It is Not a Fancy Language, So I Call it M'ish

by Paul Perrin

I very much liked DTM, it was very small and very fast. It also had enough functionality to make it useful as a utility programming language for DOS. (As an implementation I actually prefer DSM, but I don't have a VAX at home.)

However, although DTM made the leap to become windows-aware and could almost have been described as windows-friendly, it unfortunately never went windows-native. Even "long file names" were beyond it; the Wintel clock continued ticking but DTM was no longer in the race.

The future of "no fuss" M on the PC seemed to be pointing in the direction of MSM-WS, and with its recent "freeware" version, this really seemed to be the way to go. However, with its changed circumstances, it seems likely that it too will be frozen in its development, and time will erode any advantages that it may currently appear to have. So, as I have always known in my heart of hearts, to have the job done properly, I will have to do it myself. So here starts PSM, Perrin Standard M (oops, I mean Public Standard M). The M is the immediate aim, the P will depend on cooperation from others, and the S may take a bit (or a lot!) of extra work when the rest has taken on some shape—particularly areas that may be overlooked initially such as including standard error trapping, reverse $O and $Q, namespaces, exact value of naked indicator after evaluation of complex expressions, etc.

In the past, I have written a "global style" file system in Delphi on the PC. Further in the past, I have worked on the internals of a Modula-2 compiler on PDP's, and even before that, I worked in machine code on the Sinclair ZX81 (sold by Timex in the USA) including some disassembly of the ROM. So while an interpreter for M may be a stiff challenge, hopefully it will not be an insurmountable one. Lets see.

Overview of an M job

To run an M job we require:

- A local symbol table for the variables
- A global symbol table to interface to the globals
- A source code buffer for the code
- A "compiled code" buffer for the translated code currently being executed
- Some stacks to hold NEW'd variables, return addresses following 'DO's, return addresses from internal calls, etc.
- A program counter (PC) to show the currently executing command.

Parsing M(ish) code

A semi-colon in place of a command will cause the rest of the line to be ignored.

A command is made up of a single character, which may be immediately followed by a colon and an expression (the post condition), which will be followed by a single space and an argument list.

The argument list may consist of an argument optionally followed by any number of further arguments preceded by commas. The argument list is terminated by a space or end of line.

With a few exceptions, each argument may include a colon followed by a post conditional

A line is made up of an optional label, white space mixed with optional dot "level indicators" and a sequence of one or more commands.

Quotes and brackets (parenthesis) around arguments will have their "usual" meaning.

Source to Intermediate Code

The purpose of this function is to format the source in such a way that the rest of the M interpreter can worry about the tricky bits of M and not have to mess around with (too much) code parsing.

For this, a data structure is used that consists of an array of arrays of arrays of strings(!). The first index is the line number, the second is the command number, and the third is an array consisting of: command name, post-cond expression, argument list. Following the argument list, there will be entries for each argument and each argument's post condition.

The first command (command 0) for each line is a "pseudo" command, indicating that start-of-line processing is to be done (it also holds the label and dot level information).

So the code `'ROUT  ..IF A=10,B=3 SET:C=1 D=12' may become`

```
CODE[0][0][0]=%SOL          ; Special flag value
CODE[0][0][1]=              ; Null
CODE[0][0][2]='ROUT ...'    ; Label and whitespace
                              and dots
CODE[0][0][3]='ROUT'        ; 'Argument' 1 Label
CODE[0][0][4]=              ; Null
CODE[0][0][5]='3'           ; 'Argument' 2 Dot
                              Depth (ignored for the
                              moment)
CODE[0][1][0]='IF'          ; Command Name
CODE[0][1][1]=''            ; Post-Condition
CODE[0][1][2]='A=10,B=3'    ; Arg List
CODE[0][1][3]='A=10'        ; 1st argument
CODE[0][1][4]=''            ; 1st argument post-con-
                              dition
CODE[0][1][5]='B=3'         ; 2st argument
CODE[0][1][6]=''            ; 2st argument post-con-
                              dition
CODE[0][2][0]='SET'         ; Command Name
CODE[0][2][1]='C=1'         ; Post-Condition
CODE[0][2][2]='D=12'        ; Arg List
CODE[0][2][3]='D=12'        ; 1st argument
CODE[0][2][4]=''            ; 1st argument post-con
                              dition
```

Interpreter (Stack and PC Handling)

The core of the interpreter is based around managing the program counter (PC) and the stacks. This covers all the flow control of the program which is controlled by the commands `IF`, `DO`, `GOTO`, `FOR`, and `QUIT`. Other commands can be freestanding and probably don't need too much access to the internals of the interpreter.

The PC is implemented with four elements (line, command, argument, and "popped flag"). The first three are to index the "CODE" structure described above. The popped flag indicates whether an argument is being executed for the first time, or whether it has passed control elsewhere, mid-processing (i.e., The first time a "FOR" is executed it must initialize its counter and then (terminating condition permitting) call the rest of the line). At the end of the line, the FOR command is "popped" back to. At this time it must process the incrementing of the counter, and advance to the next argument, if this one is exhausted (terminating condition has been reached).

In fact, the purpose of the `%SOL` pseudo command is just this—first time through it just pushes its PC onto the stack, whereas when it is popped back to (from the end of the line or the termination of a FOR loop), it increments the line count to continue the processing. In practice this gives a very elegant command processing loop without the need for much "special case" processing to maintain the program flow.

So every time we process a command that may need to be popped back to, we "push" its PC on to the stack for later. In the case of the `FOR`, the index name, the increment and terminating values are also saved for re-use (when popped).

A second stack will be used for handling "`NEW`" variables. It will have a marker record pushed each time a "`DO`" is called and can have values (the original values of the `NEW`ed variables) pushed at any time, but these values will only be popped when a `QUIT` terminates the scope of a `DO`.

Most other commands will just do their job (update the symbol table or something similar), increment the argument part of the PC, and then let processing continue.

Expression Evaluator
- If null return null
- If literal return text
- If function, call, return value
- If variable, retrieve and return value
- If expression
- - Split expression on last operator (pop any brackets)
- - Call evaluator for right-hand-side of expression
- - Call evaluator for left-hand-side of expression
- - Dispatch operator

Source Code

Each routine will be held in a separate native file .mrd.

Global Handler - Overview

Just as a note to those who don't know, M globals can be roughly thought of as the following:

1.) Print a global list of any global, and draw a line between every 10 global references. These are your "data blocks." Number the blocks.
2.) Make a new list of the first global reference in each of these datablocks, and instead of writing the data against each of these references, write the number of the data block that they came from. These are now your "bottom level pointers."
3.) If your new list has less than 10 references (i.e., is only one block), then this is your "top level pointer" block (if your global had less than 100 records, then it is also a bottom level pointer block). Now you have finished! Otherwise, draw a line between every 10 references and number each of these blocks.
4.) Make another new list of the first reference in each of these blocks and again write the number of the block that the reference came from against the reference (e.g., `GOTO 3`).

Using this model, the "read and write" sections below should make some sense!

Global Handler - Implementation

This is likely to change considerably under development, but the interface from the main processing will be kept consistent. It will mainly consist of a buffer for the global reference, a buffer for a record value (for `SET`s), and a buffer for flags (similar to the result of the `$D` command).

In outline, the global data is expected to be held in standard b+ tree storage, two native files per global: .mgp block length 512 (pointer structures) and .mgd (the data record storage) block length 1024. No key compression will be implemented initially. Global references will be stored as single strings making the record key. They will have no outer brackets, commas separating subscripts will be held as nulls, numeric subscripts will be preceded by the ASCII character with a value of 1, and by two digits, a binary representation of their value.

The first block of the mgp file will be the top level pointer (which may also be a bottom level pointer). Interfacing to the global handler will require a global reference buffer, status buffer, and return value buffer.

Each block of the mgp is structured as follows (left and right block pointers may be added).

```
<blocktype>  :byte ; BLP (pointer to data file block),
                     or not
<keycount>   :int ; Number of records in block
<data>  : array [1..keycount] of
        <keylen>  :int
        <key>     :array [1.keylen] of char ; string
                    formed from subscripts
        <down>    :int
```

Each block of the mgd is structured as follows (left and right block pointers may be added):-

```
<datacount>: int ; Number of data records in block
<data> : array [1..datacount] of
        <keylen> :int
        <key>    : array [1..keylen] of char
        <datalen>: int
        <data>   : array [1..datalen] of char
```

The basic logic for global access is as follows (note the really neat trick of splitting/merging blocks on the way down, keeping the file consistent at each step, and avoiding the need for cascades of splits going back up the tree at any time):

Read:
- Get block 1
- Find required down pointer
- - Scan to last key not preceding required key
- If not BLP then
- - Get record pointed to by record
- - Iterate.
- If BLP then
- - Get specified block in the .mgd file
- - Find key
- - If found return data
- - If not found return ""

Write:
- Get block 1
- If nearly full then insert level
- - move block 1 to eof
- - create new first block with first key only, pointing to moved block 1
- Find required 'down' record
- - Scan to last key not preceding required key
- Get 'down' record (maybe in .mgd file)
- If down-record is nearly full then split
- - Move second half of down record to the end of its file
- - Insert a new pointer record to this new block
- - 'Get' correct pointer record for required key (may be

the new one)
- If not BLP then
- - Get record pointed to by record
- - Iterate 'Find required down record'
- If BLP then
- - Get specified block in the .mgd file
- - Find key
- - If found, change data
- - If not found insert.

Kill

Because `KILL` can remove many records at a time, it is more involved than the other commands. As all the references to be removed will be contiguous, the basic technique is to first isolate the dead references from other references by splitting the data blocks immediately before the first record to be killed and immediately after the last reference (keeping the BLP and higher pointers consistent with each split). This exercise is repeated at each higher level, unless we are at a level that has only one pointer to the dead sub-tree, and that pointer is not the first in the block. Discarding this pointer will kill the sub-tree. (Garbage collection or dataset compression to recover killed blocks is left as an exercise for the reader.)

Finally

This article gives some ideas of how things are expected to work and how they will be approached. A detailed examination would require a book to itself, and that time is probably better spent on the implementation. But hopefully this shows that such a project is not totally overwhelming, so if anyone reading this has experience, knowledge and time that they are willing to donate to the cause do let me know. If a basic system could be put together, and the source code could be made available to students and other interested parties, then the future development of M could be unlimited. After all, extending an existing system is less intimidating and more immediately rewarding than starting from scratch.

Since starting this article, some progress has been made towards an implementation based on this outline. The interpreter is being developed in Delphi, but does not really exploit any Delphi-specific features, so it should be easy to translate to other languages. The parsing and all the flow control commands have been implemented and seem to work well (local symbol table management and expression evaluation are very basic). An interesting spin-off of this could be a Delphi "M_Memo" component that can contain and execute M code, and on its own it would be an M job, but with access to the other controls on the same form (probably via user definable SSVN's). It could transform standard Delphi into a complete, M-oriented, native, visual development environment for Windows (and in only a few pages of Delphi code).

However, the language interpreter is only a small part of an M system. Multiple job handling, networking, device interfacing, data sharing, and security, etc., are not really touched upon here. Some of this functionality will probably come from interfacing the interpreter to the OS, but that is another project, and maybe one that others would like to take up when the basic system is in place. **M**

Paul Perrin has been programming since the late 1970's. He has been freelancing since 1987 and is always interested in new challenges. He also runs a free classified advertising webserver with a DTM backend (http://www.admatic.com) that is very fast, but no longer has any upgrade path. He can be contacted at Immediate Data Ltd, 95 Trevelyan Road, Tooting, London SW17 9LR, England.
Email: paul@idltd.cix.co.uk
http://www.cix.co.uk/~idltd/
Email relating to this article should be sent to psm@idltd.cix.co.uk.