIO	none
71	?INITTXW	none
72	?DEFTW (D1 , D0)	none
73	?REDTW	SYSR0 = D0, SYSR1 = D1
74	?CLRHTW	none
75	?CLRTW	none
76	?CLRTLN (D0)	none
77	?CLREOL	none
78	?CLREOSC	none
79	?CLRLEG	none
80	?DRWLEG (A0)	none
81	?OFFTCR	none
82	?ONTCR	none
83	?LFPTCR	none
84	?SETTCR (D0 , D1)	none
85	?READTCR	SYSR0 = D0, SYSR1 = D1
86	?CRLF	none
87	?HOMETW	none
88	?CURUP	SYSR2 = CY
89	?CURLFT	SYSR2 = CY
90	?CURRGT	SYSR2 = CY
91	?CURDWN	SYSR2 = CY
92	?OUCH (D0)	none
93	?BEEP (D0)	none

13.3 GRAPHIC DISPLAY SVC ROUTINES

The following are graphics display related SVC routines. Each of these, where possible, match the name in the DMXMON manual, preceded by a "?".

SVC#	MAGIC/L SYNTAX	RETURN VALUES (TO MAGIC/L)
128	?SGMODE (D0)	none
129	?RGMODE	SYSR0 = D0
130	?SDSPAT (D0)	none
131	?RDSPAT	SYSR0 = D0
132	?SMOVE (D0 , D1)	none
133	?SDRAW (D0 , D1)	none
134	?SVEC (D0 , D1)	none
135	?SDOT (D0 , D1)	none
136	?SMOVER (D0 , D1)	none
137	?SDRAWR (D0 , D1)	none
138	?SVECR (D0 , D1)	none
139	?SDOTR (D0 , D1)	none
140	?DRWCH (D2)	none
141	?SISDOT (D0 , D1)	SYSR2 = CY
142	?SONGC	none
143	?SOFFGC	none
144	?RDGCSR	SYSR0 = D0, SYSR1 = D1
145	?SGRIN	SYSR0 = D0, SYSR1 = D1, SYSR2 = D2
146	?SLTPEN	SYSR0 = D0, SYSR1 = D1, SYSR2 = CY
147	?SINTLP	none
148	?STSLP	SYSR0 = D0, SYSR1 = D1, SYSR2 = CY

13.4 SPECIAL FUNCTION SVC ROUTINES

The following are special function SVC routines. Each of these, where possible, match the name in the DMXMON manual, preceded by a "?".

SVC#	MAGIC/L SYNTAX	RETURN VALUES (TO MAGIC/L)
192	?READM (A0)	SYSR0 = D0
193	?WRITEM (D0 , A0)	none
194	?BLKRD (D0 , A1 , A0)	none
195	?BLKWRT (D0 , A1 , A0)	none
196	?CALLM (CY , D2 , D1 , D0 , A0)	SYSR0 = CY , Y , X , A BITS 31-24, 23-16, 15-8, 7-0
197	?SETWD68 (D2 , D1 , D0 , A0)	none
198	?SETVP65 (D1 , D0)	none
199	?RFSH1	none
200	?RFSR	none
201	?RFSP	none
202	?CPYDSP	none
203	?WAIT (D0)	none
204	?CLKON (D0)	none
205	?CLKOFF	none
206	?CNTON (D0)	none
207	?CNTOFF	none
208	?KYBON	none
209	?KYBOFF	none

14. ADDITIONAL MAGIC/L ROUTINES

There are a few MAGIC/L routines which where written specifically to help interface MAGIC/L to DMXMON and CODOS. Some of these may be useful in a general sense as well.

14.1 THE ?EXISTS FUNCTION

PURPOSE: To determine if a specified file exists.

SYNTAX: ?EXISTS (‹file name›)

ARGUMENTS: ‹file name› = string buffer containing the file name.

DESCRIPTION: The ?EXISTS function tests to see if the specified file exists and returns a value of -1 (i.e. TRUE) if it does. If the file doesn't exist or a condition prevents this determination (i.e. drive not open, etc.), the function returns a value of 0 (i.e. FALSE).

```
EXAMPLE:  IF ( ?EXISTS ( "MYFILE.M" ) )
             DELETE MYFILE.M
          ENDIF
```

will delete MYFILE.M, avoiding the "FILE NOT FOUND" error if the file didn't exist.

14.2 THE FIXEXT ROUTINE

PURPOSE: To add a default extension to a file name if an extension isn't present.

SYNTAX: FIXEXT (‹file name› , ‹ext›)

ARGUMENTS: ‹file name› = string buffer containing the file name.
‹ext› = string containing the period and default extension.

DESCRIPTION: The FIXEXT routine is used to insure that a file name contains at least a default extension, if an extension is not already present. The extension, if added, is inserted into the file name, so the buffer should be of sufficient length to hold the extra characters. There should not be any extra blanks or characters on the end of the file name as the extension will simply be appended if the buffer contents have no extension or drive number. If the file name contains only one character, it is assumed to be a device name, and the extension is not added.

```
EXAMPLE:  CHAR FBUF ( 20. )
          MVSTR ( "TESTNAME:1" , FBUF )
          FIXEXT ( FBUF , ".T" )
```

will result in FBUF containing "TESTNAME.T:1".

15. THE INTEGER GRAPHICS LIBRARY

This chapter describes a set of routines referred to as the Integer Graphics Library. These routines provide a means of performing simple graphics operations. All graphics coordinates are integers, and correspond exactly to the pixel coordinates of the display. The coordinate range for X is 0 to 479, and 0 to 199 for Y.

These routines are found in the source file, IGL.M, and may be added to MAGIC/L by simply compiling them with the EXT command. Please note that all arguments are enclosed within parentheses. Failure to enclose them within parentheses will lead to incorrect results, but may not immediately cause an error message.

The implementation of these routines consists primarily of converting the integer arguments to the long value required by the DMXMON SVCs, then calling the appropriate SVC. For further details on these routines, determine from the source file which SVCs are being called, the refer to those SVCs in the DMXMON Manual.

15.1 THE DASHPAT ROUTINE

PURPOSE: Set the dashed line pattern.

SYNTAX: DASHPAT (‹pattern›)

ARGUMENTS: ‹pattern› = desired 16 bit dot pattern.

DESCRIPTION: The DASHPAT routines sets the specified dash pattern. "1" bits in the dash pattern plot the corresponding dot in the line. "0" bits skip plotting of the corresponding dot.

EXAMPLE: DASHPAT (0F0F0x)

will set the dash pattern to 4 dots plotted followed by 4 dots skipped.

15.2 THE PENMODE ROUTINE

PURPOSE: Set the drawing mode.

SYNTAX: PENMODE (‹mode›)

ARGUMENTS: ‹mode› = the desired drawing mode.
0=move, 1=draw, 2=erase, 3=flip
5=draw dashed, 6=erase dashed, 7=flip dashed.

DESCRIPTION: The PENMODE routine sets the specified drawing mode.

EXAMPLE: PENMODE (1)

will select the normal draw mode.

15.3 THE SCLEAR ROUTINE

PURPOSE: To clear the entire screen.

SYNTAX: SCLEAR

ARGUMENTS: none

DESCRIPTION: The SCLEAR routine clears the entire screen, including the legend boxes.

EXAMPLE: SCLEAR

will clear the entire display.

15.4 THE SDRAW ROUTINE

PURPOSE: To draw a line from the graphics cursor position to specified coordinates.

SYNTAX: SDRAW (‹x-coord› , ‹y-coord›)

ARGUMENTS: ‹x-coord› = desired X coordinate.
‹y-coord› = desired Y coordinate.

DESCRIPTION: The SDRAW routine draws a solid line from the current graphics cursor position to the specified coordinates. The drawing mode is automatically set but not restored as part of the execution of this command.

EXAMPLE: SDRAW (XEND , YEND)

will draw a solid line from the current graphics cursor position to XEND, YEND.

15.5 THE SFILL ROUTINE

PURPOSE: To fill a rectangular area using the current drawing mode.

SYNTAX: SFILL (‹left› , ‹right› , ‹bottom› , ‹top›)

ARGUMENTS: ‹left› = X coordinate of left edge of rectangle.
‹right› = X coordinate of right edge of rectangle.
‹bottom› = Y coordinate of bottom (or top) edge of rectangle.
‹top› = Y coordinate of top (or bottom) edge of rectangle.

DESCRIPTION: The SFILL routine is used to fill a rectangular area by drawing lines using the current drawing mode. The area if filled by drawing horizontal lines from left to right. The horizontal lines start at the lesser Y coordinate of the "top" or "bottom" argument. Lines are drawn with the Y coordinate incrementing toward the larger of the "top" or "bottom" arguments. If the "bottom" argument is really the bottom edge, filling proceeds from bottom to top of the rectangle. If the "bottom" argument is really the top edge, filling proceeds from top to bottom.

EXAMPLE: SFILL (0 , 479 , 0 , 255)

fills the entire screen, going from bottom to top.

15.6 THE SGRIN FUNCTION

PURPOSE: To input graphics coordinates using the "grin" cursor.

SYNTAX: ‹exit char› := SGRIN (‹x address› , ‹y address›)

ARGUMENTS: ‹x address› = address of variable to receive X coordinate.
‹y address› = address of variable to receive Y coordinate.
‹exit char› = ASCII value of character used to exit grin routine.

DESCRIPTION: The SGRIN function is used to input X,Y coordinates using the "grin" cursor. When executed, the "grin" cursor will appear at the current graphics cursor position. It may then be steered around the screen using the cursor keys. Once a non-cursor key is pressed, the coordinates of the "grin" cursor's last position are stored in the variables whose addresses where passed to the function. The value returned by the function is the ASCII code for the non-cursor key which was pressed.

```
EXAMPLE: INTEGER X , Y
         CHAR CH
         CH := SGRIN ( & X , & Y )
```

will invoke the "grin" cursor. When a non-cursor key is pressed, the ASCII code of that key will be stored in CH. Variables X and Y will receive the coordinates of the select point. Be sure to pass addresses as arguments. Executing: SGRIN (X , Y) could cause unpredictable results.

15.7 THE SLABEL COMMAND

PURPOSE: To print a string of characters starting at the current graphics cursor position.

SYNTAX: SLABEL (‹string›)

ARGUMENTS: ‹string› = address of character string to be displayed.

DESCRIPTION: The SLABEL command is used to print a string of characters starting at the current graphics cursor position. The characters are drawn in a 6 X 10 dot cell which is erased prior to displaying the character, unless the character is the underline ("_"). The base line for drawing the characters is the third line from the bottom of the cell. Lower case letters g, j, p, and q will have descenders which reach the bottom of the cell. The bottom corner of the first cell will be displayed at the current graphics cursor position. If part a character would be drawn beyond any edge of the screen, the character is not drawn and the remainder of the string is not displayed.

```
EXAMPLE:  SMOVE 100,100
          SLABEL "Display this string."
```

will print the string starting at coordinates 100,100.

15.8 THE SLTPEN FUNCTION

PURPOSE: To input graphics coordinates using the light pen.

SYNTAX: ‹hit› := SLTPEN (‹x address› , ‹y address›)

ARGUMENTS: ‹x address› = address of variable to receive X coordinate.
‹y address› = address of variable to receive Y coordinate.
‹hit› = hit flag, 0 = no hit, -1 = hit.

DESCRIPTION: The SLTPEN function is used to input X,Y coordinates using the light pen. When executed, the entire screen will be scanned once for a light pen "hit". If a "hit" occurs, the coordinates of the "hit" are stored in the variables whose addresses were passed as arguments. Also, the return value of the function will be -1. If a "hit" does not occur, the return value of the function is 0 and the variables are left unchanged.

```
EXAMPLE:  INTEGER HIT X Y
          HIT := SLTPEN ( & X , & Y )
```

will scan the screen for a light pen "hit". Variables X and Y will receive the coordinates of the "hit", if one occurs. HIT will receive the return value of the function.

15.9 THE SMOVE ROUTINE

PURPOSE: To set the graphics cursor position to the specified X,Y coordinates.

SYNTAX: SMOVE (‹x-coord› , ‹y-coord›)

ARGUMENTS: ‹x-coord› = desired X coordinate.
‹y-coord› = desired Y coordinate.

DESCRIPTION: The SMOVE routine sets the graphics cursor position to the specified position.

```
EXAMPLE:  SDRAW ( XEND , YEND )
```

will set the graphics cursor position to XEND, YEND.

15.10 THE SPEN ROUTINE

PURPOSE: To draw a line from the graphics cursor position to specified coordinates using current drawing mode.

SYNTAX: SPEN (‹x-coord› , ‹y-coord›)

ARGUMENTS: ‹x-coord› = desired X coordinate.
‹y-coord› = desired Y coordinate.

DESCRIPTION: The SPEN routine draws a line from the current graphics cursor position to the specified coordinates using the current drawing mode. The drawing mode is set by the PENMODE command.

EXAMPLE: SPEN (XEND , YEND)

will draw a line from the current graphics cursor position to XEND, YEND using the current drawing mode.

15.11 THE SQDOT FUNCTION

PURPOSE: To obtain the state of the pixel at the specified coordinates.

SYNTAX: ‹pixel state› := SQDOT (‹x-coord› , ‹y-coord›)

ARGUMENTS: ‹x-coord› = desired X coordinate.
‹y-coord› = desired Y coordinate.

DESCRIPTION: The SQDOT function is used to obtain the state of the pixel at the specified coordinate position. The function returns -1 if the pixel is "on" (i.e. corresponding bit is 1). The function returns 0 if the pixel is "off" (i.e. corresponding bit is 0).

```
EXAMPLE: INTEGER PIX
         PIX := SQDOT ( 100 , 100 )
```

will set PIX to -1 if the pixel at 100,100 is "on", or to 0 if the pixel is "off".

15.12 THE SRDRAW ROUTINE

PURPOSE: To draw a solid line from the graphics cursor position to a point relative to that position.

SYNTAX: SRDRAW (‹x-distance› , ‹y-distance›)

ARGUMENTS: ‹x-distance› = relative X distance from graphics cursor to endpoint.
‹y-distance› = relative Y distance from graphics cursor to endpoint.

DESCRIPTION: The SRDRAW routine draws a solid line from the graphics cursor position to the endpoint the relative distances away from the current position. The drawing mode is automatically set but not restored as part of the execution of this command.

EXAMPLE: SRDRAW (50 , -10)

will draw a solid line from the graphics cursor position to a point 50 pixels to the right and 10 pixels below the current position.

15.13 THE SRMOVE ROUTINE

PURPOSE: To move the graphics cursor position to a point relative to that position.

SYNTAX: SRMOVE (‹x-distance› , ‹y-distance›)

ARGUMENTS: ‹x-distance› = relative X distance from graphics cursor to move.
‹y-distance› = relative Y distance from graphics cursor to move.

DESCRIPTION: The SRMOVE routine moves the graphics cursor position to the point the relative distances away from the current position.

EXAMPLE: SRMOVE (-25 , 30)

will move the graphics cursor to a position 25 pixels to the left and 30 pixels above the current position.

15.14 THE SRPEN ROUTINE

PURPOSE: To draw a line from the graphics cursor position to a point relative to that position, using the current drawing mode.

SYNTAX: SRPEN (‹x-distance› , ‹y-distance›)

ARGUMENTS: ‹x-distance› = relative X distance from graphics cursor to endpoint.
‹y-distance› = relative Y distance from graphics cursor to endpoint.

DESCRIPTION: The SRPEN routine draws a line from the graphics cursor position to the endpoint the relative distances away from the current position, using the current drawing mode.

EXAMPLE: SRPEN (5 , 5)

will draw a line from the graphics cursor position to a point 5 pixels to the right and 5 pixels above the current position.

15.15 THE TCURSOR ROUTINE

PURPOSE: To set the text cursor position to the specified row and column.

SYNTAX: TCURSOR (‹column› , ‹row›)

ARGUMENTS: ‹column› = desired column position. Leftmost position = 1.
‹row› = desired row or line. Uppermost row = 1.

DESCRIPTION: The TCURSOR routine is used to set the display's text cursor position. The actual position on the screen will be dependent on the current text twindow.

EXAMPLE: TCURSOR (1 , 1)

will position the text cursor in the upper left hand corner of the current text window.

15.16 THE TWINDOW ROUTINE

PURPOSE: To set the desired text window location and size.

SYNTAX: TWINDOW (‹top offset› , ‹lines›)

ARGUMENTS: ‹top offset› = offset from the top of the screen to the top of the window.
‹lines› = vertical size of the text window in number of lines.

DISCRIPTION: The TWINDOW routine is used to set the text window for the display. The text window is specified by an offset from the top of the display and the number of text lines the window should hold. For proper operation, it is essential that the specified offset and number of lines not place the bottom of the window beyond the bottom of the display. The width of the text window is always the full width of the display.

EXAMPLE: TWINDOW (0 , 24.)

will set a text window which starts at the top of the screen and includes 24 lines.

16. THE MAGIC/L 68000 ASSEMBLER

MAGIC/L provides a built-in assembler for the 68000 processor. Routines written in assembly language may be interspersed with routines written in MAGIC/L. The unique ability of MAGIC/L to allow interactive assembly programming makes MAGIC/L an excellent environment in which to learn how to program in assembly language.

The MAGIC/L 68000 assembler supports the syntax suggested by the "MC68000 Microprocessor User's Manual" supplied with your DATAMOVER board. The major syntactical difference is that the MAGIC/L assembler requires spaces to separate the tokens in the assembly statement.

The MAGIC/L 68000 assembler is a one pass assembler. The major ramification of this is that forward jumps and forward references are not generally supported. The MAGIC/L assembler does support limited use of forward references through the use of the local symbol facility.

This chapter describes the various aspects of assembly language programming in MAGIC/L including:

- a complete syntax description
- the calling protocol between MAGIC/L and assembly
- the MAGIC/L local symbol facility
- How to access data structures from assembly routines

One final word about assembly language programming. By its nature, assembly language programming is machine dependent. The less assembly code, the more transportable a program is. We recommend that wherever possible, code be written in MAGIC/L first. Then if speed requirements are not met, the most time critical routines can be converted into assembly language.

Note: This chapter assumes familiarity with assembly language programming.

16.1 ENABLING THE 68000 ASSEMBLER

Before writing assembly code, it is necessary to activate the assembler. The routine .MAC performs this function. Once the .MAC routine has been executed, all of the routine names associated with the assembler (instruction names etc.) are accessible to the programmer.

The routine .END de-activates the MAGIC/L assembler. You should invoke .END before new MAGIC/L routines are defined in order to avoid naming conflicts. (For example the 68000 instruction EXT conflicts with a MAGIC/L function of the same name). An example of how a MAGIC/L source module containing assembly code might be organized is shown below:

```
DEFINE DEF1
   .
   .
END

.MAC
<assembly code>
   .
   .
.END

DEFINE DEF2
   .
   .
END
```

16.2 DEFINING AN ASSEMBLY LANGUAGE ROUTINE

The defining word ENTRY is used to create a named routine which is written in assembly language and callable from MAGIC/L. The ENTRY statment has the same syntax as the DEFINE statement (see: "MAGIC/L User's Manual):

```
ENTRY <name> [<routine-type>]
```

where <name> is the name to be given to the assembly routine and <routine-type> is an optional type declaration. Routine type is typically INTEGER, LONG, or REAL if the assembly routine acts like a function (i.e. returns a value). The routine-type declaration does not affect the way the routine is assembled, but allows the compiler to later treat the routine properly when it is used within an expression. Some examples of the use of ENTRY appear below in this chapter.

16.3 REGISTER ASSIGNMENT IN MAGIC/L

The 68000 has eight data registers referred to as D0 through D7, and eight address register referred to as A0 through A7. In the sections that follow, three symbols will be used to specify a general register:

An	One of the address registers
Dn	One of the data registers
Rn	Either an address or a data register

MAGIC/L reserves five address registers and one data register for internal use. These reserved registers are given alternate names in the 68000 assembler. Table 1 describes the register naming conventions, and restrictions on the usage of the reserved registers.

In general, if it is necessary to make use of one of the reserved registers, the contents of that register may be pushed onto the "RP" stack and restored before re-entering MAGIC/L.

Table 1. MAGIC/L Register Usage

68000 register	MAGIC/L name	restrictions and description of usage
D0-D6	D0-D6	may be used for any purpose
D7	VT	Must be preserved by assembly code usage: Pointer to MAGIC/L execution vector table
A0-A2	A0-A2	may be used for any purpose
A3	W	may be used for any purpose; W contains a parameter field pointer on entry to the routine. usage: parameter field pointer (see BUILD and ACTION in "MAGIC/L User's Manual) Execution vector on call to XEQ (see below)
A4	IP	Must be preserved Usage: The MAGIC/L Instruction Pointer.
A5	MSP	The MAGIC/L argument stack pointer must be adjusted by the routine according to the number of input and output arguments for the routine (see below).
A6	NXT	Must be preserved Usage: NXT is a pointer to the NEXT routine used for re-entry from assembly into MAGIC/L
A7	RP	Must be preserved Usage: This is the stack pointer register named "SP" in the Motorola assembler. It is used by MAGIC/L for subroutine nesting.

16.4 ADDRESSING MODES

The MAGIC/L assembler supports the 68000 addressing modes with a syntax similar to the Motorola assembler. Table 2 lists the various addressing modes and the assembler syntax which specifies the modes. Note that the spaces indicated in the examples are required.

Table 2. Addressing Modes

Mode	Name of mode	Syntax	notes
0	data register direct	Dn	
1	address register direct	An	
2	register indirect	(An)	
3	auto-increment	(An)+	
4	auto-decrement	-(An)	
5	reg with displacement	X (An)	1
6	index	Y (An , Rn.s)	2,3
7/1	Absolute Long	Z	4,5
7/4	Immediate data	# N	6

Notes on table 2.

1. X is a 16-bit sign-extended displacement
2. Y is an 8-bit sign-extended displacement
3. ".s" specifies the size of the index register
   .W for 16-bit index, .L for 32-bit index
4. Z is a 24-bit absolute address
5. All address references will generate absolute long addressing mode, PC relative addressing is not currently available.
6. N is a numeric value. The size of this operand is specified by the opcode

16.5 ASSEMBLER OPCODES

Almost all Motorola 68000 opcodes are supported in the MAGIC/L assembler. This section lists all of the instructions supported by the MAGIC/L assembler grouped by instruction syntax. The assembler performs extensive syntax checking which virtually eliminates the possibility of coding an illegal instruction. The next section describes the error and warning handling of the assembler.

The following symbols are used in the syntax description:

symbol	meaning
Dn	a data register
An	an address register
‹ea›	An effective address
‹opc›	one of the listed opcodes
‹label›	an address as specified by a LABEL or local symbol
.s	specifies that the opcode accepts an operation-size specification. (If the specification is omitted, a WARNING is issued. The size may be: .B byte operation .W Word operation (default) .L Long operation

tion. Lef

Set according to condition instructions

Scc ‹ea›

ST	SF	SHI	SLS	SCC	SCS	SNE	SEQ
SVC	SVS	SPL	SMI	SGE	SLT	SGT	SLE

Branch according to condition

Bcc ‹label›

BRA	BSR	BHI	BLS	BHS	BCC	BLO	BCS
BNE	BEQ	BVC	BVS	BPL	BMI	BGE	BLT
BGT	BLE						

```
-----------------------------------------------------------------
Decrement and Branch according to condition
        DBcc    Dn , <label>

DBT     DBF     DBHI    DBLS    DBCC    DBCS    DBNE    DBEQ
DBVC    DBVS    DBPL    DBMI    DBGE    DBLT    DBGT    DBLE
Note: DBRA may be used as an alternate opcode for DBF
-----------------------------------------------------------------
One operand instructions
        <opc>   <ea>

CLR.s   TST.s   NEG.s   NEGX.s  NOT.s
JMP     JSR     PEA     TAS     NBCD
-----------------------------------------------------------------
Two operand, effective address and data register
        <opc>   <ea> , Dn

CHK     DIVS    DIVU    MULS    MULU    CMP.s
-----------------------------------------------------------------
Two operand, effective address and address register
        <opc>   <ea> , An

LEA     ADDA.s  CMPA.s  SUBA.s
-----------------------------------------------------------------
Two operand, data register and effective address
        <opc>   Dn , <ea>
EOR
-----------------------------------------------------------------
Two operand, effective address and data register, either order
        <opc>   Dn , <ea>
or      <opc>   <ea> , Dn

ADD.s   AND.s   OR.s    SUB.s
-----------------------------------------------------------------
Immediate operand
        <opc>   # nnn , <ea>

ADDI.s  ANDI.s  CMPI.s  EORI.s  ORI.s   SUBI.s
-----------------------------------------------------------------
Quick operand
        <opc>   # nnn , <ea>

ADDQ.s  SUBQ.s  MOVEQ
-----------------------------------------------------------------
Move instructions
        <opc>   <ea> , <ea>

MOVE.s  MOVEA.s

-----------------------------------------------------------------
Special two register or two effective address instructions
        <opc>   Rx , Ry
or      <opc>   <ea> , <ea>

ABCD    ADDX.s  CMPM.s  EXG     SBCD    SUBX.s
```

```
-----------------------------------------------------------------
Bit Instructions
        <opc>   Dn , <ea>
or      <opc>   # n , <ea>

BCHG    BCLR    BSET    BTST
-----------------------------------------------------------------
Shift Instructions
        <opc>   Dx , Dy
or      <opc>   # n , Dy
or      <opc>   <ea>

ASR.s   ASL.s   LSR.s   LSL.s   ROXR.s  ROXL.s  ROR.s   ROL.s
-----------------------------------------------------------------
Miscellaneous instructions

EXT.s   Dn
SWAP    Dn
UNLK    An
LINK    An , # nn
TRAP    # trapnum
STOP    # stopnum
-----------------------------------------------------------------
No operand instructions
        <opc>

NOP     RESET   RTE     RTR     RTS     TRAPV
-----------------------------------------------------------------
The following opcodes are NOT yet implemented:

MOVEM   MOVEP   MOVE to/from SR,USP,CCR
-----------------------------------------------------------------
```

16.6 ASSEMBLER ERROR AND WARNING MESSAGES

The MAGIC/L 68000 Assembler performs extensive syntax checking. When an error is detected, it is immediately reported. In most cases, a WARNING message is given and the assembler will generate code and continue. If assembly code is entered from the keyboard, then no code is generated when an error is reported. The assumptions made by the assembler in generating the possibly erroneous code are included in the discussion of each error condition. The error conditions reported by the MAGIC/L assembler are listed below in alphabetical order.

Default Operation Size

This is a diagnostic message which indicates that the opcode accepts an operation size specification and none was present. Although the instruction will default to a WORD operation, it is highly recommended that the operation size always be specified. For example:

```
ADD     D0 , D0
```

would generate this message (and assume ADD.W)

Displacement Overflow

A displacement overflow is reported whenever a specified value requires more bits than are available in an instruction. The assembler will mask the overflowed value and use the truncated value as the operand. For example the instruction:

```
TST.W   70000 (A0)
```

will generate a displacement overflow since the addressing mode is limited to a 16-bit displacement.

Illegal addressing mode

An error was detected in the syntax of an operand specification. For example:

```
TST.W  100 D0
```

would cause this error.

Illegal Reference Class

An addressing mode was specified which is not allowed by this operation. This error indicates that the instruction was coded syntactically correctly, but when executed it will cause an "illegal instruction" trap. Example:

```
CMPI    # 100 , A0
```

would cause this error since "A0" is not a data alterable operand.

Multiply defined label

A local symbol was defined more than once within a local symbol range. Subsequent definitions of a local symbol are ignored.

Too many operands

Three or more operands were specified an instruction.

Wrong number of operands

The wrong number of operands was specified for the particular opcode. For example:

```
TST     D0 , D1
```

would cause this message.

16.7 THE LINKAGE BETWEEN MAGIC/L AND ASSEMBLY LANGUAGE

The linkage between routines written in MAGIC/L and those written in assembler is very straightforward. There are three aspects to the linkage:

- Accessing arguments passed from MAGIC/L
- Returning values on the argument stack
- Exiting from an assembly routine

The most important part of the MAGIC/L-assembly linkage is the argument stack. All values passed in either direction between MAGIC/L and assembly are passed via the argument stack. On entry to an assembly routine, MSP contains the address of the top of the argument stack. THE VALUE OF MSP MUST BE MAINTAINED BY THE ASSEMBLY ROUTINE. If MSP (A5) is required by the assembly routine, its contents may be saved temporarily on the RP stack.

In some situations, it is also necessary to be able to access MAGIC/L variables from assembly routines. This type of linkage is described in section 16.10.

16.7.1 ACCESSING INPUT ARGUMENTS

The calling routine will push values in the order that the arguments appear in the argument list. Because the argument stack is built in decreasing direction, that is, towards lower memory addresses, the MSP will point to the last value pushed. Other argument words will be at increasingly positive displacements from MSP. For example, in the MAGIC/L call:

```
FOO ( X Y Z )
```

assuming X Y and Z are INTEGERs, the values of Z Y and X will be at displacements 0, 2, and 4 from MSP respectively. The three argument words might be accessed using index mode like this:

```
MOVE.W    (MSP) , D0    ; Puts the value of Z into D0
MOVE.W  2 (MSP) , D1    ; Puts the value of Y into D1
MOVE.W  4 (MSP) , D2    ; Puts the value of X into D2
```

Auto-increment mode would be more efficient and would automatically adjust MSP:

```
MOVE.W  (MSP)+ , D0     ; Puts the value of Z into D0
MOVE.W  (MSP)+ , D1     ; Puts the value of Y into D1
MOVE.W  (MSP)+ , D2     ; Puts the value of X into D2
```

Remember that MAGIC/L calls are by value, not by reference (pointer). In the case of multiple word arguments (e.g. LONG or REAL values) the least significant portion of the values are pushed first and therefore have the more positive displacement. This implies that individual elements are stored on the stack in same order as they are in normal data storage. For example, in a routine called with a single LONG integer:

```
LSUB ( 100L )
```

the top of the stack has the most significant word of the argument value. The LONG argument may be referenced as two integers like this:

```
MOVE.W  (MSP)+ , D0      ; most  significant word
MOVE.W  (MSP)+ , D1      ; least significant word
```

Of course, the argument may also be referenced as a long:

```
MOVE.L  (MSP)+ , D0      ; pop a long value of the stack
```

16.7.2 RETURNING A VALUE FROM ASSEMBLY LANGUAGE

As mentioned before, the argument stack is used for passing values both to and from assembly routines. The argument stack pointer (MSP) must be properly manipulated by each assembly routine. That is, each assembly routine must pop from the stack all input values, and then push onto the stack all values to be returned. Of course there is no need to pop and push values if, for example, there is one value in and one value out (as in the case of TRIP above). In fact, the stack pointer must simply be adjusted according to the following formula:

```
new SP := old SP + ( ( <words in> - <words out> ) * 2 )
```

Thus in the case of TRIP there is no change in the contents of MSP, and the returned value simple replaces the input value. For more complex cases, the programmer must insure that the proper number of auto-decrement and auto-increment instructions are used. If index mode is used to access input arguments, it is possible that a constant will have to be added to MSP before NEXT is called.

As with the user stack (A7 or RP), single bytes should not be pushed or popped from MSP. Although the hardware does not prevent this, MSP must point to a word address (even address), when NEXT is called.

16.7.3 EXITING FROM ASSEMBLY ROUTINES

The window back into MAGIC/L from assembly language is the special macro NEXT. All assembly routines must exit by invoking NEXT.

The following example is an assembly function which triples its input argument value. Such a routine might be used like this:

```
X := TRIP ( Y )
```

TRIP might be coded like this:

```
ENTRY TRIP INTEGER              ; declare name and type
        MOVE.W   (MSP) , D0     ; Get input value from stack
        ADD.W    D0 , D0        ; Double it
        ADD.W    D0 , (MSP)     ; Add D0 back to the stack
        NEXT                    ; re-enter high-level MAGIC/L
```

16.8 LABELS IN THE ASSEMBLER

The MAGIC/L LABEL facility (see MAGIC/L User's Manual) may be used by the MAGIC/L 68000 assembler for the creation of labels that represent memory locations. The labeled locations may be used in conjunction with any of the instructions. Within the assembler, LABELs are most useful to name subroutines. Remember that all references to LABELs must follow the LABEL definition.

```
LABEL   TOPLOOP
          .
        RTS
          .
          .
        JSR     TOPLOOP
          .
```

16.9 LOCAL SYMBOLS

The MAGIC/L 68000 assembler supports a local symbol facility which allows for both forward and backward references within a block of assembly code. When used in a forward reference, a local symbol may only be the operand of a branching instruction.

Local symbol names are a single digit followed by a dollar-sign. There are ten local symbols available named 0$ through 9$. Local symbols are assigned a location by using the local symbol name with a colon appended.

The scope of a local symbol is by default between the .MAC and .END statements which activate and de-activate the assembler. The scope may be narrowed by using a .LOCAL statement. Any number of .LOCAL statements may be used between a .MAC statement and a .END statement.

Important: Forward References using local symbols are accomplished by chaining all references together. Due to this chaining, a local symbol name MAY NOT BE USED WITHIN AN EXPRESSION if it is a forward reference. Another implication of the chaining is that forward references may only be the target of branch instructions. FORWARD JMPS OR JSRS ARE NOT ALLOWED.

In the example below, a single local symbol is used as the beginning of an iteration loop:

```
.MAC
;       Fast routine to add two INTEGER arrays
;               VADD ( ARR1 , ARR2 , LENGTH )
;       DEFINE VADD
;               INTEGER ARR1 ( 1 ) ARR2 ( 1 )
;               INTEGER LENGTH
;         ITER LENGTH
;           ARR1 ( I ) += ARR2 ( I )
;         LOOP
;       END
ENTRY VADD
        MOVE.W  (MSP)+ , D0     ; pop count to D0
        MOVEA.L (MSP)+ , A0     ; Load pointer to second array in A0
        MOVEA.L (MSP)+ , A1     ; Load pointer to first array in A1
0$:                             ; Begin a looping structure
```

```
        ADD.W   (A0)+ , (A1)+   ; add datum from second array to first
        SUBQ.W  # 1 , D0        ; decrement counter
        BNE 0$                  ; if counter is not yet zero, jump back
        NEXT

.END
```

VADD is a simple routine, but is typical of assembly language usage in MAGIC/L. This code will execute several times faster than the high level equivalent routine (commented out) and allows MAGIC/L to compete in terms of speed with the best optimizing compilers.

This example has one logical flaw, it does not check for an initial value of zero for the count. The best way to do this is with the use of the DBF instruction, which also illustrates the use of local symbols in forward references.

```
ENTRY VADD      ; new version with check for zero length
        MOVE.W  (MSP)+ , D0     ; pop count to D0
        MOVEA.L (MSP)+ , A0     ; Load pointer to second array in A0
        MOVEA.L (MSP)+ , A1     ; Load pointer to first array in A1
        BRA     1$              ; go directly to DBF
0$:                             ; Begin a looping structure
        ADD.W   (A0)+ , (A1)+   ; add datum from second array to first
1$:     DBF     D0 , 0$         ; decrement counter and
                                ; branch if not -1
        NEXT
```

Local symbols may also be used to build structures similar to the IF-ENDIF or IF-ELSE-ENDIF structure in MAGIC/L.

```
        CMP.W   D0 , D1 ; compare D0 and D1
        BNE     1$      ; skip the code if not equal
        .
        .               ; code executed conditionally
        .
1$:                     ; Local symbol 1$ is defined here
        .
```

The following example is a schematic of an IF-ELSE-ENDIF in assembly language:

```
        CMP.W   D0 , D1 ; compare D0 and D1
        BNE     3$      ; skip the code if not equal
        .
        .               ; code executed if D0 == D1
        .
        BRA     4$      ; skip "<>" code
3$:                     ; Label 3$ is defined here
        .
        .               ; code executed if D0 <> D1
        .
4$:                     ; Label 4$ is defined here
        .
```

Accessing MAGIC/L variables from assembly routines is an important part of using assembly code to improve performance. MAGIC/L provides a simple means of accessing normal MAGIC/L variables and variables within records structures.

16.10.1 ACCESSING MAGIC/L VARIABLES

MAGIC/L variables may be accessed from assembly language. This is accomplished by using the PTR function (see MAGIC/L User's Manual) which will return the address of the variable. The assembler will generate Absolute Long addressing mode for the address. For example, suppose there are two integer variables X and Y. A routine which returns the value (2 * X + Y) could be written in assembly as follows:

```
ENTRY 2XY INTEGER                ; returns the value 2X plus Y
        MOVE.W  PTR ( X ) , D0   ; pick up value of X
        ADD.W   D0 , D0          ; double it
        ADD.W   PTR ( Y ) , D0   ; add in value of Y
        MOVE.W  D0 , -(MSP)      ; push on stack
        NEXT
```

This technique refers only to variables which are not within a record structure since the address of these variables is fixed. Variables within a record structure refer to different locations depending on which record is currently active. The procedure for accessing variables within records is described in the next section.

16.10.2 ACCESSING VARIABLES WITHIN RECORD STRUCTURES

An extremely powerful feature of the MAGIC/L assembler is the ability to directly reference variables which are components of a record structure. A pointer to the desired record is either passed to the routine as an argument or obtained by loading the Current Record Pointer (CRP) of the record type. The offset of a variable within the record is generated with the instant command $O. The format of the $O construct is:

$O ‹name-of-record-component›

The two word sequence is treated as an INTEGER whose value is the offset of the record component within the record structure. In the following example 2XY is implemented assuming the X and Y are variables contained in a record of type XY_REC.

```
; Assume this record has been defined and a
; variable of this type has been declared and activated
RECORD XY_REC
  INTEGER X
  INTEGER Y
ENDRECORD
```

```
ENTRY 2XY INTEGER                ; returns the value 2X plus Y
      MOVEA.L PTR ( XY_REC ) , A0
                                 ; load CRP into A0
      MOVE.W  $0 X (A0) , D0     ; load from A0 plus the offset of X
      ADD.W   D0 , D0            ; Double the value of X
      ADD.W   $0 Y (A0) , D0     ; add in from A0 plus the offset of Y
      MOVE.W  D0 , -(MSP)        ; push on stack
      NEXT
```

16.11 CALLING A HIGH LEVEL ROUTINE FROM ASSEMBLY LANGUAGE

Sometimes it is desirable to call a high level routine from within assembly code. This is accomplished by calling the function XEQ with the execution vector to the high level routine in W. The execution vector for a routine is generated by the instant command BASE (see MAGIC/L User's Manual). The MSP must, of course, be at its appropriate value. All non-reserved registers will be undefined upon return. The following example shows a call to the function WARN, which takes a warning code on the stack:

```
PARAMETER W-CODE := 777k                ; define a warning code

      MOVE.W  # W-CODE , -(MSP)         ; push code onto the stack
      MOVEA.W # BASE WARN , W           ; Execution vector into W
      JSR     XEQ                       ; call it
      ‹normal return›
```

This is a general purpose facility that works on most MAGIC/L functions and subroutines. Special classes of definitions, such as COMMANDs, operators, control words, etc. may not be invoked by XEQ because they require special processing at compile time.

17. APPENDIX

17.1 TIPS ON LEARNING MAGIC/L

The following is a list of suggestions, reminders, and notes that will hopefully be of use to those first learning MAGIC/L. Though MAGIC/L is similar to other languages, there are differences which can prove to be stumbling blocks when first starting to work with the language. You should also refer to Appendix D in the MAGIC/L Users Manual for a more extensive list of tips on debugging.

1. One of the primary differences between MAGIC/L and other languages (except FORTH) in the area of statement syntax is that a space is required between each defined word or literal. Failing to include a space will usually result in an "Undefined token" error. However, if you are a little unlucky, the missing space may result in a previously defined WORD causing a program bug.

2. The assignment operator is ":=" as in PASCAL; not "=" as in BASIC and C. In MAGIC/L, the "=" is a utility routine that prints the specified list of integer variables.

3. The comparison for equality operator is "=="; not "=" as in BASIC and PASCAL.

4. When compiling a program, don't ignore "Redefined" messages. Though MAGIC/L many of its own routines, the routine which was redefined may be in your own program. This should be investigated if you are not sure which.

5. Be careful not to redefine the variables I, J, and K. The new variable would replace the standard loop counters defined for DO and ITER loops.

17.2 MEMORY USAGE BY MAGIC/L

The MAGIC/L Language makes use of two areas of memory in addition to the memory requirements of DMXMON. The code portion of MAGIC/L occupies low memory starting at $1800. The symbol table portion of MAGIC/L occupies high memory. The location of the symbol table will depend on the actual top of memory, and how many bytes are to be reserved as specified by the MAGIC/L variable ?MRES. The top of the symbol table will fall at top-of-memory - ?MRES.

Normally ?MRES is set to zero, so the symbol table will fall at the actual top of memory. However, if an application requires some memory space protected from use by MAGIC/L, ?MRES may be set to the required number of bytes. Since ?MRES is used only when MAGIC/L performs a cold start, it is necessary to save MAGIC/L with ?MRES set before it can take effect. Executing this modified version of MAGIC/L will result in a reserved area of the specified size at the top of memory, with the MAGIC/L symbol table falling just below it.

In the "68000 world", with the popularity of UNIX and C Language, a great deal of work is done using lower case. This differs with the CODOS environment which expects commands to be in upper case only. Fortunately MAGIC/L supports both. MAGIC/L treats upper and lower case as interchangable when used in keywords and names for definitions, variables, etc. This means that you have your choice of using either upper or lower case when entering MAGIC/L programs.

In contrast, both DMXMON and CODOS expect to execute commands which are in upper case. To better support the use of lower case, DMXMON Version 1.2 or higher will automatically "fold" lower case letters to upper case in each command before executing them. This allows commands to be entered in lower case and still be executed the same as if they were in upper case. This "folding" also applies to file names which are passed as arguments in DMXMON SVC's.

To add this "folding" feature to CODOS, load or execute the FOLD_CMD.C file on the MAGIC/L Distribution disk. This could be done by the command:

```
FOLD_CMD
```

if executed from CODOS, or:

```
.FOLD_CMD
```

if executed from DMXMON. This will patch the CODOS command processor routine to fold all lower case letters to upper case prior to executing each CODOS command. If you plan to use the FOLD_CMD.C file regularly, you may wish to add it to the STARTUP.J file.

Though it makes no difference to MAGIC/L whether you use upper or lower case, MAGIC/L seems to have a slight bias towards lower case. MAGIC/L error messages will be predominantly lower case. In addition, appended file extensions are typically displayed as lower case. The MAGIC/L statements shown in this manual are typed in upper case only to make them stand out better against the lower case text.