Readable and Efficient Parameter Access Via Argument Records

Bill Stoddart

Department of Computer Science
Teeside Polytechnic
Middlesborough, Cleveland TS1 3BA
United Kingdom

Abstract

An Argument Record allows the input parameters and local variables of a high level Forth definition to be accessed by name. Method selection prefixes such as `to` then become widely applicable, allowing closer action object binding and reducing the proliferation of dictionary words as new data types such as floating point numbers and matrices are introduced. Such enhancements are of particular interest because they can be freely intermixed with classical Forth. Standard Forth techniques for describing compiler extensions prove to be an ideal tool for implementing these enhancements. A package that requires just 1200 bytes of dictionary space is sufficient to provide a wide range of facilities, and produces efficient re-entrant code.

Introduction

This paper stems from work done in conjunction with Universal Machine Intelligence, Ltd. on the use of 83 Standard Forth as a basis for a robotics control language. The envisaged applications include operations on information expressing the robot's "world view" in terms of the position and orientation of physical objects within its domain.

Classical Forth has a mixture of closely related strengths and weaknesses when tackling complex problems of this nature. The pressure to produce short readable words often results in a superior decomposition of a problem and a better interface between components than would result from Pascal. On the other hand, some well understood operations, such as matrix multiplication, are quite tricky to express in Forth, and when coded bear little resemblance to a text book description of the underlying algorithm.

Recent published work on enhancing Forth's power of expression [DUFF 84], [KORT 84] has described techniques that are of particular interest because they are of general applicability and may be freely intermixed with classical Forth. These techniques include formalised parameter passing with named arguments, and the use of a prefix syntax based on method selectors. [DUFF84] also considers the inheritance of class characteristics, a topic that is not attempted here.

In this paper I hope to show that the classical techniques for extending the Forth compiler, and the underlying architecture of the Forth system, are effective tools for providing a wide range of general enhancements. With a package that generates just 1200 bytes of compiled code when implemented on our 8086 Forth system, the features provided include: formalised parameter passing with argument lists, call by reference and call by value, a prefix syntax that can hide "type" details and which is applicable to both dictionary objects and named parameters, local variables, the dynamic allocation of storage for local arrays, and named loop indexes.

The implementation techniques are such that when used appropriately, these enhancements will generate more efficient solutions than classical Forth.

General Approach

We extend the Forth compiler in the usual manner using a combination of new defining words and immediate words. The normal syntax of words separated by spaces is maintained throughout. We use a naming convention by which immediate words defined in the package are named in lower case, whilst normal Forth operators are in upper case.

First Example: An Argument List With CONST Parameters.

Consider the following definition of `WITHIN`

```
: WITHIN ( n1 n2 n3 -- flag   true if n2 <= n1 < n3 )
   { const n1  const n2  const n3 }
n1 n3 <   n1 n2 < NOT   AND  ;
```

In this definition `{` commences the description of an "argument list". The defining word `const` is used to create the temporary dictionary entries `n1` `n2` and `n3`. The "argument list" is terminated by `}`, which will compile a run time operator to move three values from the stack into the "argument record", at the same time saving overwritten values of the argument record on the return stack. The "argument record" is an area of memory at a fixed offset from the user area pointer. See fig 1.

When `n1` is encountered during compilation of the remainder of the definition, it compiles a run time operator that will return its value to the stack. That is, it compiles a run time operator that will return the value in the argument record cell associated with `n1`. This run time operator is extremely efficient, consisting of only 6 bytes of machine code (including `NEXT`).

When compilation reaches `;`, the temporary words `n1` `n2` and `n3` are removed from the dictionary, and a run time operator is compiled that will restore the argument record to its condition on entry to the definition, and perform the `EXIT` function.

The argument list is not used to return results! These are just left on the stack in the usual Forth manner.

VAR Parameters and Access Method Selection

Consider the definition:

```
: +! ( n addr --  add n to contents of addr )
   { const n  var x }
   val x   n +   to x  ;
```

A parameter defined by `var` differs from `const` in that it is associated with additional access method selection prefixes `val` and `to`. These immediate operators set values in an action key, and cause `x` to compile an appropriate postfix operator. In fact the above definition will generate exactly the same code as:

```
: +!
   { const n   var x }   x @   n +   x !  ;
```

NUM Parameters, A Local Variable Facility

A `num` parameter has the same access methods as a `var` but is matched against a value on the stack rather than an address. Assigning a value to a `num` parameter within a definition places that value in the associated cell of the argument record. Such assignments are "local" to the definition

Figure 1. Argument record setup for WITHIN.

Before:

	parameter stack		argument record		return stack
		UP–>	status		
	x		a0	top–>	ret addr
	n1		a1		
	n2		a2		
top–>	n3		a3		
			a4		
			a5		
			a6		
			a7		

After:

	parameter stack		argument record		return stack
		UP–>	status		
top–>	x		a0		ret addr
			a1		a7
			a2		a6
			a3	top–>	a5
			a4		
			n1		
			n2		
			n3		

Key:

n1 n2 n3:	stack parameters for WITHIN
a0 .. a7	contents of the argument record on entry
UP	user area base pointer, the argument record for a task is immediately below the user area
status	contents of user variable 0. STATUS returns the address of this user variable, and would be used to define ARG. (See CODE AND INTERRUPT ROUTINE MANIPULATION OF ARGUMENT RECORDS).

in which they are made. A typical use is shown in the following definition of the greatest common divisor function:

```
: 1ST-GCD ( n1 n2 -- n3  n3 is the gcd of n1 and n2 )
   { num x  num y }
   BEGIN   val y WHILE
     val x  val y MOD  ( leave remainder on stack )
      val y  to  x
     to y  ( assign remainder to y  )
   REPEAT   val x  ;
```

This is used as:

```
16 24 1ST-GCD . 8 ok
```

Additional local variables may be declared that are purely internal to the definition. For example, we would could define the gcd function as:

```
: 2ND-GCD ( n1 n2 -- n3   functionally equivalent to 1ST-GCD )
   { num x  num y   0 num remainder }
   BEGIN  val y  WHILE
     val x  val y  MOD   to remainder
     val y  to  x
     remainder to y
   REPEAT   val x  ;
```

In this example, the 0 in the argument list is compiled as part of the definition in the normal way. When the run time operator compiled by } moves stack values into the argument record, 0 becomes the initial value of remainder. It follows that internal parameters such as remainder must follow parameters that are initialised from stack values provided externally.

External Argument Lists

The argument lists described above are internal to a high level Forth definition. This causes the definition to assign stack parameters to an argument record when executed. Where an argument list is declared externally, a following definition can access the argument record, but does not assign parameters to it. This allows the definition of words that operate on an argument record set up at a higher level of nesting.

Consider the following example in which $CLOSE (addr1 addr2 n--flag) returns true if character strings of length n at addr1 and addr2 are found to match in at least 75% of character positions.

```
{ num aptr  num bptr  const n  num m }
: NEXT-CHARACTER   1 aptr +!   1 bptr +!  ;
: $CLOSE ( addr1 addr2 n -- flag )
   DUP { const a$point  const b$point  const n  num mismatched }
   n 0 DO
     a$point C@   b$point C@   =  mismatched +!
     NEXT-CHARACTER
   LOOP   n   val mismatched   /   3 >  ;
```

NEXT-CHARACTER manipulates the first two parameters in the argument record set up by $CLOSE. The external argument list prior to NEXT-CHARACTER must match the argument list in $CLOSE in terms of the number of argument record cells allocated. The two lists do not need to match in terms of parameter types. In this example the access methods $CLOSE requires for a$point and b$point allow these to be const parameters, whereas the pointer incrementing function performed in NEXT-CHAR accesses the same parameters (the same cells in the argument record) as num parameters.

Code and Interrupt Routine Manipulation of Argument Records

External argument lists allow the assembly of code that manipulates argument records. A macro ARG, can be defined to assist this process, and is similar to the macro many Forth assemblers provide to access USER variables. In code, the definition of NEXT-CHARACTER as described in the previous section would then be:

```
{ num aptr   num bptr   const n  num m }
CODE  NEXT-CHARACTER
   aptr ARG INC   bptr ARG INC   NEXT   END-CODE
```

Use of the argument record in interrupt driven applications is also possible. For example, consider

an interrupt driven version of TYPE that outputs to a serial port. TYPE could set up two argument record cells as a pointer and a counter, initialise the transmitter, and execute STOP. The transmitter would then issue its "transmitter buffer empty" interrupt, and an associated interrupt routine would access the argument record, place the next character to be transmitted in the transmitter buffer, and decrement the counter.

The interrupt routine would be invoked each time the transmitter buffer became empty. When decrementing the counter produced a value of zero, the interrupt routine would disable the transmitter and wake the associated task, which would continue execution at the word following STOP.

Interactive Testing

Interactive testing is an essential part of Forth. It's not just a debugging tool. It's an effective way of relating to the concepts being developed in the solution to a problem.

With the exception of named loop indexes, all the facilities described in this paper can be used in both compilation and execution states.

The special argument list terminator }X can be used to set up an argument record in interpret mode. For example:

```
20 30 { num x num y }X
val x val y + . 50 ok
```

When }X moves values to the argument record the overwritten values in the record are not saved on the return stack, as this would interfere with the operation of the text interpreter.

Recursion

Recursion is supported in the usual way, by using RECURSE to compile a re-entry to the current definition. A recursive definition of the factorial function would be:

```
: NFACT ( n1 -- n2   n2=n1! )
     { const n }
     n
     IF   n 1- RECURSE  n *   ELSE  1  THEN  ;
```

RECURSE compiles the code field of the most recent dictionary entry. The parameter n is not counted as a dictionary entry for this purpose. In implementation terms, { must save the information that tells the system its latest definition, and } must restore this information.

Defining a New Parameter Type

The listings include definitions of the parameter definers const var num dnum fvar fnum and index. In this section we discuss the definition of the matrix parameter definer mat in some detail.

Functional Specification

The defining word mat is used within an argument list in the form:

```
{ ... mat m ... }
```

to define a matrix parameter m.

When the corresponding argument record is set up, m will be matched against the address of a 64 byte area of memory used to hold the elements of a 4 x 4 matrix with 32 bit real elements.

Subsequent use of m (without prefixes) within the scope of the argument list will return the address of the matrix to the stack.

When used with prefixes the following are allowed:

`to m (addr-)` moves the 64 bytes at addr to `m`.

`of m (i j-addr )` returns the address of matrix element `i j`.

`val of m (i j- -f- x )` returns the value of matrix element `i j` to the floating point stack.

`to of m (i j- x -f- )` stores the top element of the floating point stack as matrix element `i j`.

Definition

Each cell in the argument record is supported by two run time operators which return its address or its 16 bit contents respectively. These are the "cell address returner" and the "cell contents returner".

In this case the argument cell will contain the address of a matrix. All matrix operations consist of placing this matrix address on the stack with the "cell contents returner" and optionally selecting a subsequent operation via method selection prefixes.

The subsequent operation will be an "access method" for the matrix. These access methods are defined as normal Forth words, in code or high level Forth as required. They can be given temporary headers because they will be accessed from an "action table" (see below). Corresponding to the actions defined in the functional specification, we define access methods:

`MCOPY addr1 addr2 --`
Copy 64 bytes from addr1 to addr2. The matrix action table (see below) will associate this method with the prefix `to`.

`MADDR-ELEMENT i j addr --`
Return the address of element `i j` of the matrix at `addr`. The matrix action table will associate this method with the prefix `of`.

`M@-ELEMENT i j addr -- ; -f- x`
Return the value of element `i j` of the matrix at `addr` to the floating point stack. The matrix action table will associate this method with the prefix combination `val of`.

`M!-ELEMENT i j addr -- ; x -f-`
Store the top value from the floating point stack in element `i j` of the matrix at `addr`. The matrix action table will associate this method with the prefix combination `to of`.

We now compile the action table that matches the allowable method selectors with the given methods. Note that prefixes such as `to` and `of` set bits in a variable known as the `ACTION-KEY`. The word `ACTION` returns the contents of this variable and re-initialises the variable to 0. The value -1 is used to terminate the table.

```
CREATE MATRIX-ACTIONS
     to     ACTION , ] MCOPY [
     of     ACTION , ] MADDR-ELEMENT [
     val of ACTION , ] M@-ELEMENT [
     to of  ACTION , ] M!-ELEMENT [
    -1 ,
```

Finally, we are ready to define `mat` as:

```
: mat MATRIX-ACTIONS CCR ARG ; IMMEDIATE
```

Here `CCR` returns the address of an area of memory that contains the headless code fragments that are the cell content returners for each cell in the argument record. The `CREATE .. DOES>` action

of mat is encapsulated within ARG. The CREATE phase constructs a dictionary entry (for m say) in the "argument names area", which is a section of memory above the main dictionary. The parameter field of this entry will contain its position in the argument list, and the addresses provided by MATRIX-ACTIONS and CCR.

The DOES> phase executes immediately when the defined parameter m is subsequently encountered within the definition. It determines the argument record cell associated with m and compiles the associated cell content returner. It then checks whether any method selectors have preceeded m, and if so matches the specified ACTION key against the action fields in the MATRIX-ACTIONS table, and compiles the associated run time operator.

The Scope of an Argument List

The scope of an argument list is that part of the input stream during which its parameter names remain in the dictionary. For an internal argument list, this is until the end of the definition containing the argument list. For an external argument list, this is until the start of the next argument list.

Use of Method Selectors With Other Defining Words

When a set of run time operators and an action table such as MATRIX-ACTIONS has been defined, it is a simple matter to apply the appropriate method selectors to a define an intelligent data object in the dictionary.

Usage

Suppose we define:

```
: MATRIX  CREATE  64 ALLOT  ;
```

Then the phrase:

```
MATRIX-ACTIONS PREFIXED MATRIX AMAT
```

will define an intelligent matrix AMAT that is used with the same syntax as the parameter mat m. For example we can say:

```
12.7 E 4  to  4 3  of  AMAT
```

Use of AMAT without prefixes returns the base address of the matrix, giving compatability with a plain MATRIX.

Implementation

PREFIXED creates two new words. The first is a dumb headless matrix, equivalent to | MATRIX AMAT. The second is an immediate word AMAT whose action is to compile or execute the first word (depending on STATE), then check for a specified access method and execute or compile that. PREFIXED will work with any defining word. For example we can say:

```
16B-ACTIONS PREFIXED VARIABLE X
3 to X etc.
```

Such intelligent data objects require an additional 7 bytes of dictionary space (on our DTC system) compared with their dumb equivalents.

Dynamic Allocation of Space for Local Data Structures

The argument record provides a convenient area for holding 16 and 32 bit local variables. When

an equivalent facility is required for a larger amount of data, such as an array, we must assign this in a different area.

We provide this facility with the words: `internal`, `£CAPTURE` and `£RELEASE`. In the matrix multiplication example described below, these are used as follows:

```
: M* { ..... internal mat m4 ... }
     64 m4 £CAPTURE
     ...
     64 m4 £RELEASE  ...  ;
```

`internal` compiles a literal that is returned to the stack when `M*` is subsequently executed, and then becomes the initial content of the argument record cell associated with the following parameter `m4`. This value provides information from which the address of the argument record cell associated with the parameter may be calculated.

`£CAPTURE` is used within the definition to compile a run time operator that requires stack values `n1` and `n2`, `n1` being the number of bytes to be allocated, and `n2` being the literal value compiled by `internal`. In the above example `n1=64`, and the run time operator will allocate 64 bytes of memory space and place a pointer to the allocated area in the argument record cell associated with `m4`.

`£RELEASE` expects a stack value `n`, and releases `n` bytes of allocated memory.

In the present implementation, space is allocated by advancing the parameter stack pointer. Parameters currently on the stack are hidden until the space is released, and a balanced stack is required between `£CAPTURE` and `£RELEASE`. This provides the most convenient option in terms of memory management.

Named Loop Indexes

Description

Loop index parameters allow a named index to be associated with a particular loop. For example:

```
: TEST
     { index i }
     5 0 with i DO  i .  LOOP ;
    TEST 0 1 2 3 4 ok
```

The prefix `with` is used to associate index `i` with the following loop. It compiles a run time operator that will store a pointer to the top of the return stack in the argument record cell associated with `i`. Use of `i` within the loop will return the index of the loop associated with `i`.

Loop indexes differ from all other parameters considered here in two ways.

Firstly, the default action of a loop index is not to return either the address of or the contents of a cell in the argument record. In fact the use of an index without a prefix will compile a "cell address returner" followed by another operator that evaluates the loop index. Compilation of the second run time operator is arranged by including the hidden prefix `DEFAULT` in the defining word `index`.

Secondly, an index will ignore a preceeding `to` and leave the `to` flag in the `ACTION-KEY` set. This allows the use of a phrase such as:

```
i j val of amatrix to j i of bmatrix
```

Thus `to` is allowed to ride over `i j` and apply to `bmatrix`.

Advantages

Named loop indexes are more convenient than the Forth words `I` `J` and `K`. For example, in the matrix multiplication:

```
C = A * B
```

we can write the elements of `C` as

$$Cij = \sum_{k=1}^{n} Aik * Bkj \qquad (1)$$

To code this using the FORTH operators `I` `J` and `K`, we must use `I` as the inner loop index, rather than `K` as suggested in (1). As a first step we might rewrite the formula as

$$Ckj = \sum_{i=1}^{n} Aki * Bij$$

This gets the `i` `j` and `k` of our formula a little closer to the `I` `J` and `K` of Forth, but a second problem arises when we assign the product. By the time the assignment takes place we have left the inner loop and all loop indexes have changed their names!

Matrix Multiplication Example

The word `M*` defined below performs a floating point multiplication of `4*4` matrices `m1` and `m2`, and places the result in matrix `m3`.

`4*4` matrices are the most general tool available for the description of three dimensional space, and are extensively used in the control of robot manipulators [PAUL81].

```
: M* { mat m1  mat m2  mat m3   internal mat m4
       index i  index j  index k }
    64 m4 £CAPTURE
      5 1 with i DO
        5 1 with j DO   FØ.
          5 1 with k DO
            i k val of m1   k j val of m2   F*   F+
          LOOP   to i j  of  m4
        LOOP
      LOOP
      m4 to m3
    64 #RELEASE  ;
```

The definition has been coded for the 8087 numeric coprocessor. The floating point operations `F*` and `F+` take their arguments from the floating point stack, and return results to the floating point stack. `FØ.` places zero on the floating point stack. An interesting feature of the definition is that the same source code could apply to a system that held floating point values on the parameter stack. This transparency does not occur because the 8087 is hidden from the programmer, but because of the extreme simplicity of stack usage obtained with named parameters.

The results of the calculations are initially assigned to `m4`, and only copied to `m3` when the matrix multiplication is complete. This allows the destination `m3` to be the same matrix as `m1` or `m2`, so that we can define operations such as:

```
: MATSQUARE ( m --  m*m to m )
     DUP DUP M*  ;
```

The Advantages of Prefix Syntax and Access Method Selection

Now that a fairly complete example has been described, we have the material to review the reasons for including operaton prefixes in this package.

Reduced Number of Words

Although 4 new run time operators are defined to access a matrix, the only new word the user requires to invoke them is the parameter defining word `mat`.

Action Object Binding and Data Type Hiding

Prefixed data objects select an access method for their data. This gives us a greater confidence that the access method will be the appropriate one, and reduces the amount of source code modification when an algorithm is recoded for a different data type.

As an example, if `X` is a `2VARIABLE` a programmer might erroneously attempt to return its contents with `X @` rather than `X 2@`.

If a prefix syntax is available, the programmer can request the value of `X` with the phrase `val X`. This will be the phrase used whether `X` is a 16 bit integer, a 32 bit integer, a 32 bit real, a 64 bit real, a complex number, or a data record describing an antique Chinese vase.

Data Structures in Extended Memory Space

In a system with a memory that extends beyond Forth's Standard 64k bytes, it becomes easy to hide the difference between a data structure held in Forth's memory, and one held in extended memory. The difference is in the access methods, and can be hidden behind the definition of defining words. For example, if all matrices were held in extended memory rather than Forth memory, the access methods `MCOPY` etc. would be specified differently, but from then onwards all source code could be compatible with a system that held its matrices in the Forth memory space.

Efficiency of Compiled Code

Memory Requirements

Including an argument list in a definition generates an additional 4 bytes of compiled code. The run time operator `(SETUP)` which moves data into the argument record requires a cell count as a 1 byte in line value (3 bytes total). The run time operator `(RESTORE)` which restores the argument record and performs the `EXIT` function also requires 3 bytes. Two bytes are saved by having the `EXIT` function performed by `(RESTORE)`.

To access a parameter without method selection prefixes requires 2 bytes of compiled code. To access a parameter with method selection prefixes requires 4 bytes.

To add a new parameter type the overhead of an action table must be incurred, in addition to the definition of the access methods themselves. However, this overhead is more than offset by using headless words for the access methods. Once access methods and an action table are set up, they may be used in combination with existing defining words to create intelligent data objects in the dictionary. The additional overhead for intelligence is 7 bytes per object. This is the best place to incur the overhead, because a dumb object which is only able to return its parameter field address may be manipulated by reference from within compiled code, and this makes intelligent objects something of a luxury.

Speed of Execution

Argument records cannot compete with Forth for simplicity of parameter passing, but once the execution overhead associated with setting up the argument record has been paid, access to parameters is very efficient. This is assured in the following ways:

- by dedicating two separate run time operators to each cell in the argument record, (the cell address returner and the cell contents returner);
- by the ability to write access methods in code;
- and by an ability to access the argument record from a code definition when optimising the time critical section of a routine (as when re-coding an inner loop).

Further Developments

(1) A way should be found to indicate that an operator in an action table has the "immediate" attribute. This would provide a facility that is already familiar through use of immediacy in Standard Forth. An important application that can be forseen at present is the support of method selection prefixes during target compilation. Named loop indexes could also be implemented more efficiently.

The normal method of flagging immediacy by setting a bit in the length byte of the words name field is not available, since the action table operators are already compiled into the action table.

Instead we could define:

```
15 PREFIX IMMED
```

`IMMED` would be used to flag certain entries when defining an action table.

`MATCH` would compare key values on the lower 15 bits only, and on finding a match would inspect bit 15 of the action table key value, and return an immediate flag in addition to the operator's cfa.

(2) Modified versions of `DOES>` and `;CODE` are required to allow argument lists to be used within the `CREATE` phase of defining words.

Conclusions

With a package that produces 1200 bytes of compiled code it has been possible to add the following enhancements to Forth:

- Argument lists and named parameters
- facilities equivalent to call by reference and call by value
- local variables, and the dynamic allocation of memory for local arrays, etc.
- named loop indexes
- a prefix syntax that invokes a method selection mechanism, and which can be used with both argument list parameters and classical Forth defining words.

Tools are provided for extending these facilities by adding new prefixes and parameter types.

The new facilities integrate perfectly with classical Forth. Given the functional specification (glossary entry) for a word, the word may be coded with the help of the facilities described or without them. It will appear the same to the rest of the system. In addition the facilities allow fully interactive debugging.

[DUFF84] and [KORT84] have argued convincingly that methods similar to those described here enhance the readability and modifiability of algorithms coded in Forth, and in his introduction to the "Enhancing Forth" issue of The Journal [V2.3], Lawrence Forsley writes: "It is an exercise for the reader to determine whether these concepts enhance Forth or supercede it". In writing this package, I have been surprised how effectively Forth can describe the enhancements and integrate them into its repertoire. This has reinforced my feeling that Forth has an essentially correct approach, and that formalised parameter passing and prefix syntax could rapidly be accepted as an integral part of the language.

As we discover the best way to write these enhancements, we will learn about Forth itself. One area this will cover is the optimisation and generalisation of words from which the enhancements are built. Some of these will be of great general utility. For example, this package uses a word `DELETE ( addr1 addr2--)` to delete all dictionary entries between `addr1` and `addr2`. This is

used when deleting parameter names from the dictionary at the end of a definition. It is also used in other packages that support separate headers and temporary compilation aids.

The use of state smart words will be another area for debate, as will the manipulation of dictionary entries for data objects that are implemented as immediate words that compile run time operators. What do we mean by the "body" of such a word, for example?

References

[DUFF84] Charles Duff and Norman Iverson: "Forth Meets Smalltalk" *Journal of Forth Application and Research*, Vol 2, No. 3, 1984.

[KORT84] Siem Kortweg and Hans Nieuwenhuyzen: "Stack Usage and Parameter Passing" *Journal of Forth Application and Research.* Vol 2, No. 3, 1984.

[PAUL81] Richard P. Paul: *Robot Manipulators, Mathematics Programming and Control*, MIT Press. 1981.

Manuscript Received June 1985.

Bill Stoddart received a BSc in Mathematics and an MSc in Statistics from Sheffield University, U.K. He has worked as a Systems Engineer and Yoga teacher and is now a Senior Lecturer in the Dept. of Computer Science at Teeside Polytechnic, Cleveland, U.K. He has implemented a multi-tasking 83 Standard Forth which is used for teaching and project work in the Polytechnics Real Time Laboratory.

Appendix A: Terminology

Access method. An operation used to access data. For example `@` `!` and `+!` are access methods for 16 bit integer data.

Access method selection prefix. An immediate word such as `to` `val` `with` or `of` which is used before a named parameter or a prefixed data object to select the access method to be used.

Argument. A value which is passed as input to a Forth word via the stack, and which may be copied to the argument record and accessed by name if an argument list is present.

Argument list. A source code description of stack arguments which gives each argument a temporary name. When used within a high level definition, an argument list compiles a run time operator that moves arguments from the stack into an argument record.

Argument record. An area of memory at a fixed offset from the user area pointer. A multi tasking system has an argument record for each task.

Data object. Data plus the operations that access the data. Used in preference to "data structure" to avoid the connotation of a typed template associated with the use of the term "data structure" in Pascal. Nothing fancy is implied. A Forth constant is a data object within the terms of this definition. On the other hand a data object could be something more complex, such as a relational data base.

Named argument. A temporary dictionary entry created during interpretation or compilation of an argument list, and used to access an argument that has been copied to the argument record.

Parameter. Used synonymously with "argument".

Prefix. An abbreviated form of "access method selection prefix".

Appendix B: System Description

8086 Code Level Features

The system has a direct threaded NEXT, and thus the code field of each definition contains native code.

The following 8086 registers are dedicated to Forth:

SI Forth's threaded code instruction pointer
BX Forth's user area base pointer
SP Forth's parameter stack pointer
BP Forth's return stack pointer.

Indirection is indicated in the assembler by using @r, where r is an appropriate register name. For example:

```
-4 ) BX@ PUSH
```

will push the contents of the location 4 bytes below the address held in BX.

The assembler instruction LODS which is used to access in line arguments for (SETUP) and (RESTORE) is equivalent to:

```
SI@ AL MOV SI INC
```

The instruction CBW converts a byte in AL to a word in AX.

Extensions to the 83 Standard

| Causes the following definition to be created with a separate head. See SNIP

```
<>    n1 n2 -- flag
```

True if n1 not equal to n2.

```
BIT         n1 -- n2
```

n2=2^n1, for example 3 BIT . 8 ok

```
DELETE      addr1 addr2 --
```

Removes all dictionary entries whose link field addresses are between addr1 and addr2, where addr1 is below addr2.

```
S0    -- addr
```

A user variable containing the base address of the parameter stack.

SNIP Removes all separate headers from the dictionary.

Floating Point Extensions

The system uses an 8087 co-processor and floating point operations generally take values from, and return values to, the floating point stack. The notation -f- is used to show the pre and post condition of the floating point stack, as in:

```
F@ addr -- ; -f- x
```

Fetches floating point value x from addr, and leaves x on the floating point stack.

The remaining floating point operations used are:

```
F!    addr -- ; x -f-        stores x at addr.
F*    x1 x2 -f- x3           x3 = x1 * x2
F+    x1 x2 -f- x3           x3 = x1 + x2
F0.         -f- 0
```

```
SCR 16                                                    WJS 20MAY 85)
 0 ( ACTION PREFIX to of val with
 1
 2 | VARIABLE ACTION-KEY
 3
 4 | : NULL  0 ACTION-KEY ! ;  NULL
 5
 6 : ACTION   ACTION-KEY @   NULL ;
 7
 8 : PREFIX   CREATE C,   DOES>
 9    C@ BIT   ACTION-KEY @ OR   ACTION-KEY !  ;
10
11 0 PREFIX to   IMMEDIATE    1 PREFIX val IMMEDIATE
12 2 PREFIX of   IMMEDIATE  | 3 PREFIX DEFAULT
13 4 PREFIX with IMMEDIATE
14
15

SCR 17                                                    WJS 13MAY85)
 0 ( MATCH
 1
 2 | : MATCH ( addr1 key -- addr2   addr2 is execution addr or 0
 3    SWAP OVER ( key addr key )
 4    IF
 5      BEGIN ( key addr )  DUP @   -1 = ABORT" action?"
 6        2DUP @   <> WHILE   4 +
 7      REPEAT   2+ @   SWAP DROP
 8    ELSE  DROP  THEN  ;
 9
10
11
12
13
14
15

SCR18                                                     WJS 20 MAY85)
 0 ( Argument list support
 1
 2 | 8 CONSTANT AREC-SIZE   | -16 CONSTANT AREC
 3 | VARIABLE DP''   | VARIABLE £ALLOCATED
 4 | VARIABLE AFLAG   0 AFLAG !
 5
 6 | : PNA   S0 @ 580 -   S0 @ 480 -  ;
 7
 8 | : CHECK ABORT" arg list?" ;
 9
10 | : ALLOCATE ( n -- )
11    £ALLOCATED @ +
12    DUP   0  AREC-SIZE 1+   WITHIN NOT   CHECK
13    £ALLOCATED ! ;
14
15
```

```
SCR 16
 0 Words defined with PREFIX set individual bits in ACTION-KEY.
 1
 2 ACTION returns the ACTION-KEY value, and re-initializes the
 3 ACTION-KEY to 0.
 4
 5 For example: to of ACTION . 5 ok
 6
 7
 8
 9
10
11
12
13
14
15

SCR17
 0 MATCH is used in the DOES> phase of ARG and in the DOES> phase
 1 of PREFIXED, to select an access method. For examples of the
 2 type of table searched by MATCH see I16-ACTIONS and
 3 MATRIX-ACTIONS.
 4
 5 addr1 is the address of an action table (such as I16-ACTIONS).
 6 key is an ACTION value.
 7
 8 If the ACTION value is 0, MATCH returns a value of 0. This is
 9 used to indicate that no additional action is to be executed or
10 compiled. (See ARG and PREFIXED).
11
12 If the key value is not matched by any entry in the table, an
13 error is reported.
14
15 Otherwise the cfa of the required access method is returned.

SCR 18
 0 AREC-SIZE returns the number of cells in the argument record.
 1
 2 AREC returns the offset of the argument record from the base of
 3 the user area.
 4
 5 AFLAG is set by } (in compilation mode) and tested by ;
 6
 7 DP'' is used to manage memory allocation in the PNA.
 8
 9 £ALLOCATED holds a count of the arg record cells allocated.
10
11 PNA returns the limits of the parameter names area, which is
12 the area used to hold the temporary dictionary entries required
13 for named parameters defined in an argument list.
14
15 ALLOCATE (n-) allocates n cells in the argument record.
```

```
SCR 19
 0 ASSEMBLER  ( Run time operators                          WJS 11MAY85)
 1 CREATE CAR ( cell address returners ) 7 C, ( code size)
 2    AREC ) BX@ DI LEA   DI PUSH   NEXT
 3    AREC 2+ ) BX@ DI LEA   DI PUSH   NEXT
 4    AREC 4 + ) BX@ DI LEA   DI PUSH   NEXT
 5    AREC 6 + ) BX@ DI LEA   DI PUSH   NEXT
 6    AREC 8 + ) BX@ DI LEA   DI PUSH   NEXT
 7    AREC 10 + ) BX@ DI LEA   DI PUSH   NEXT
 8    AREC 12 + ) BX@ DI LEA   DI PUSH   NEXT
 9    AREC 14 + ) BX@ DI LEA   DI PUSH   NEXT
10
11 CREATE CCR ( cell content returners ) 6 C,
12    AREC      ) BX@ PUSH   NEXT    AREC  2 + ) BX@ PUSH   NEXT
13    AREC  4 + ) BX@ PUSH   NEXT    AREC  6 + ) BX@ PUSH   NEXT
14    AREC  8 + ) BX@ PUSH   NEXT    AREC 10 + ) BX@ PUSH   NEXT
15    AREC 12 + ) BX@ PUSH   NEXT    AREC 14 + ) BX@ PUSH   NEXT

SCR20
 0 ( Run time operators                                     WJS 15MAY85)
 1
 2 | CODE (SETUP)
 3    LODS   CBW   AX CX MOV   AREC ) BX@ DI LEA
 4    BEGIN   DI@ AX MOV   BP DEC   BP DEC
 5      AX BP@ MOV   DI@ POP   DI INC   DI INC
 6    LOOPZ UNTIL   NEXT   END-CODE
 7
 8 | CODE (RESTORE)
 9    LODS   CBW   AX CX MOV   AX AX ADD   AX DI MOV
10    AREC ) BXDI+@ DI LEA   SP BP XCHG
11    BEGIN   DI DEC   DI DEC   DI@ POP   LOOPZ UNTIL
12    SI POP   SP BP XCHG   NEXT   END-CODE
13
14
15

SCR21
 0 ( Opening and closing an argument list,  new ;       WJS 20MAY85)
 1 | : {MK1   0 £ALLOCATED !    PNA OVER DP'' !   DELETE  ;
 2
 3 : {   {MK1   LAST 2@  ;  IMMEDIATE
 4
 5 : }   LAST 2!   STAGE @
 6    IF   COMPILE (SETUP)  £ALLOCATED @ C,   -1 AFLAG !   THEN ;
 7       IMMEDIATE
 8
 9 : }X   ?EXEC   2DROP  £ALLOCATED @ 2*
10    0DO  STATUS AREC +  I +   !   2 +LOOP   ;
11
12 : ;   AFLAG @
13    IF   0 AFLAG !   COMPILE (RESTORE)  £ALLOCATED @ C,   {MK1
14    ELSE   COMPILE EXIT   THEN
15    ?CSP   SMUDGE   [COMPILE] [  ;   IMMEDIATE
```

```
SCR19
 0 Each cell of an argument record has two associated run time
 1 operators that return its address and its contents respectively.
 2
 3 The code for these headless operators is provided in two tables,
 4 whose base addresses are returned by CAR and CCR. Each cell
 5 address returner is 7 bytes in length, and each cell content
 6 returner is 6 bytes. The arrangement given is specific to a DTC
 7 system.
 8
 9 Compiling an access method for a named parameter consists of
10 compiling either the cell address returner or the cell contents
11 returner for the argument record cell associated with the
12 parameter, followed by an optional second operator as determined
13 by ACTION.
14
15

SCR20
 0
 1 (SETUP) is the run time operator compiled by } to move stack
 2 parameters into the argument record. As new values are moved
 3 into the argument record, values that will be overwritten are
 4 saved on the return stack.
 5
 6 (RESTORE) is compiled by ; if an argument record is present in
 7 the definition. It restores the contents of the argument record
 8 to their state on entry to the definition, and performs the EXIT
 9 function
10
11
12
13
14
15

SCR21
 0 { marks the start of a new argument list by setting the number
 1 of allocated cells to zero, resetting DP'' to the start of
 2 the PNA, and deleting any parameter names currently in the
 3 dictionary. LAST is a 2 cell user variable giving the name and
 4 code fields of the latest dictionary entry.
 5
 6 } closes an argument list, restoring LAST. In compile mode
 7 (SETUP) is compiled, followed by an in line value giving the no.
 8 of allocated cells in the argument record. AFLAG is set to mark
 9 the presence of an argument record in the definition.
10
11 }X closes an argument record and initialises it with stack
12 values. This is useful during interactive testing.
13
14 ; is redefined to test AFLAG and conditionally compile (RESTORE)
15 and re-initialise the argument record description.
```

```
SCR 22
 0 ( ACT ARG-INTERPRET ARG DARG                              WJS 16MAY85)
 1
 2 | : ACT STATE @ IF , ELSE EXECUTE THEN ;
 3
 4 | : ARG-INTERPRET ( pfa )
 5      DUP C@ 1+ NEGATE   £ALLOCATED @ +  ( pfa cell-no )
 6      OVER 1+ @   DUP C@   ROT * + 1+ SWAP ( cfa1 pfa )
 7      3 + @  ACTION MATCH ( cfa1 cfa2 )  >R   ACT   R>
 8      ?DUP IF  ACT  THEN  ;
 9
10 : ARG
11    HERE  DP'' @   DP !   DP'' !
12    ICREATE IMMEDIATE   £ALLOCATED @ C,   1 ALLOCATE   , ,
13    HERE   DP'' @   DP !  DP'' !    DOES> ARG-INTERPRET  ;
14
15 : DARG   1 ALLOCATE   ARG  ;

SCR 23
 0 ( PREFIXED                                                WJS 20MAY 85)
 1
 2 : PREFIXED ( addr -- )
 3    >R   '   >IN @ >R   | EXECUTE   R> >IN !
 4    LAST 2+ @    CREATE IMMEDIATE   ,    R>   ,
 5    DOES>   DUP >R   @ ACT   R> 2+ @   ACTION MATCH   ?DUP
 6    IF   ACT   THEN ;   EXIT   Example usage follows
 7
 8 Suppose BASE is defined as:      40 USER BASE
 9 We define BASEX as a prefixed version of the same user
10 variable with:    40 I16-ACTIONS PREFIXED USER BASEX
11
12 We can then define:   : BINARY 2 to BASEX ;   etc.
13
14 A limitation of this technique is that ' BASEX >BODY  will not
15 return a pfa containing the same information as ' BASE >BODY.

SCR 24
 0 ( Some action tables                                      WJS 18MAY85)
 1
 2 | CREATE NO-PREFIXES -1 ,
 3
 4 CREATE I16-ACTIONS
 5     to ACTION ,  ] ! [
 6    val ACTION ,  ] @ [ -1 ,
 7
 8 CREATE I32-ACTIONS
 9     to ACTION ,  ] 2! [
10    val ACTION ,  ] 2@ [    EXIT
11
12 CREATE R32-ACTIONS
13     to ACTION ,  ] F! [
14    val ACTION ,  ] F@ [
15
```

```
SCR 22
 0 ARG is the base defining word for all named parameters within
 1 an argument list. The CREATE phase of ARG executes while a
 2 parameter list is being scanned. A dictionary entry for the
 3 parameter is built in the parameter names area, the number of
 4 the cell allocated for the parameter is compiled, along with 2
 5 additional values which are an action table address and the
 6 address of either the CAR or CCR table.
 7
 8 The DOES> phase of ARG is executed when the parameter name is
 9 subsequently encountered. This phase is state dependant. The
10 arg record cell associated with the parameter is determined,
11 (this is not known until the arg list is completed) and the
12 c.a.r or c.c.r for the cell either compiled or executed. The
13 action key value is then searched for in the action table of the
14 parameter, and the run time operator associated with the key
15 is either compiled or executed.

SCR 23
 0 The CREATE phase of PREFIXED expects the address of an action
 1 table to be on the stack. PREFIXED must be followed in the
 2 input stream by 2 words, say DDDD and CCCC. DDDD must be an
 3 existing defining word (such as USER in the source screen
 4 example). CCCC is the name of the new dictionary entry.
 5
 6 A headless word of type DDDD is created along with an immediate
 7 word CCCC. CCCC will compile or execute the headless operator
 8 depending on STATE, and will then attempt to match the key
 9 provided by ACTION against entries in the action table at addr.
10 If a key match is obtained, the second cfa is compiled or
11 executed depending on STATE.
12
13
14
15

SCR 24
 0 Each entry in an action table consists of a method selection key
 1 followed by the execution address of the operation associated
 2 with that key.
 3
 4 The action tables are used in the definition of named parameter
 5 types such as const num var fnum. These and others are on the
 6 following screen.
 7
 8 Action tables are also used in the definition of prefixed data
 9 structures. See previous screen.
10
11
12
13
14
15
```

```
SCR 25
 0 ( Some argument definers                                    WJS 20 MAY85)
 1
 2 : const   NO-PREFIXES   CCR   ARG   ;   IMMEDIATE
 3 : num     I16-ACTIONS   CAR   ARG   ;   IMMEDIATE
 4 : var     I16-ACTIONS   CCR   ARG   ;   IMMEDIATE
 5
 6 : dnum    I32-ACTIONS   CAR   DARG  ;   IMMEDIATE
 7
 8 EXIT
 9
10 : fnum    R32-ACTIONS   CAR   DARG  ;   IMMEDIATE
11 : fvar    R32-ACTIONS   CCR   ARG   ;   IMMEDIATE
12
13
14
15

SCR 26
 0 ( Dynamic memory allocation for 'internal' data     WJS 20MAY85)
 1
 2 CODE £RELEASE    AX POP    AX SP ADD    NEXT    END-CODE
 3
 4 | CODE (£CAPTUR) ( n1 n2 -- )  LODS    CBW    DI POP
 5    AX  DI SUB    DI NEG    BX DI ADD ( arg rec cell addr to DI )
 6  AX POP     AX SP SUB    SP DI@ MOV    NEXT    END-CODE
 7
 8 : £CAPTURE   COMPILE (£CAPTURE)    £ALLOCATED @ 1- 2* C, ;
 9 IMMEDIATE
10
11 : internal
12   £ALLOCATED @ 2*    AREC -    [COMPILE] LITERAL   ;    IMMEDIATE
13
14
15

SCR27
 0 ( Named loop indexes                                        WJS 20MAY85)
 1 | CODE (GET-INDEX)    DI POP    DI@ DI MOV    DI@ AX MOV
 2    2 ) DI@ AX SUB    AX PUSH    NEXT    END-CODE
 3
 4 | CODE (SET-INDEX)    DI POP    -4 ) BP@ AX LEA    AX DI@ MOV
 5    NEXT    END-CODE
 6
 7 | CREATE INDEX-ACTIONS
 8          DEFAULT ACTION , ] (GET-INDEX) [
 9    with DEFAULT ACTION , ] (SET-INDEX) [
10
11 : index   INDEX-ACTIONS CAR ARG    COMPILE Ø
12    DOES>   ACTION-KEY @ 1 AND    DUP >R    NEGATE ACTION-KEY +!
13     DEFAULT ARG-INTERPRET    R> ACTION-KEY !   ;    IMMEDIATE
14
15 SNIP
```

```
SCR 25
 0 These defining words are used to name parameters in an argument
 1 list. The CREATE .. DOES> action is done within ARG for args
 2 requiring 1 cell in the argument record, and within DARG for
 3 arguments requiring 2 cells (32 bit local variables).
 4
 5 Before ARG or DARG is invoked, two addresses are placed on the
 6 stack. These are the address of the action table to be
 7 associated with the type of parameter being defined, and the
 8 address of an area of memory that contains a set of headless run
 9 time support routines for each cell in the arg rec. For these
10 examples the run time support routines return the cell addr or
11 the cell contents (see CAR and CCR). ARG will compile these
12 addresses into the parameter field of the named argument. (See
13 ARG).
14
15 For a deviation from this general pattern see index

SCR 26
 0 £RELEASE will release n bytes of internally allocated space by
 1 advancing the stack pointer.
 2
 3 £CAPTURE compiles (£CAPTURE) and a following in line argument.
 4
 5 (£CAPTURE) uses n2 and its 1 byte in line argument to determine
 6 the address of a cell in the argument record. It allocates n
 7 bytes of internal space by moving the stack pointer, and places
 8 the address of the allocated area in the argument record cell.
 9 The value n2 is generally the initial content of the cell, and
10 will have been calculated by the word internal when the
11 argument list was read by the compiler.
12
13 The operations performed here minimise run time calculations and
14 allow us to set an addr determined at run time in an a parameter
15 type defined as a CCR. See matrix multiplication example.

SCR 27
 0
 1 (SET-INDEX) addr -- expects the address of an arg rec cell
 2 and stores a pointer to the location that will hold hold the limit
 3 and index of the subsequently entered loop.
 4
 5 (GET-INDEX) addr- n expects the address of an arg record
 6 cell that contains a pointer into the return stack previously
 7 set by (SET-INDEX). Returns n, the index of the associated loop.
 8
 9 INDEX-ACTIONS returns the address of the action table for
10 index. DEFAULT is always set for an index, by code in the
11 DOES> phase of index.
12
13 index is the defining word that names an index parameter. 0 is
14 compiled as the initial index value. The DOES> phase executes
15 when the index name is encountered during subsequent compilation
```

Errata

Note that throughout the preceding paper, the symbol # has erroneously been replaced by the symbol £. We regret any confusion caused by this error.