Stack Frames and Local Variables[1]

George B. Lyons
280 Henderson St.
Jersey City, NJ 07302

Abstract

The stack frame technique for dynamic local variable storage in Algol and descendant languages may be adopted in Forth to expand the number of elements easily handled directly on the stack, reducing the need for variables. The required extensions to Forth can preserve the interactive quality of Forth by using the parameter stack exclusively and by using a built-in set of local variable names rather than an argument list compiler. Special techniques such as wildcard names, code pointer arrays, and modifications of `TO` can reduce the overhead involved.

Forth provides a computation stack to store temporary data along with stack operators to rearrange that data, but lacks a facility found in Algol and similar post-Fortran languages to allocate variable storage on the stack. In these languages variables so allocated are local to a particular procedure and referred to loosely as "local variables", even though that term encompasses statically as well as dynamically allocated variables. When their storage is dynamically allocated such variables allow calling procedures recursively and recovering memory occupied when procedures are completed. The allocation is accomplished by pushing on the stack a block of space, called a stack frame, to hold all the variables for each recursive call to a procedure, and dropping the block when returning from each call. Each variable is then represented in object code by an offset relative to the frame boundary instead of by an absolute memory address. Data is fetched from and stored into the framed memory the same as ordinary variables, in contrast to Forth's emphasis on rearranging the order in which data is stored on the stack.

CPU makers have supported dynamic local variables with hardware registers specifically for base relative addressing on the stack, as in the Burroughs mainframes and the Intel 8086 microprocessor. In Forth, however, local variables have not been used much, at least outside of Europe; as early as 1977 Paul Bartholdi at the Geneva Observatory introduced Forth extensions to employ local variables on the stack[2]. They are rarely found in commercially available systems, and the Forth Standard[4] not only does not include them, but prohibits indexing into the stacks of the Forth pseudo-machine as well.

More recently, and with increasing frequency, a number of papers have appeared expressing a need for local variables in Forth and describing a variety of Forth extensions to provide them [3,4,10,11,12,15,16,17]. This paper develops further the particular technique of using a set of local variable names built into the system and available for use in all procedures, as opposed to defining new local variable names in every instance. Methods for implementing this technique are examined from the standpoint of achieving general applicability, minimizing the costs incurred, and preserving the interactive features of the Forth environment.

[1] An extension of remarks at the FORML conference in Taiwan, Republic of China, September, 1984.

The Role of Local Variables in Forth

Because local variables have not gained wide acceptance among Forth users their role in programming requires explanation. Here I am concerned only with their operational role in relation to stack frames. In broadest terms that function is to allow more data to be handled on the stack, rather than in other data structures, than are conveniently handled with Forth stack manipulations. The stack can then replace the permanent storage of numerous scalar variables, and also replace other more complex forms of dynamic storage such as a Pascal style heap, for arrays and other lengthy structures, when the full complexity of such facilities is not needed. This paper concentrates on storing relatively small sets of scalar values on the stack, leaving to future work analysis of larger structures in detail. However, for the greatest potential use stack frame facilities should provide room for large data structures, and some systems in use limit the size of stack frames, e.g. Duff's "methods stack"[4].

Forth stack operations work well for certain types of calculations, for example, evaluating algebraic expressions, where all the intermediate results can be pushed on the stack in precisely the order in which they must later be pulled off. Stack operations become inconvenient when several values must be preserved for repeated use in a calculation with both access and modification in a random order. Code to handle even only four or five entries on the stack with `DUP`, `SWAP`, `ROT` etc. while preserving their values becomes complex, placing a burden on the cpu to run it and a burden on the programmer to figure out how to write it (beside which readability problems pale in comparison).

Juggling several items on the stack usually involves shifting data onto the return stack with `>R` or using `PICK` and `ROLL`. `>R` is supposed to be used with caution, yet I find it necessary quite frequently, a sign that other tools are needed. `PICK` and `ROLL` are such tools but are not impressive in terms of efficiency. They both require a literal argument to be compiled, and `ROLL` reads and writes every element above the target item on the stack. `>R` works easily when only one value is shifted, yet even that value becomes buried under the parameters of any `DO` loop subsequently entered. Shifting more values makes a program resemble the Towers of Hanoi game (where disks stacked on needles are rearranged only by moving them onto other needles...). Moreover, moving data between stacks has no obvious relation to any algorithm being encoded.

Variables thus remain the most reasonable place to store more data than can fit comfortably on the stack, but are oriented toward data accessed by many procedures, requiring longer term storage and global definition. For small sets of temporary values for a single procedure they have disadvantages; they require writing declarations, consume permanent storage both for the data and their headers, and require extra code to move their values to and from the stack when used recursively. Local variables fill the role of global variables in supporting a more random sequence of access than do stack manipulations, while retaining the stack's function of dynamic storage allocation.

Common Local Variable Names

The previous papers on this subject differ in their interpretation of the role local variables should play in Forth. Glass[6] and LaQuey[11], for example, express a desire to eliminate stack manipulations entirely and to include a declaration of named arguments in Forth definitions, making them more resemble procedures in other languages. Bartholdi[2] (at least in his early paper) and Bowhill[3] go only so far as to define some Words, such as Bartholdi's `PAR0`,`PAR1`,`PAR2`,... which are used in common by all procedures to access successive elements of a stack frame. This emphasizes providing random access to the stack rather than the declaration of variable names.

The second approach strikes me as being most Forth-like, in the sense that a minimal solution is sought that addresses the most important problem, thus conserving the available resources. The immediate problem with `DUP`, `SWAP`, `ROT` etc. is not that they do not describe the data manipulated, but that they cannot easily access more than three values. The use of common names also eliminates

the storage for ordinary variables defined in great numbers, keeping their values on the stack and eliminating their headers altogether, along with their associated declarations.

A need for descriptive variable names is still present but is lessened, at least in my opinion, by the resemblance of much programming to mathematics. In mathematics it is common to use the same familiar symbols such as x,y,z or i,j,k in different contexts; creating unique names everywhere can actually be a burden. Common, reused symbols can be kept short, which contributes to readability in its own way. Fully descriptive names are difficult to achieve, because there is always some context which must be learned before even supposedly descriptive names can be understood.

Declaring unique names everywhere has disadvantages of its own. To reduce the storage required by many variables Glass[6] and others use systems for making the headers exist only at compile time, and even only within a single definition. The compiler handling the declarations in these systems adds to the size of the Forth system. Making the variable names available only at compile time departs from the open, flexible, and interactive quality of Forth, due in part from being able to access whatever you want whenever you want. Moreover, my objective here is not to replace Forth's stack operators, which have been found useful for many applications, but merely to supplement them where they are inconvenient for particular problems.

Storage For Local Variables

Local variables can be stored on the parameter stack, the return stack, or a third stack created just for locals, and designs have been presented using all three [2,3,4,8,15]. Using the return stack (as in [15]) is incompatible with the interactive interpreter mode that is central to the Forth environment, where the return stack is needed to control execution of the interpreter. Like `>R` and `R>`, return stack frames can be used only within colon definitions, which then cannot be tested stepwise or broken into smaller pieces, at least not without some special debugging system[15]. Some parameters, such as those of a `DO` loop, can reasonably be kept on the return stack because the associated code must be compiled to define branch destinations, but the more Forth's facilities depend on the return stack the more Forth becomes just another compiled language.

Some systems[2,8] have stored values on the parameter stack but have also involved the return stack for frame control information. One control item is the pointer to the base of the previous stack frame, which must be saved when a new frame is created, to allow a return into the old frame. Another item may be an execution address for a Word which will close a stack frame automatically upon exit from a definition, seeking to make creation and removal of a frame an inherent part of colon definitions. Burdening definitions this way adds even more inflexibility in addition to weakening the interpreter.

A third stack for locals (as in [3] and [4]) preserves Forth's interactive quality but appears to be less efficient than storing locals on the parameter stack. First, when argument data passed on the stack are to be framed for repeated use, they must be moved to the other stack, instead of just leaving them on the stack where they are first generated. Second, the storage for the other stack must be managed as part of the memory map, adding code to the system. If there are going to be large arrays on the other stack, space must be divided between that stack and the parameter stack. Pointers to a separate local stack must be initialized and managed in conjunction with system aborts and so on. The parameter stack, by contrast, is already taken care of in the system.

These inefficiencies of a third stack apply to the return stack as well. They may also relate to the design of hardware implementations of Forth. The return stack is normally relatively small and accessed only from the top, in contrast to the parameter stack which can potentially hold long lists of values even when only the top two or three are used at a time. The return stack is therefore an excellent candidate for a completely hardware or on-chip implementation, and one with minimal data paths, making it useless for stack frames. The parameter stack already requires a complex form even when in hardware and may allow for overflow into separate memory, and is thus a more likely candidate for hardware stack frame support.

Of course, in some cases moving parameters off the stack can allow a routine to run faster as, for example, when parameters are used repeatedly to generate a long list of elements built on the parameter stack to be returned as results. Getting the parameters out from under the results at the start eliminates having to copy the whole list over the parameters at the end. The more usual situation is for a routine to take more data on the stack as arguments than it returns as results, as long as there are so many arguments and temporary parameters that a stack frame is required to handle them in the first place. Many procedures, after all, do not return any results, or when returning lengthy results do so by storing them into a data structure off the stack. Without more evidence on the balance of arguments and results the cost of using more than the existing parameter stack seems unjustified; ordinary variables can always be used in the special cases where the parameter stack is inconvenient.

A Parameter Stack Frame System

The most obvious way to create stack frames is just to mark the current position of the parameter stack top with a base pointer (denoted BP) whenever base relative addressing is desired. To allow nesting of frames the value of the base pointer must be saved whenever it changes, and the most obvious place to save it is on the parameter stack at the place marked. Figure 1 illustrates this for a Word taking three arguments on the stack when called and creating a stack frame to handle them as variables, along with two additional local values. As the Word is executed Arg1,Arg2,Arg3 are on top of the stack with the stack pointer at position (a). Creating the stack frame pushes BP on the stack at (d) and copies the stack pointer into BP. Processing within the Word pushes and drops additional elements above the variables, leaving the stack pointer at indeterminate position (b) while the frame is open.

Figure 1. A Parameter-Stack Frame

(c)	Arg3	(3rd on stack before creating frame)	<-- P-3
	Arg2	(2nd on stack before creating frame)	<-- P-2
(a)	Arg1	(top stack element before creating frame)	<-- P-1
(d)	BP	(saved value of base pointer)	<-- BP
	Var1	(first value pushed after framing)	<-- P1
	Var2	(second value pushed after framing)	<-- P2
(b)	...	(more values pushed after framing)	<-- P3,4,...

The stack frame defined in Figure 1 has no explicit top or bottom; with appropriate code any element pushed before or after allocating the frame is accessible via base-relative addressing. The range from Arg3 to Var2 is only implicit in how the elements are used, just as there are no explicit argument lists in Forth generally. Yet the simple model of stack frames just described is new; previous systems [2,8] save BP on the return stack and use BP to mark an explicit bottom of the frame. Saving BP in the frame itself does have an effect, which is a distinction between arguments to a Word and temporary variables created within the Word. The working variables above BP can be redefined during calculation, including adding more variables, while the argument list is fixed, at least in size, and is unlikely to be reused for different data while the frame is open. This restriction and the fact that offsets from BP are negative for arguments, while positive for additional variables, seem to me to be very minor effects compared to the effects on system control of using the return stack.

Frame Handling Words

For some Words to allocate and deallocate the stack frames described above `S[` and `]S` are suggested, pronounced "frame" and "unframe", and used in the form " ... `S[` ... m n `]S`" with literals m,n explained below. Because the return stack is not affected such expressions may appear outside as well as inside colon definitions, can appear repeatedly inside such definitions, and can be factored, opening a frame in one definition and closing it in another. Though the symbols `S[` and `]S`, like `@` and `!`, have to be memorized, they are short and suggestive of "opening" and "closing". Their hieroglyphic quality can be used to alert the reader that a low level system operation is meant, and to avoid confusion with descriptive names in an application program, such as one about video or photographic frames. As fundamental data handling tools, these Words should be defined in machine code. BP should be a processor register where possible, to maximize the speed of base-relative addressing to access frame elements. Indeed, without adequate processor support for base-relative addressing, direct addressing of static variables will be much faster than either dynamic variables or conventional stack operations when those are complex.

`S[` simply pushes and sets BP. `]S` restores BP but in addition removes framed data from the stack and replaces them with a selected number of elements at the top of the stack to be left as results. `]S` is derived from a similar syntax "m `ARGS`...n `RESULTS`" introduced earlier by Jekel[8]. Argument m is the number of initial arguments, or elements below BP, to be dropped, and n is the number of results to be pushed. In the type of stack frame used here m and n are not needed until closing the frame and are thus both arguments of `]S`. The results to be left are taken from the top of the stack (under m,n) and copied to the point m cells deeper than BP in the frame, while first restoring BP so as not to destroy its saved value. In Figure 1, returning one result would copy the value at point (b) to point (c) and make (c) the top of stack. All stack elements above BP are automatically dropped without knowing their count. While `]S` does a lot of work it does it all in one step, compared to managing the stack one piece at a time with numerous stack operations. It may be worthwhile to have special variants of `]S` which build in specific values of m and n, for example when m or n is zero.

One more Word that would be useful in opening a frame is one to allocate space by adjusting the stack pointer. The phrase n `LOCALS` is descriptive of this, allocating space for a number of local variables. For speed, the values on the stack should not be initialized; just decrement the stack pointer `2*n`. `LOCALS` complements another Word I have found necessary in Forth, `DROPS`, to drop large blocks from the stack; the two functions could actually be combined.

Accessing Frame Elements

Bartholdi's Words `PAR0,PAR1,PAR2` were mentioned above as terms to access values on the stack, but Bowhill[3] has defined a different system of names based on operator symbols, including `@1`, `!1`, `@2`, `!2`, etc.. Both these nomenclatures were introduced with different stack frame systems than the one described here but can be adopted with new definitions. All these names share an important trait; they return the value of the designated cell instead of its address, in contrast to ordinary Forth variables. This accomplishes two objectives. First, it increases efficiency by eliminating `@` and `!` references from code. Second, it is necessary with some processors such as the Intel 8086, where a separate 64K address space can be assigned to the stack; addresses within this space cannot be operated on by the ordinary `@` and `!` functions.

With Bartholdi's names returning values, a prefix operator, `TO`, which he defined in another paper[1], is needed to store into parameters[9]. The expressions `TO PARi` and `+TO PARi` may represent storing into and incrementing parameter `i`. `TO` is a general purpose operator independent of data type, e.g. it can also store into a double width parameter `2PARi`. The `TO` construct has not been universally adopted, even for variables. It may be avoided with the alternative nomenclature introduced by S.A. Bowhill[3].

In Bowhill's approach operator names such as @ ! +! are converted into parameter references by appending a cell index suffix, as in @1 @2 etc. indicating the usual operation but on the specified frame element. These names are short and reveal what operation is done. Their hieroglyphic quality avoids confusion with any application variable names which also happen to have suffixes, and may alert the reader that a fundamental or built-in facility is meant. Their pronunciation could be as simple as "fetch one" for @1, with the full meaning clear from the context. Their only problem may be a difficulty in thinking of operator symbols as representing values, giving an unnatural feel to the code. One could always add a prefix P to overcome this, if you did not mind the extra length. Eliminating the split between TO and its following object, these operator based Words are easier to implement with code giving the fastest execution, based on the discussion of TO below.

An important point in defining a set of built-in local variable names is the need for names for more than one data type. To make full use of common parameters the system should provide double width along with single width. Incrementing should also be provided along with fetch and store. As an initial working set, I suggest names be given to the eight cells above and below the BP mark in the stack frame. PAR1,PAR2,... can be made a little more convenient by shortening to P1,P2, etc. In Bowhill's nomenclature the operators 2@ 2! +! and D+! all need associated parameter names for frame offsets from −8 to +8. That means ninety-six Words to be added to the Forth dictionary; assuming somewhat optimistically sixteen bytes per Word, at least 1.5 K bytes of memory would be required unless some special techniques were employed.

Wildcard Names

With so many parameter names to be added to the Forth dictionary much space would be saved if only the root name existed in the dictionary, with the numeric suffixes interpreted separately. It is really essential for readability, however, that the suffixes be written as part of the names, not separated by a space, as would be required for the normal Forth interpreter to separate the parts of the names. The confusing blanks can be eliminated by expanding the string matching function used when searching the dictionary to include wildcard matching as used in Utrecht and described by Joosten[9].

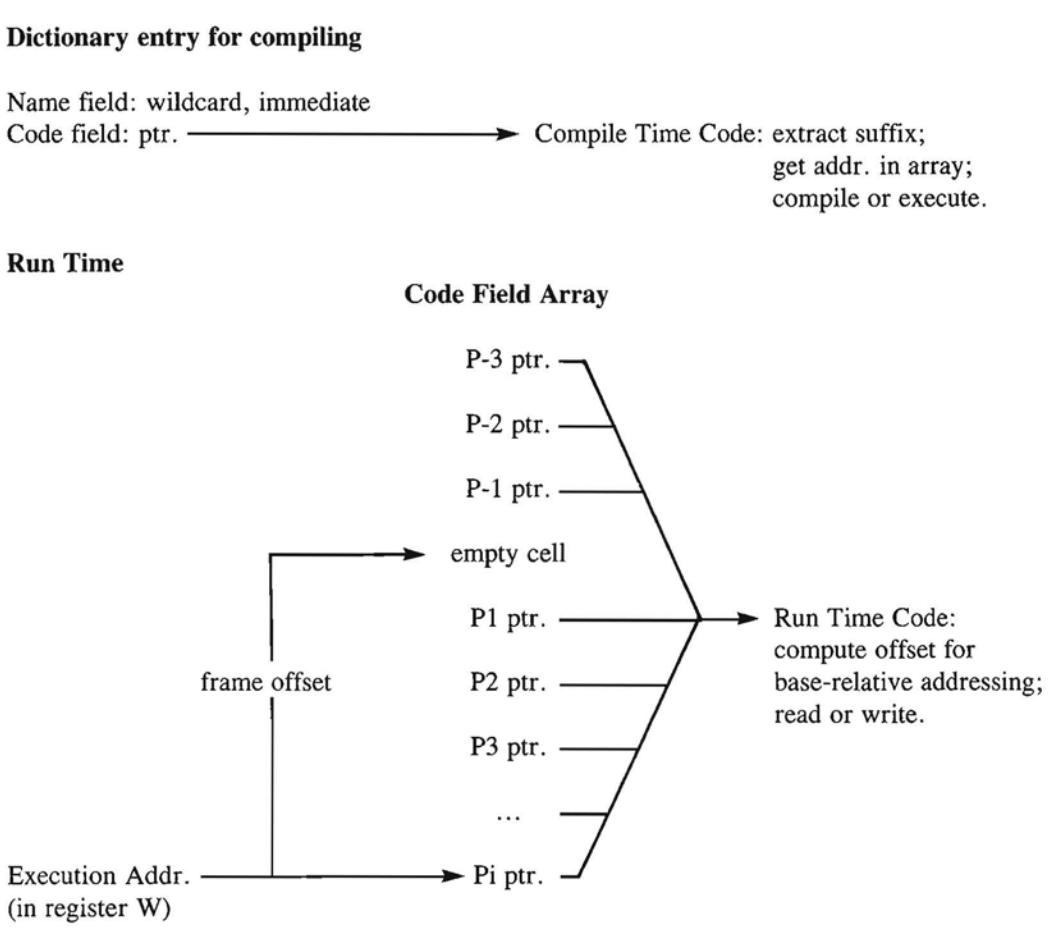
The details are dependent on the particular Forth system, but essentially names are flagged in the dictionary to indicate they take indeterminate, or wildcard suffixes. The search routine finds a match when the initial characters of a search string match, regardless of suffix characters past the length of the dictionary entry. The Words found must then be immediate and process the suffix themselves at compile time. For Bowhill's names the operators @ ! +! etc. would appear in the dictionary twice, in their existing versions and in a second wildcard version which represent stack frame elements. The regular versions must appear first in the search order, so that when no suffix is written the regular versions are found instead of the wildcards. Similarly any names like @X or !Y defined in the application will not be confused with the parameter names.

Code Pointer Arrays

There is a range of alternatives for what wildcard parameter references will compile, with different combinations of execution speed, compiled code size, and kernel support code size. The compiled code must execute quickly if local variables are to be generally useful; machine code must be used throughout and techniques to reduce the size of code should be carefully considered. In the earliest system Bartholdi compiled every reference as an in-line literal, i.e. as an execution address of machine code, followed by the frame offset for a parameter. The two cells comprising this in-line literal could be reduced to one cell if a unique execution address were used for each parameter. Normally unique execution addresses would involve entire dictionary entries for each parameter, but a method exists where only a code pointer for each parameter, addressing a single code segment shared by all the parameters, is required.

Referring to Figure 2, headerless code pointers may be arranged in an array forming a map of the stack frame around the BP point, with the offset within the array being the same as the offset of the parameter from BP. At run time the common code segment can compute the offset by subtracting the fixed base of the array in memory from the pseudo-machine register "W", set by the address interpreter to the address of the current code pointer in the array when executing that compiled address. When computing the offset is especially difficult, however, as on eight-bit processors, the fastest execution requires a separate code segment for each pointer, with the offsets imbedded in the code as immediate machine instruction operands[3].

Figure 2. A Code Field Array



Using TO with Local Variables

The original version of `TO`[1] sets a flag interpreted at run time by the Word compiled after `TO`, for selecting among fetching, storing, or incrementing. In that case only two arrays of code pointers described above are needed, one for single and one for double width parameters. The different operations on each parameter type would be executed by branches within the common code segment for each data type, addressed by all the elements in the array for each type. Execution would branch, according to the value of the flag, to separate routines for fetching, storing, or incrementing the parameter, after computing the offset. The kernel overhead for this version of `TO` is small but it adds

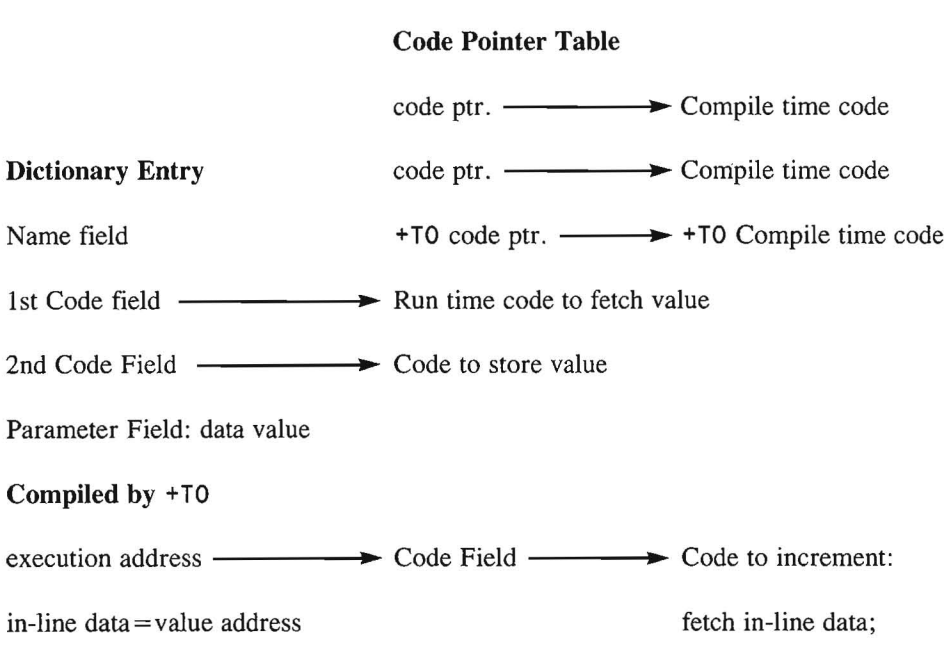
run time work for testing and clearing the flag, and takes up space in the application by compiling TO. Other, more efficient, versions of TO have therefore been sought[13,14].

Some improvement in TO is obtained by using multiple code pointers in each Word, one for fetching, one for storing, one for incrementing, and so on. TO and +TO must then be "state-smart", with code to both compile and execute the operation for use in the compiling or interpreting system state. When compiling, TO and +TO select which execution address to compile for the Word following in the source text. Executing compiled code is then faster since no TO ! or @ is executed at run time. Compiled code is also shorter because TO itself is not compiled. Dictionary headers, however, are lengthened by the extra code pointers. A single use of the extra pointer makes up for this extra length by saving the space otherwise used to compile TO or +TO etc.

Only fetching and storing, though, can be certain to be used with a variable, so the extra code pointers, for incrementing and other functions, are just wasted space. One way to reduce that space is to define only one extra function, one which returns the parameter field address; other functions can then take the form of ordinary post-fix operators taking an address on the stack. Even this address function, however, still might not be used. A more serious impact of multiple code fields occurs when using the code pointer arrays introduced above; an entire array is required for each of the code fields. I would therefore propose a new, hybrid version of TO. Variables or other Words usable with TO should have exactly two code pointers, one for fetching and one for storing; for all other functions such as +TO, compile an in-line literal, i.e. an operator followed by data. For ordinary variables the data would be the parameter field address, and for stack frame elements the data would be the frame offset.

The operators to be compiled by +TO etc. will be different for each data type, even though compiled by the same Word. A compact way to tabulate the different operators compiled is to use the code pointer already needed to define run time action, to also perform a compile time function. As in Figure 3, the code segment addressed by the first code pointer may be preceded by a table of additional execution addresses. This table is accessed at compile time by prefix operators such as TO, through the first code pointer addressing the end of the table. In the table are addresses for

Figure 3. Structure of Value Words



compile time routines unique to the data type, which in turn will compile associated run time operators followed by data; the data will be computed differently for each type. The table exists only once for all instances of a data type rather than burdening every header with duplicates of this information. The work needed to access the table is done only at compile time.

When applied to a local variable, wildcard name, the storing and fetching code segments above are modified to become immediate, compile time routines instead of the actual run time segments. `TO` therefore must test not only the system state but also whether the following Word is immediate, and execute any compiling procedure included in its definition. These compile time procedures interpret the numeric suffix and compile the corresponding element in the arrays of run time code pointers described above. Separate arrays are needed for fetching and storing, for each data type. With double and single width local variables covering the plus and minus eight cell range around BP, four arrays are needed occupying one hundred thirty-six bytes.

The discussion of the hybrid version of `TO` above has focused on compiled code. There is an additional overhead in this approach avoided by the original simple flag based method, which is code to execute the indicated operations when the system is interpreting. The execution addresses compiled by `+TO` and other special functions cannot be simply passed to `EXECUTE`, because they look for in-line literal data. Rather than include entire variants of such in-line operators for interpreter mode execution, it may be easier always to compile the operators and then execute what was compiled if the system is interpreting, simultaneously restoring the dictionary pointer. This technique is applicable to almost anything in Forth, but can cause problems when applied to large expressions like entire definitions. Its use with one Word at a time, however, should be straightforward.

References

[1] Bartholdi, Paul. "The TO Solution." *Forth Dimensions*, Vol 1, No.4:38-40. San Jose, CA: Forth Interest Group, 1979.

[2] _____. "Another Aid for Stack Manipulation and Parameter Passing in Forth." September, 1977. Reprinted in *1982 Rochester Forth Conference on Data Bases and Process Control*, Rochester, NY: Institute For Applied Forth Research, Inc., 1982.

[3] Bowhill, S. A. "Fast Local Variables For Forth." *1982 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1983.

[4] Duff, Charles, and Iverson, Norman. "Forth Meets Smalltalk." *Journal of Forth Application and Research*, Vol. 2, No. 3:7-26. Rochester, NY: Institute For Applied Forth Research, Inc., 1984.

[5] *Forth-83 Standard*, Mountain View, CA: Forth Standards Team, 1984.

[6] Glass, Harvey. "Towards a More Writable Forth Syntax." *1983 Rochester Forth Applications Conference*, Rochester, NY: Institute For Applied Forth Research, Inc., 1983.

[7] Greene, Ronald. "A Proposal for Implementing Local Words in Forth." *Journal of Forth Application and Research*, Vol. 2, No. 4:39-42. Rochester, NY: Institute For Applied Forth Research, Inc., 1984.

[8] Jekel, R. N. "Local Variables For Forth." *1980 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1981.

[9] Joosten, Rieks. "Techniques Working Group Report." *1982 Rochester Forth Conference on Data Bases and Process Control*, Rochester, NY: Institute For Applied Forth Research, Inc., 1982.

[10] Korteweg, Siem, and Nieuwenhuyzen, Hans. "Stack Usage and Parameter Passing." *Journal of Forth Application and Research*, Vol. 2, No. 3:27-50. Rochester, NY: Institute For Applied Forth Research, Inc., 1984.

[11] LaQuey, Robert. "Local Variables." *1984 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1985.

[12] Levy, George. "Arrays and Stack Variables." *1984 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1985.

[13] Lyons, George. "Alternatives to TO." *1982 Rochester Forth Conference on Data Bases and Process Control*, Rochester, NY: Institute For Applied Forth Research, Inc., 1982.

[14] Schleisiek, Klaus. "Multiple Code Field Data Types and Prefix Operators." *Journal of Forth Application and Research*, Vol. 1, No. 2:55-64. Rochester, NY: Institute For Applied Forth Research, Inc., 1983.

[15] _____. "Error Trapping and Local Variables—One Year Later." *1984 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1985.

[16] Shaw, George. "Rock & Roll Programming: An Innovative Approach to Local Variables." *1984 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1985.

[17] Volk, William. "Named Local Variables in Forth." *1984 FORML Conference Proceedings*, San Jose, CA: Forth Interest Group, 1985.

Manuscript Received February 1985.

Mr. Lyons received a B.A. in economics and mathematics at the College of Wooster, and has done graduate study in economics at Columbia University. Since 1970 he has served as a business economist at Bankers Trust Company in New York, one of the nation's largest banks, where he has been involved in the application of computers to economic research and forecasting.