A Forth Profile Management System

John Michaloski

Center for Mechanical Engineering
National Engineering Laboratory
National Bureau of Standards
Washington, D.C. 20234

Abstract

A Profile Management (PM) system for source code control has been developed which partitions source blocks. Profiles refer to sections of code which can be loaded by name, and whose status can be accessed through PM. Profiles can be sub-divided, and a query language allows the user to create conventional load blocks from them. All profile information is embedded as commenting structures. The package occupies approximately 3K of memory with 35 source blocks, and has been effectively used to manage an application with over 4 megabytes of source code.

Introduction

Profile Management (PM) is a powerful extension to the Forth programming environment that attempts to bridge the gap between file convenience and the realities of block management. Typically, Forth source code is mapped directly onto disk blocks and it is the responsibility of the user to manage the relationship among disk blocks off-line. Attempts have been made to develop a Forth file system to handle source code and offer a level of abstraction to insulate the user from the realities of the system implementation, [2]. Commands such as loading, backing up, and copying files provide the user housekeeping chores that at a low level are file dependent.

Unfortunately, many programming environments require something different than an alphabetized directory of the files resident off-line on disk. As a member of a small programming team generating a large amount of system software for a real-time robot control system, a simple list of a thousand program names is not helpful. Plus, the incremental and building block style of Forth encourages the quantity of code to grow beyond reasonable comprehensibility. The author's source code alone easily consumes 2 megabytes of disk space and proved difficult to manage. Much thrashing occurs within a Forth block management system which mixes the tasks of maintaining a working copy while modifying and enhancing the code for a system upgrade. At a certain point, the desire to streamline and factor Forth code becomes overpowering. This dilemma led to a rethinking of the mechanism to handle source code.

Much of the problem can be attributed to the building block programming style of Forth. Shared low level components among high level users requires some system of communication. Several problems result from this shared resource approach. First how do we load these low level routines? Loading all the low level routines is not only time consuming, but unfeasible with some dictionaries limited to 64K of memory. Second, which high level routine is responsibile for loading the shared low level component? At any point in

time, the current status of the Forth's system configuration is not readily available. Normally, most Forth systems individually handle these problems off-line using some explicity convention among programs and programmers.

PM attempts to handle the problems of off-line source code management, as well as the problem of the overall load status of the machine. The basis of this new approach is the "profile", i.e. a partition of source code on the disk. Each profile identifies a section of code that can be used throughout the loading process, plus allowing several commands to interact in the debugging phase. Further, each profile can be subdivided into smaller partitions that act as an individual profile or as a part of the parent profile. Finally, because the profiles are embedded as a commenting structure, the profile management system is only a stepwise upgrade from a Forth block management system.

Fundamental Programming Concepts

A convenient upgrade from the current Forth source block management system played a critical role in the design of PM. Without this feature, users would be reluctant to install and use the code. The key design insight was the embedding of PM within the commenting structure of the source code. Then, profiles could be produced from symbolic pattern matching done in a separate pass. The backbone of this commenting pattern is the character delimited, end-of-line comment. PM uses the character delimiter "%" as the signal of a start of a comment. Whenever a per cent sign is executed, the interpreter input stream pointer >IN is reset at the beginning of the next line. Every percent comment assumes that an entire line is sixty four characters of length. The following Forth definition performs the till end of line comment. Note that the word % is immediate, so that % is effective within the interpreter as well as the compiler.

```
: %       ( -- )
  >IN @ 64 + 64 MINUS AND >IN ! ;
IMMEDIATE                          % comment till end of line
```

Using % as the delimiting symbol, we extend this commenting convention to apply to the PM keywords %NAME and %NAME-END. In their dormant state, these keywords merely act as comments so that maintaining the profile management system requires little overhead. This flexibility enables PM to be used both on-line with keywords used for status updates, as well as off-line, to create load module images that use the keywords as comments when loading. Further, upgrading to a profile system can be a gradual process since PM and regular block loading procedures can exist together.

```
: %NAME       ['] % EXECUTE ; IMMEDIATE  % Dormant Name
: %NAME-END   ['] % EXECUTE ; IMMEDIATE  % Dormant Name-end
```

Once the PM package is loaded, these keywords perform a different function. %NAME corresponds to the opening of a profile and the %NAME-END that sequentially follows this %NAME in the disk blocks closes the profile. A more thorough description will be presented later, but the fundamental point is that when %NAME is executed, the word following it in the input stream is entered into the dictionary, and the interpreter is set ahead to the block and line following the paired %NAME-END. This requires the use of a loading word that loads a series of contiguous blocks to be modified to account for the sudden jump ahead in blocks.

```
: THRU  ( from to --)          % New version of THRU, whereby
   1+ SWAP                     % BLK is checked to determine
   DO  I  LOAD                 % if it has been modified during
       R> DROP BLK @ >R        % the load. For instance,
   LOOP  ;                     % by incrementing ahead.
```

Extending this insight into contiguous block loads, the following definitions were added to enhance the load process, –v and vSKIP. vSKIP is a word that skips or ignores the next block, (i.e. don't interpret). This is useful for disk blocks that contain numeric, non-ASCII data as a part of some virtual memory capability. The word –v acts as a continuation signal. When executed, –v immediately resets the input stream to the first character in the next block. Thus, continuation works within the contiguous block load while interpreting as well as compiling. This is especially useful for lengthy source code compilations that may need to cross block boundaries.

```
: vSKIP 1024 >IN !                % Used with new version of THRU
    1 BLK +! ; IMMEDIATE          % Don't interpret next block

: -v                              % Used with new version of THRU
    0 >IN !                       % Continue with next block by
    1 BLK +! ; IMMEDIATE          % resetting >IN immediately
                                  % compiling or interpreting.
```

Profile Management Technique

Profiled names are added to the dictionary via a pairwise addition of %NAME and %NAME-END chunks of code. The action of the word %NAME is as follows: the word following %NAME in the input stream is added to the dictionary and liked into the named profiles list. Parameters saved with each profile are the source block of the name entry, the character position within the block of the line following the name entry, and the name, which is marked "N", not entered. Then, %NAME contiguously scans ahead in the disk blocks until the atching %NAME-END entry is found. At this point, the block and character position of the matching %NAME-END entry are saved in the named profiles parameter list. Finally, the interpreter is reset to the line following the position of the %NAME-END. The overall data structure of a names entry list within the dictionary is illustrated below. The level entry in the data structure will be discussed later when the issue of embedded named profiles is presented. HERE represents a field for the starting dictionary address of each profile entry. This feature is useful for dynamic interaction with the FORTH dictionary.

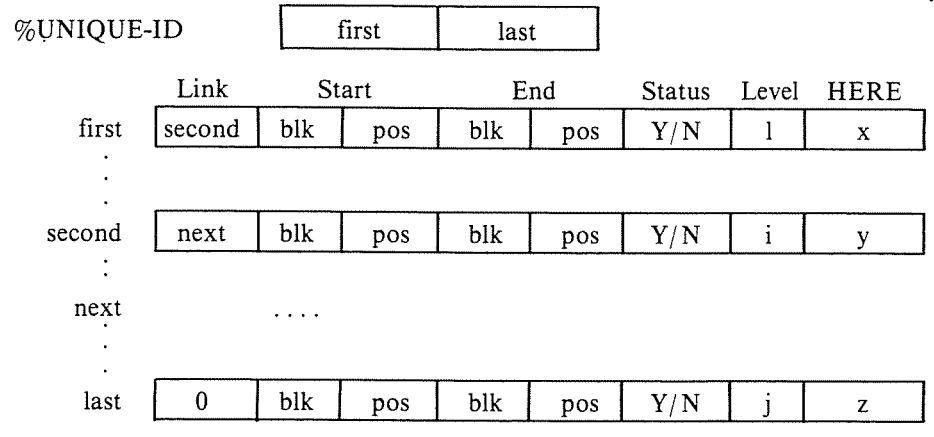

Figure 1. Profile Data Structure

Building a Profile List

Additions of named profiles can be done functionally within an extended THRU or symbolically via the PM command NAMES. The difference between the two methods of profile creation is significant. The extended THRU interprets and compiles words outside

limits of the %NAME and %NAME-END matching pairs. The NAMES command takes as input a beginning block and an ending block. Scanning this range of blocks, %NAME and %NAME-END matching pairs are symbolically extracted to create profiles. All other code with the NAMES command is ignored. Both methods of the profile generation ignore the contents between the %NAME and %NAME-END pair. The following example illustrates the similarities and differences.

Start of contiguous blocks of code
BLOCK = 10

```
        : defl  ... ;
%NAME  sampleA  "rest of line commented out"
        : def2  ... ;
        : def3  ... ;
%NAME-END sampleA  " rest of line commented out"
        : def4  ... ;
```

BLOCK = 20
End of contiguous blocks of code

COMMAND OPTIONS :

1) 10 20 THRU
2) 10 20 NAMES

Figure 2. THRU vs NAMES Profile Creation

When the extended THRU option is used, the profile entry "sampleA" is added as well as the colon definitions "def1" and "def4" to the dictionary. However, when the NAMES option is used, the profile entry "sampleA" is the only addition to the dictionary. Note that in both cases "def2" and "def3" were not added to the dictionary. These definitions are a part of the named profile and are loaded by invoking the word "sampleA."

To load the contents of this profile, simply invoke the name "sampleA." Every profile performs the following sequence. First, the input stream status is saved. Second, the interpreter is set to the line following the profiles name. This enables fractions of blocks to be loaded. For this reason, the word "def1" will not be recompiled with an extended THRU or compiled with a symbolic NAMES load. Third, each block within the range specified by the %NAME and %NAME-END partition is loaded. During this profile interpreting, the definitions "def2" and "def3" will be compiled. When the source word %NAME-END is executed, the input stream is reset to the end of the block so that interpreting terminates. Thus, the definition "def4" will be ignored. %NAME-END completes the loading with a "Y" marking of the profile. Finally, the profile loader restores the input stream to the original status to resume interpreting. The save and restore operation permits profiles to execute in a variety of environments. Profiles can be listed sequentially within a block or can cross-reference another profile.

Profile Status

With the creation of a profile list, status of the names can be displayed to the user. The PM command ?NAMES allows the user to examine the current profile configuration. This command displays all the complete names in the list and describes each name's starting and ending block, and the status, (i.e. "Y", yes the name has been loaded or "N", the name has not been loaded.) From the previous example of a profile with one entry the following list will be created:

```
?NAMES
sample A                    10  TO      20    N
```

Should "sampleA" now be invoked, its execution will change the status from "N" to "Y" as well as load the words "def2" and "def3." The ?NAMES command will now display:

```
sampleA

?NAMES
sample A                    10  TO      20    Y
```

Any subsequent attempts to invoke "sampleA" will not allow the profile to be loaded. This screening process provides the basis for the prerequisite profile management. Since profile generation uses two passes, the first pass creates a profile list so that name invocation during the second pass is independent of any ordering of profiles. Thus, prerequisite loading profiles can be created within a profile without deference to actual profile loading order. The following code outline illustrates the use of prerequisites naming and the independence of profile order.

```
%NAME part1
        part3
%NAME-END part1
          . . .
%NAME part2
        part3
%NAME-END part2
          . . .
%NAME part3
          " prerequisite code "
%NAME-END part3
```

Excluded from this prerequisite ordering format are profile loads using circular reasoning, (i.e. two profiles that call each other.) Although embedding a name invocation offers a flexible mechanism for dealing with load prerequisites, this option is available only when PM is on-line, that is, resident in memory at the time of the load. Off-line load files created by the source code generator of the PM, which will be discussed later, do not account for cross referencing of prerequisites specified within a profile. Therefore, prerequisite specification should be exclusively limited to on-line PM.

Profile Compilation Monitoring

While designing PM to address the static relationships among load modules is important, PM includes dynamic program development tools to assist the programmer while developing software. The command DEBUG was added so that the user can watch the progress of a PM load. As blocks load, their status is echoed to the toggles, so that every other invocation activates the echoing action. Another area of improvement was the addition

of an interpreting enhancement that monitors the dictionary free space pointer, HERE. Additions and removals of dictionary object code, plus resuming compilations, unfold from this slight upgrade.

The command KILL removes a profile that has been added to the dictionary space. KILL also resets the PM list pointers and status of those profiles loaded after the killed profile. In other words, after several profile name loads, should the first name be killed, all those names loaded that followed the original name will have their status reset to "N", not loaded.

Further, as a debugging tool, the user could load in a profile, execute some code, and as necessary, KILL this error tainted code from the dictionary, repair the code and load in the profile once again. Because this load, go, repair, kill load sequence is unfortunately prone to occur over and over, several features were incorporated into PM to reduce this headache. First, after the first invocation of a profile, that name will act as the default on any subsequent PM core commands. This allows the programmer to use longer file names without remorse over extended typing. Second, the command GO was added to exploit this default concept to load a profile. Thus, the debugging sequence has been streamlined to the following series of commands:

```
name1                          ( load in default name once )
(repair code)
KILL                           ( reset the dictionary )
(repair code)
GO                             ( load in the last accessed PM name )
KILL                           ( reset the dictionary )
(repair code   )
GO
```

While this approach to code development is helpful, many times simple compile time errors within loads would necessitate a KILL and a complete reload of the profile source code. The addition of a dictionary and a block tracking mechanism within the interpreter of PM allows for two more timely debugging tools. Whenever an error stops the loading cycle, the user needs only enter the command ED, and the block editor will grab the block that was interpreting when the error occurred as the edit block. Once the user has corrected the error, the PM command RESUME will start the interpreting from the beginning block of the error. The reason this capability is available, is that before interpreting any new block, HERE and BLK are saved. Thus, if an error that requires modification of the current loading block occurs, RESUME forgets back to HERE, and uses the saved BLK as the beginning of the interpreter loop and fetches the last block from the profile of the name to terminate interpreting. The addition of these PM dynamic compilation features all stem from an enhancement of the FORTH word INTERPRET.

```
: INTERPRET     ( last 1st -- )
  DO I  BLK !                  % Set interpreter block number.
     I SCR !                   % Set editing block number.
     I pmBLK !                 % "Save current block and
     HERE pmHERE !             % dictionary pointer for RESUME.
     ?DEBUG IF I. .S THEN      % Check for debug monitor.
     0 >IN ! INTERPRET         % Then interpret code.
     R> DROP BLK @ >R          % Check for reset blocks during
  LOOP ;                       % interpret.
```

Finally, software development is a growing process so that the static block boundaries originally defined are soon exceeded. For this reason, the command RELINK was included so that as programs grow, the last block of a profile can be reset to reflect this change. After allocating new space, a RELINK command scans from the original %NAME to find its corresponding %NAME-END to expand (or shrink) the profile.

Profile Extensions

So far, the discussion of profiles has been limited to an on-line system of a single dimension. However, various enhancements permit Profile Management to adapt to the various requirements of a users programming environment. Although each of these enhancements is modest, the total package performs a variety of tasks that lend a Forth programmer a useful helping hand.

Incorporating recursion within the defining sequence allows profile management to become a multi-dimensional hierarchical management system. Now, instead of loading entire profiles of code, the user can selectively add portions of profiles as required.

Within this hierarchical structure, Profile Management need not be restricted to single loading modules. The opportunity for repeated loading profiles offers a convenient vehicle for help, diagnostic, and other status reporting loads. Although slower due to the necessity of interpretation, these repeatable load modules will not require dictionary space.

The addition of code generation permits Profile Management to negotiate the constraints imposed by a system's memory. Profile images created by a code generator reside in disk blocks that can later be accessed on-line as required. The heart of the code generator is a query program which systematically interrogates a user concerning the need to load a particular profile. For each "yes" response to a profile name query, the code generator produces a text entry on disk matching this request. The generated text is a range-of-blocks load derived from the parameters of the selected profile. Overhead involved with maintaining load modules is limited to the symbolic commenting style of Profile Management. Thus, PM attempts to offer a simple off-line query system to handle the creation of a variety of load modules.

Profile Recursive Embedding

Profile Management enables named profiles to embed definitions recursively. %NAME match causes a named entry to be inserted in the profile list. During the search for the matching %NAME-END, a %NAME pattern match will trigger an embedding addition to the profile list. Each embedded %NAME and corresponding %NAME-END pairing are restricted to complete containment within the outer profile.

To help clarify the powers of embedding profiles, consider the following profile outlined example.

```
%NAME  application
       ...
       %NAME part1
              ...
       %NAME-END part1
              ...
       %NAME part2
              ...
              %NAME part2,1
                     ...
              %NAME-END part2,1
                     ...
              %NAME part2,2
                     ...
              %NAME-END part2,2
       %NAME-END part2
              ...
       %NAME part3
              part1                    % prerequisite : part1 loaded
                  ...
%NAME-END application
```

The initial profile setup occurs via a functional THRU or a symbolic NAMES command. Then, ?NAMES command will display a profile list with each offspring of a parent indented to highlight the embedding. In this example, the ?NAMES command will display the following profile list:

```
10  18  NAMES
?NAMES

application                  10  TO    18     N
   part 1                        10  TO    12     N
   part 2                        13  TO    15     N
      part 2,1                       13  TO    13     N
      part 2,2                       14  TO    15     N
   part 3                        16  TO    18     N
```

After the initial profile list setup, the status of each profile name is "N", not loaded. Should we wish to load the entire application package, the outer profile "application" will do so. As we are loading, subsequent %NAME words provide an embedded aspect. Each %NAME within the input stream executes a dictionary search for the profile name immediately following. Should this profile exhibit a loaded status, the input stream will be reset to the position of the line and block following the corresponding %NAME-END. Profiles with a non-loaded status reset the input stream to the line following the word %NAME. Thus, %NAME performs a variety of functions dependent on the status of the profile.

Loading continues in this manner until a %NAME-END is encountered. With matching names, the status of the profile is marked "Y", loaded. After all the embedding has unwound so that the initial level is reached, the input stream is reset to the end of the block, thus

terminating the profile load. The invoking of the word "application" would have the following effect on the profile list.

```
application
?NAMES
application                    10  TO    18     Y
   part 1                          10  TO    12     Y
   part 2                          13  TO    15     Y
      part 2,1                        13  TO    13     Y
      part 2,2                        14  TO    15     Y
   part 3                          16  TO    18     Y
```

Suppose the user had only desired "part2,2" of the application. Then instead of specifying the entire "application", the profile name "part2,2" would exclusively load this portion. No other items are required or loaded. The ?NAMES command now displays this status:

```
part2,2                        ( load part 2,2)
?NAMES
application                    10  TO    18     N
   part 1                          10  TO    12     N
   part 2                          13  TO    15     N
      part 2,1                        13  TO    13     N
      part 2,2                        14  TO    15     Y
   part 3                          16  TO    18     N
```

Profile List Branch

Profile lists can grow to be quite large, so the command USE allows the user to set a given profile name as the start of the profile list. The profile list stops before either the end of the list or the next profile of the same level as the USE. The syntax for this optional command is:

[USE profile-name]

This command is temporary, that is, it only lasts for the next profile list access. An example of this subprofile list option is illustrated with the ?NAMES command.

```
USE  part2   ?NAMES        ( use part2 brancj )

part 2                         13  TO    15     Y
   part 2,1                        13  TO    13     Y
   part 2,2                        14  TO    15     Y

?NAMES                         ( default to full profile list )

application                    10  TO    18     Y
   part 1                          10  TO    12     Y
   part 2                          13  TO    15     Y
      part 2,1                        13  TO    13     Y
      part 2,2                        14  TO    15     Y
   part 3                          16  TO    18     Y
```

Multiple Load Profiles

So far, profiles have been limited to one load per profile. Suppose we wanted to create a named profile that only displayed status information. Unfortunately, this named profile could only be loaded one time, thus rendering it ineffectual for repeated status updates. For this reason, the profile type %NAME-LOAD was added to the PM system. %NAME-LOAD is paired with a %NAME-END to create an entry in the dictionary as well as the profile list that differs from %NAME in that it can load its profile repeatedly. The status of a %NAME-LOAD entry is signified by an asterisk "*". For example, the following code outline includes %NAME-LOAD profiles at the outer level and embedded within %NAME profiles.

BLOCK 20

```
%NAME application

        %NAME   partl
                " some code to be explained "
        %NAME-END   partl

        %NAME-LOAD   a-help
                " some help report "
        %NAME-END   a-help

%NAME-END application

%NAME-LOAD   help

        %NAME  helpl
                ...
        %NAME-END   helpl

%NAME-END help
```

BLOCK 40

The corresponding ?NAMES profile list to this code outline now includes the asterisk denoting the %NAME-LOAD profile.

```
?NAMES
application          20  TO    30     N
   part 1                20  TO    23     N
   a-help                23  TO    30     *
help                 30  TO    33     *
   helpl                 30  TO    31     N
```

At loading time, Profile Management treats embedded %NAME-LOAD profiles differently than %NAME profiles. %NAME-LOAD profiles can only be loaded by an explicit reference to the profile name. That is, embedded offspring %NAME-LOAD profiles will not be loaded within their parent profile load unless an explicit reference to the offspring profile name is invoked. The rationale behind this protocol is that %NAME-LOAD profiles are designed for report help and status updates that are not important during loads. The following code outline illustrates the use of a help report to be given at load time of "application."

```
: "   34  WORD DUP 1+          % assume immediate string
        SWAP  C@  TYPE  ;      % print is paired double quotes

%NAME application
      a-help                   % Note the two pass invoke.

   %NAME-LOAD  a-help
      "Issue some printed text"   % Issue message with
      CR " And Variable Status "  % status display.
      VAR-X @ .
   %NAME-END  a-help           % a-help invoked, not during
                               % an embedded load.
%NAME-END application
```

However, at any time help is required, "a-help" will be available for inspection. This help could be given regardless of the load status of any part of the application program.

Profile Selection

At the basic level, each profile has been treated as an individual component, as opposed to part of an entire list. A query system addition to PM makes use of the list to systematically question a user about selecting profiles to load. This query process was included in PM for users of other people's software to have the ability to select component parts. Combined with prerequisite profile embedding, the query system is a handy user interface load mechanism. A typical query sequence would have this format:

```
Use? (Y/N)      profile-name
```

The user would then supply a capital Y to signify, "yes" load this profile or a capital N, "no" don't load this profile. A "yes" response skips over all the offspring of a profile, while a "no" response skips to the next name in the list which may or may not be an offspring.

The query system coupled with a code generator proves even more valuable. Now, a user can select profiles to load which will produce an image on disk. Each "yes" response within the query system moves a line of text to a disk block. An example of this line of code generated text looks like this:

```
10  18  THRU     % application
```

Input to the code generator is a starting block number. The code generator keeps track of the line position on the disk as lines of text are being produced. The code generator does not check to see if the block it is overwriting is important, so caution must be exercised in choosing block numbers. In addition, profiles that do not fall on exact block boundaries are not partially loaded. The overall format of the select command is

```
[ USE  profile ]  [ TO>DISK  block# ]  SELECT
```

where the user has the option to set the profile list branch from which to select and the option to generate code on disk starting at the given block number.

Commands

A summary of the commands and important keywords are as follows:

i) KEYWORDS : embedded within source code

%NAME	"name"	:	denotes beginning of a name unit
%NAME-LOAD	"name"	:	denotes beginning of a name load unit
%NAME-END	"name"	:	denotes end of a name unit

Each "name", when later interpreted, will load the range of blocks of a name and name-end unit.

ii) COMMANDS : creating a profile

n	n+i	NAMES	
n	n+i	THRU	: Within this range of blocks the %NAME %NAME-END pairs will be extracted to create named entries. Code outside these named units will be loaded.

COMMANDS : dynamic

1) [USE name] ?NAMES

Display current names in profile giving name, start and end blocks, and a load status. Optionally, USE assigns the name unit to profile, as opposed to the entire PMS.

2) KILL [name]

Kill the profile name so that the dictionary is reset to the status before the profile was added. As a side effect KILL any profiles that have been added after this name. If no name specified, the default is used, i.e. last loaded name.

3) GO [name]

Load into the dictionary the profile name. If no name is specified, the default is used, i.e. last loaded name.

4) RESUME

Resume loading into the dictionary the last loaded name from the point of error. Forget dictionary back to the start of this block and then begin compiling from the beginning of this block.

5) DEBUG

Toggles a compiling monitor which displays the last successfully completed block and its stack contents.

6) RELINK [name]

Resets the final block pointer parameter field. Starts scanning for the final block from the original block parameter fields contents. Allows profile names to grow or shrink in size without a complete redefinition of the profile list. If no name is used, the default name is used, i.e. the last name loaded.

7) [USE name] [TO>DISK block#] SELECT

Named units of ranges of blocks are selectively loaded. Interactively each name is presented to the user to load. With each Y response, the name and all descendants are assumed loaded. With each N response, the next name in the profile is used in the query.

TO>DISK block# delays the actual load and instead creates a load file starting at the block given by the user. Each Y response will post another range of blocks to load on disk. The actual start and finish block numbers THRU are written on the disk so the PM need not be present when loading. Optionally, USE assigns the name unit to select.

Summary

Profile Management offers a powerful extension to a Forth programming environment. Essentially PM is an off-line file system that provides status information of source code location on disk as well as on-line information concerning what named source code modules have been loaded. PM utilizes a two pass approach to scan through embedded comments to extract the beginning and end disk blocks of a range of source code grouped together as a unit, known as a profile. From this information, the user can either 1) continue working on-line and selectively load names, or 2) create an off-line load profile based on the current name profile.

Although PM displays a number of useful functions, extensions to any file system could perform them. The current version of PM stresses an efficient upgrade from the current Forth disk block management style as well as dealing with the building block approach of Forth. With many microcomputer systems headed toward a multi-tasking environment, the advent of profile-like status could prove advantageous when dealing with several programming systems simultaneously.

References

[1] L. Brodie, *Starting FORTH*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[2] P. Reese, "A Disk Operating System for FORTH, An In-depth Look at How a DOS operates," in *BYTE* April, 1982 pp. 322–341.

Mr. Michaloski received a BS in Mathematics from the University of Maryland in 1976, and an MS in Information and Computer Science from Georgia Tech in 1978. His work at NBS has been involved with the upper levels of a real-time robot control system. Currently, his efforts are directed at automating the metaknowledge about programs.