Staapl's Macro Forth for PIC18

TOM SCHOUTEN

August 4, 2008

Abstract

Staapl is a collection of abstractions for metaprogramming microcontrollers from within PLT Scheme. The core of the system is a programmable code generator structured around a functional concatenative macro language called Coma. Combining Coma extended with Forth's structured programming macros and a Forth syntax frontend yields Purrr, a Forth macro language with partial evaluation. This document describes a particular specialized instance of Purrr, namely the `staapl/pic18` dialect for Microchip's PIC18 architecture, henceforth called "The Forth".

1 Introduction

The Forth is a Forth dialect specifically tailored to flash ROM based microcontrollers. A Forth typically enables direct low–level machine access in a resource–friendly way while providing a solid base for constructing high–level abstractions. The Forth is built on top of a purely functional compositional macro language for called Coma. The Coma primitives are built from (virtual) machine language rewrite rules. In essence, Coma can be ported to any real or virtual machine, but in this document we concentrate on the port to Microchip's PIC18.

Staapl contains an interaction system that supports incremental development for tethered systems. The idea behind this is to keep what is good about a stand–alone Forth implementation, but simulate it on a host system such that the target code can be kept minimal. The interaction system is documented elsewhere.

The PIC18 port implements a *stack machine model* as a thin layer atop the machine architecture. The implementation consists of a collection of PIC18 assembly code rewrite rules[1]. This manual is intended for an audience familiar with assembly language programming and somewhat familiar with the ideas behind the Forth language.

1.1 Expansion and Contraction

To understand the Forth from a high level, it is instructive to use the *substitution model* of code translation. A program is essentially a collection of macros which will be *instantiated*, meaning they will eventually be translated to *executable machine code*.

A sequence of *words*, which are textual entities separated by whitespace, can be *expanded* to another string of words by *substituting* each word with the sequence comprising its definition. This process continues indefinitely until a word can no longer be expanded into a string of words. Such a word is called a *primitive*. These primitives have a direct relation to executable machine code. For example, the word `double` might expand to the string `dup +`.

Thus far this model describes just the semantics of a *macro assembler*. The Forth is an extension of this paradigm because it allows a process of *contraction* too. For example the sequence `1 +` might contract to the shorter sequence `increment`.

So words can be expanded to sequences, but sequences can also be contracted to words, or different sequences. This model is simple but powerful. It allows a unified approach to the problems of *optimization* and *metaprogramming*. The process of expansion supports *bottom up programming*: building larger things out of smaller things. The process of contraction allows *program specialization* through *parameterization* using general high level components to describe specialized efficient code.

In reality, instead of operating on concatenative code as described above for the higer level semantics, the processes of expansion and contraction is implemented as operations on target code using Coma *macros*. A macro

[1]This system is part of the public API of Staapl, and should make it straightforward to create different machine architecture backends. The rewrite rules system is accompanied by an assembler generator. Including an assembler in–core has the advantage of not having to serialize expressions dependent on target addresses.

is a function that generates machine code, or transforms previously generated machine code. Forth compilation (macro instantiation) is *incremental* and it reads from left to right. The inclusion of rewrite rules for virtual, non–compilable machine instructions is what enables metaprogramming and partial evaluation.

1.2 Rationale

The Forth is not a standard (ANS) Forth. It is fairly minimal, and takes from standard Forth just those elements that are essential to build a proto language: procedure and macro words, stacks, global variables and control structures. The key differences are the use of 8–bit data words, separate code and data spaces, and the lack of reflection, which is entirely replaced by layered metaprogramming.

So, why Forth? First, the absence of lexical variables and the simple semantics of function composition make it very convenient to use for metaprogramming. Program generation is just concatenation of sequences.

Second, the Forth paradigm by itself supports a reasonably high level of programming without requiring heavy run-time support while retaining precise control over the underlying machinery. Microcontroller programs often contain a mix of simple but highly specialized hand–tuned low–level code that has to be as efficient as possible, and a bulk of high–level management code that is complex but less time critical. Due to the use of stacks, Forth supports a referentially transparent programming style resembling *functional programming*. It allows factoring of programs into a collection of small functions called *words*. Such a style promotes reuse, layered abstraction and individual testability.

The Forth aims to be minimal from an on–target kernel perspective. It is a native Forth built entirely on top of macros, and doesn't requiring a runtime kernel. Effective boot code consists of a couple of instructions to setup the stack pointers and there is no further run-time overhead. Forth macros are composable, just like functions, and can be used to add language features that cannot be expressed using composition of procedure words alone. The macros make up a purely functional stack language based on function composition, and can use a rich dynamic type system (borrowed from Scheme) that can be used to perform compile time computations.

The downside of removing reflection and independence is the loss of interactive development. However, considerable effort has been spent on keeping

the system interactive through simulation, and reasonably *introspective*, just like a reflective self–hosting Forth. It is possible to inspect and modify machine code and data state while running, execute arbitrary code, and compile and upload code on–the–fly.

2 The Forth Language

2.1 How Forth?

A Forth program consists of a sequence of *words*. There are two classes of words. A *procedure word* refers to a program fragment that is represented as an individually executable chunk of machine code instructions. A *macro word* is a function that represents a compile time action, which eventually results in machine code. In this manual we abbreviate these names to *word* and *macro* respectively.

```
macro
: increment  \ n -- n+1
    1 + ;

forth
: double-increment \ n -- n+2
    increment
    increment ;
```

The words `macro` and `forth` switch between macro word and procedure word definitions respectively. In the code above, `increment` is a macro while `double-increment` is a procedure word. The backslash character `\` is used to start a line comment.

One way to view the Forth is as a macro assembler. There is a fairly direct relationship between a program text and compiled machine code. I.e. `+` is the machine's addition operator. Compared to a traditional macro assembler, macros do not only expand to asembly code, they also *recombine* with previously generated assembly code. In the example above, the procedure word `double-increment` corresponds to the code

```
double-increment:
       addlw  (1 1 +)
       return 0
```

Compared to standard Forth, the Forth thus uses a simplified *implicit* metaprogramming syntax. Standard Forth uses *explicit* metaprogramming in the form of the words `[` and `]` which switch between compile and interpret mode. Here, the programmer does not explicitly indicate which code will run at compile time. Instead, the programmer decides only which words are *instantiated* and which will be *inlined* using the `forth` abd `macro` sections respectively. All compile time behaviour is implemented using rewrite rules, which are specified outside of Forth programs. More about that later. Most compile time computations are based on partial evaluation. If possible, some macros perform computations instead of generating code to perform a computation at program run time.

The example above illustrates this use of partial evaluation. The double occurence of the word `increment` has been partially evaluated to the machine operation `addlw (1 1 +)`. The code between parenthesis indicates a function that can be evaluated at compile time, here producing the numeric value 2. The machine instruction `addlw` ADDs its Literal argument to the Working register representing the top of the data stack.

Partial evaluation is an optimization technique often used in the implementation of functional programming languages. This approach works for The Forth because it is possible to interprete a *subset* of the procedural Forth dialect as a purely functional stack language based on function composition, and the realization that function composition is associative. This means that the time at which function a composition occurs becomes a parameter to play with. This makes it possible to move some of the composition to compile time, as is shown in the example above. Composition at run time is implemented by the target machine's instruction flow.

However, it is not correct to call this partial evaluation in the strict sense, because the host's type system is different from the target's. Let's go back to the example `(1 1 +)` above. The integer operation `+` when it is done at compile time has infinite precision. The same goes for the other integer arithmetic operations. In order to be able to represent the result on the target, results of computations need to be truncated to the data word size, which is 8 bits for data and 16 bits for code addresses. This technique enables the use of arithmetic operations that are not available at run–time in a way that is fairly transparent: it is possible to *read* source code looking only at the high level meaning of code, without worrying when the evaluation happens.

In order to effectively *write* programs, the programmer does have to worry about whether a certain construct is compilable. In practice however, this is

quite straightforward. One way of looking at the approach is to view procedural The Forth as the *projection* of a clean purely functional, compositional, high–level language, onto a restricted procedural semantics.

The partial evaluation of arithmetic expression is but one example of this powerful construct. By relaxing the requirement that all macros need to be *compilable* in isolation, one can use macros to construct language *idioms*. Idioms are sequences (compositions) of macro words that yield some compilable construct. A non–compilable construct is called an *ephemeral* macro. An example of an ephemeral macro is `begin`. It is not compilable without a balanced `again` or `until`. This approach enables the use of high level compile–time operations as long as they eventually lead to constructs representable in low–level form, or can be *projected* to some representable construct, as is the case for numbers.

2.2 Tool Chain

In Staapl, the meta–programming and code generation occurs on a system which is *different* from the one executing the final machine code. Two computer systems are involved: the *host* system runs a compiler program to produce compiled programs from source code while the *target* system eventually executes these compiled programs. The main reason for this distinction is of course the lack target resources to support the tool chain.

The host–target distinction is important from the point of *interaction*. Procedure words exist physically on the target chip in the form of machine code, and can be *executed* interactively. Macro words exist only in the translation phase from source code to machine code, and have no direct representative as an accessible code word, and as such cannot be executed. However, The Forth includes the possibility of executing macros that produce constant values, as if they were present in compiled form. Similarly, some basic arithmetic operations are simulated if they are not instantiated as machine code.

2.3 Factoring

The most compelling property of Forth is its ease of performing *composition*: syntactically, a program is merely a concatenation of the names of sub–programs, represented as words. If a sequence of words occurs in more than one place in a program, one can give a name to the sequence, and substitute

the occurrence of the sequence in the source code by the newly defined name. This technique of program evolution is called *(re)factoring*, and is essential for controlled growth.

In short, when a pattern emerges in the source, it is time to increase the abstraction level and perform some *correctness preserving program transformations* to isolate the code pattern and give it a name. In Forth this usually means to change the order of some words so a sequence can be cut out and replaced by a single name referencing a procedure or macro.

Factored *procedure* words are important because they allow physical (on chip) code reuse, which limits the necessary target code space. Factored *macro* words are important because they allow the construction of language features that are not expressible as a composition of procedure words.

Macro words can be composed just as easily as procedure words. A category of words necessarily implemented as macros are *control words* which change the flow of control to something else than the default sequential word execution. Another example is *optimization*; some macros can be combined to code that is simpler or has more efficient representation than the sum of the parts. A third example in is the use of *idioms*, which are sequences of macro words that behave as if they were simply composed words, but have only a meaning when combined in a certain way, allowing the expression of constructs that are impossible to express as procedures. Compile time computations have access to a type system that is substantially richer than the raw machine words used at run time.

3 Programming Model

The Forth is a compiled language, and works without a run–time kernel. A program is defined in terms of *composable macros*. Compilation of a The Forth program is a function which maps a *source file* and a *dictionary* to an updated dictionary and a chunk of binary machine code. It is factored into the following steps:

- Parsing of program text into macros and procedural code.
- Construction of an extended compiler from the base compiler and the named macros.
- Compilation of the code body to assembly language, using the extended compiler.

- Construction of dictionary items for the procedural code, and assembly of binary machine code, statically bound to functionality represented by the updated dictionary.

The *dictionary* of target words is a symbolic index into binary target code. It contains information necessary to execute code during interactive development.

There are two main ways of structuring the namespace of applications. One is built on top of PLT Scheme module system which allows the construction of a program as a directed acyclic graph of component modules, each with its own namespace.

Alternatively, following the more traditional Forth style, programs can be constructed incrementally in linear layers in a single flat namespace that allows redefinition of words. All code is early bound, which means upper layers cannot influence functionality in lower layers. All late binding needs to be implemented explicitly using vectored code.

The Forth uses *partial evaluation* as an interface to the metaprogramming system. This is implemented using primitive macros expressed as machine code rewrite rules, and a concatenative composition mechanism.

4 Language Features

This section deals with language features that are different from standard Forth.

4.1 Partial Evaluation

In order to see how partial evaluation works, it is a good idea to look at how it is implemented. The transcript below shows the effect of incremental compilation. Compilation works by pushing data on a *compilation stack*. The data on this stack is *dynamically typed*, with the type indicated by a symbolic type tag.

We start with entering a number

```
>> 1
        qw      1
```

The first line is the compilation input, the remaining lines are the contents of the compilation stack. The type tag `qw` indicates a Quoted Word. In order to be compilable, the word needs to be reducable to a numeric value. We go on by entering another number.

```
>> 2
        qw      1
        qw      2
```

There are now two numbers on the compilation stack. Next we enter an operation.

```
>> +
        qw      (1 2 +)
```

The result is a quoted word, where the word can be reduced to a number by evaluating a computation. This is the simplest example of a compile–time computation[2].

When a compilation is done, all the data left on the compilation stack needs to represent a compilable program. In this case, we have a single quoted number 3, which is certainly compilable. Let's start over with a clean compilation stack and type just the operation.

```
>> +
        addwf   POSTDEC0, 0, 0
```

This is quite different. What is present on the compilation stack is an assembly program that will perform the computation `+`. It works by adding the second word on the run time data stack to the working register, and then popping off the second word. Popping is done by a post–decrementing read: read the value pointed to, then decrement the pointer. This is equivalent to popping the 2 top numbers, adding them and pushing the result.

These two examples illustrate how partial evaluation is implemented: by inspecting the compilation stack, the macro `+` knows what code to generate: either the value can be computed at compile time, and the resulting program just quotes the resulting number, or the computation has to be

[2]Note that his gives a clear example of the relation between the Scat and the Coma languages. The stack of machine code is simply used as the parameter stack of another stack language. For the arithmetic operators, Coma is basically Scat looking *inside* the tagged data structures.

postponed until run time, in which case the appropriate machine instruction is generated.

In the case of the binary operator `+` there is a third possibility: one of its operands might be known at compile time. Starting with a clean compilation stack, providing only one argument yields

```
>> 1 +
        addlw   1
```

which adds the number 1 to the working register, which implements the top of the data stack. The resulting code is still an operation, but it is less general than the one before. The composition `1 +` has been evaluated to a single machine instruction[3].

4.2 Nested Constructs

Because Forth syntax is merely a succession of words, creating nested structures requires some kind of stack[4]. For procedure word nesting, this is the *return stack* which is active at run–time. It records where to continue after terminating the current procedure.

For nested language structures created using macros, this stack is called the *macro stack* or control stack and is accessible at compilation time (macro execution time) using the macros `>m` and `m>`. All words that implement nested structures are defined in terms of these two words. For example

```
macro
: begin  sym dup >m label: ;
: again  m> jump ;
```

[3]This illustrates that while the macro language is purely concatenative, the operations of concatenation and compilation do not commute. In other words, compilation preserves semantics but the machine program resulting from concatenation of individually compiled macros might be different or might not even exist when trying to separate an idiom. Also, in the presence of projected semantics for number operations, even the semantics is slightly different.

[4]For the coma Coma language, which has an s-expression syntax and inherits program quotations from Scat, these structured programming constructs are not necessary because they can be replaced with higher order macros thus keeping the language purely concatenative. For the implementation of the higher order forms for conditional execution and looping however, the mechanism described here is used.

The macro `begin` creates a new code identifier, duplicates it and places a copy on the macro stack before creating a jump target using that identifier. The word `again` pops the symbol from the macro stack, and uses it to compile a jump instruction. As long as there is a balancing `again` for every `begin`, the resulting code is compilable.

Too many occurences of `begin` lead to non–compilable constructs because the macro stack is not empty. Too many occurences of `again` lead to non–compilable constructs because of macro stack underflow: `m>` will be evaluated without values on the stack.

In the definition of `begin` there is the word `sym`, which creates a new symbol. In an of itself `sym` is not compilable, because the symbol value is not representable on the target system. However, the words `label:` and `jump` will consume symbol values to yield constructs that are compilable: assembler labels and jump instructions.

It is legal to use `>m` and `m>` anywhere in macro code as long as the eventual use is balanced. A typical use is in metaprogramming constructs which use a literal value multiple times. For example, a macro that converts a number to a two byte value can be written as

```
macro
: lohi    \ number -- low high
    dup >m
      #xFF and
    m>
      8 >>> ;
```

This will take a literal value, duplicate it and put one copy on the macro stack. The low byte literal is computed by applying a bitmask. The high byte literal is computed by retreiving the value from the macro stack, and shifting its bits to the right by 8. Note that the shift operator `>>>` is only defined at compile time and is thus an ephemeral macro. If the macro `lohi` occurs in a code composition after a computation that yields a literal value, the composition is compilable. The computation runs at compile time so the intermediate results use infinite precision: there is no 8–bit limit for data representation.

4.3 Named Macro Argument

Instead of using the macro stack, in some cases it is a better idea to use local names in macro definitions[5]. Local names are sometimes frowned upon because they make factoring more difficult if not used wisely. However, writing low level macros it can sometimes be convenient to reduce the stack shuffling that is otherwise necessary. This is the same macro expressed with the syntax for local names. This binds the words between bars `|` to the respective macro stack literals, associating the top literal to the rightmost word.

```
macro
: lohi | number | \ -- low high
    number #xFF and
    number 8 >>> ;
```

4.4 Quoting

By default, all symbolic words refer to their primary semantics. Either this happens by compiling a reference to an on–target precedure word, or the execution of a macro which will inline code or perform other compile time operations. This primary behaviour can be changed *only* by quotation.

The word `'` pronounced *tick* or *quote* will wrap a reference to the following word as a literal value that can be manipulated at compile time in the same manner as numbers or other compile time data objects. This reference is represented by the macro word that compiles the behaviour of the word. If such quotations survive to the target, they will be encoded as a number representing the address if the procedure word that corresponds to the reference. If the reference corresponds to a macro, it has to be *unquoted* using one of the higher order macros like `run`, `execute`, `compile`, ...

The function of the word `run` is to simply undo the effect of the quote. If `foo` is defined, `' foo run` is equivalent to `foo`. If a literal value does not precede `run`, the semantics is delegated to the procedure word `~run`[6]. The `execute` word has a lower level semantics and operates on addresses only, while `compile` is like `run` but won't delegate to `~run`.

[5]For Forth procedures, local names are not supported.

[6]This is a general pattern. Macros that perform partial evaluation will delegate to a procedure identified by a tilde prefix. In the core compiler, these words are stubs that can be redefined whenever the run–time behaviour is implemented, but this is not necessarily so.

4.5 Structured programming

Forth control words are implemented in terms of conditional and unconditional jumps to (anonymous) target code labels. Anonymous target code labels can be created at compile time using the word `sym`. Passing such a value to `label:` creates a jump target. Obviously, this can happen only once per unique code label. Passing the label to `jump` creates an unconditional jump, passing it to `or-jump` creates a conditional jump if false. Togehter with the macro `m-swap` whcih exchanges the two top values on the macro stack, these words can be used to create the classical structured programming words.

```
macro
: if      sym dup >m or-jump ;
: else    sym dup >m jump m-swap then ;
: then    m> label: ;

: begin   sym dup >m label: ;
: again   m> jump ;

: do      begin ;
: while   if ;
: repeat  m-swap again then ;
: until   not while repeat ;
```

This is a specification of the control flow words with stack effects. The effect on this stack is indicated as here as `m:` < in > `--` < out >. Similarly using `x:` for the *auxiliary stack*[7].

```
if     \ ? --      m: -- label
else   \ --        m: l0 -- l1
then   \ --        m: label --

for    \ count --  m: -- label   x: -- loopcount
next   \ --        m: label --   x: loopcount --

begin  \ --        m: -- label
```

[7]Because the return stack a limited resource on PIC18 and much wider than the data word, an extra byte stack is used for storing temporary data.

```
again  \ --        m: label --
until  \ ? --      m: label --
while  \ ? --      m: l0 -- l0 l1
repeat \ --        m: l0 l1 --
```

4.6 Booleans

In the Forth all predicates are macros that can be optimized into efficient machine language conditional branch and skip instructions. By convention macros that produce boolean values are postfixed by a question mark `?` character. The macro `if` can consume these ephemeral boolean values and generate the appropriate conditional jump instruction. Take a (simplified) example from `serial.f` the serial port driver

```
macro
: rx-ready? \ -- ?
    PIR RCIF high? ;

forth
: receive   \ -- byte
    begin rx-ready? until
    RCREG @ ;
```

The macro `rx-ready?` generates an ephemeral boolean derived from the bit at position RCIF (ReCeive Interrupt Flag) in the special function register PIR (Peripheral Interrupt Register). This boolean is consumed by the `until` macro word, which is eventually implemented in terms of the `if` macro word (which itself is implemented in terms of the primitive `or-jump` word). This code illustrates a useful pattern: abstract each condition in a macro, naming it appropriately with a postfix question mark to make the code that uses the condition more transparent.

4.7 Tail Call Optimization

A procedure word followed by the `;` or `exit` instruction is translated into a jump. This allows for the use of *recursion* to write loops, without overflowing the return stack. The following code does the same as `receive` in

the previous example by calling itself recursively until the condition becomes true. This example has multiple exit points (see below).

```
: receive
    rx-ready?
      not if receive ; then
    RCREG @ ;
```

4.8 Predicates for Inspection

The Forth contains a collection of predicates that will produce a boolean *without consuming the original arguments*. This contrasts with some standard Forth predicates. These predicates are named by appending a question mark `?` to the standard Forth name. For example:

```
=   \ a b -- ?
=?  \ a b -- a b ?
```

While these constructs are somewhat essential for the factorization of efficient control macros, apparently they are a bit confusing for people used to Forth. So beware!

4.9 Indirect memory access

The PIC18 architecture has separate instruction and data memory spaces. The Forth uses two pointer registers to access these memories: the `a` (Array) register accesses volatile RAM, and the `f` (Flash) register accesses non-volatile programmable ROM memory. Indirect addressing using the `@` and `!` words is only supported for global variables, which are implemented as literal addresses. Indirect access can be implemented by overriding the `~@` and `~!` words.

Indirect access through the `a` register might be more convenient and efficient. The words `@a`, `@a+`, `@+a` and `@a-` use the 4 relative addressing modes on the PIC18: indirect, postincrement, preincrement and postdecrement. The `a` register can be accessed through the low and high bytes `al` and `ah`. An abbreviation for storing both high and low words is provided:

```
: a!!   \ lo hi --   | store 2 bytes in the a register
    ah ! al ! ;
```

Similarly, to read Flash memory, the words `@f`, `@f+`, `@+f` and `@f-` can be used. The `f` register can be accessed similarly through the byte parts `fl` and `fh`.

5 Control Flow

This sections deals with ways to escape from sequential code execution in addition to the standard structured programming idioms. The unifying idea is that you can use two stacks to roll your own control abstractions. The *return stack* is used to record nesting state at run time and to implement computed jumps. The *macro stack* is used for compile time computation of control flow.

5.1 Multiple Entry and Exit Points

Since procedure words are just assembler labels representing machine code addresses, and straight line code is translated to straight line machine code, there is no reason for a word not to have multiple entry points. In fact, this can be quite convenient. The following code defines two words.

```
: double-increment
    1 +
: increment
    1 + ;
```

The second one increments the top of stack value by one, while the first one increments the top of stack value by two. The code in the first definition just *falls trough* to the last definition as if the sequence "`: increment`" wasn't there. Similarly, a procedure word can have multiple exit points. In the code

```
: safe-turn-on
    problem? if ; then turn-on ;
```

the word `turn-on` is executed if the `problem?` condition is false. If the condition is true however, the word exists trough the `;` word inbetween `if` and `then`.

Macros can't have multiple entry points, and need to use explicit tail calls to get to this behaviour. However, they do support multiple exit points, where the exit `;` is implemented as a jump past the end of the code generated by the macro.

5.2 Vectors

A *vector* is a variable containing a word address. The interface consists of two words

```
invoke   \ var --     | execute the code stored in var
->       \ var --     | set var to point to code
```

and a parsing word

```
vector   \ <name>     | create a vector variable
```

The word `invoke` is implemented as `2@ execute/b`, which fetches 2 bytes from a double variable and passes them to the `execute/b` word which executes the code pointed to using byte addressing. The word `->` stores the address of the code following it in the variable, and then *exits* the word in which it occurs. So it will not execute the code after the arrow, jus change the value of the variable. I.e. the code

```
vector current-op
: will-inc   current-op -> 1 + ;
: will-dec   current-op -> 1 - ;
```

defines a word `will-inc` that when executed changes the subsequent behaviour of `current-op invoke` to `1 +`. Similarly, the word `will-dec` changes the subsequent behaviour to `1 -`. By itself, the words `will-inc` and `will-dec` don't do anything except for setting the *value* of the vector variable `current-op`: the word `->` is an *exit point* for these setter words.

As the name in the example indicates, vectors can be used to set *current* behaviour, folling the Forth mantra "Don't set a flag, set behaviour."

5.3 State Machines

The `route` word is a different mechanism for implementing dynamic behaviour. It can be used to construct *byte code interpreters* using dispatch tables. While vectors are generic because they can point to arbitrary code, byte codes are more specific: they map state representation (a number) to behaviour by using explicit *interpretation*.

Vectors work well if there is a small number of invokation points and a large number of state changing words, or for implementing late binding.

When the number of alternatives is fixed, i.e. in the implementation of *finite state machines*, byte codes are often easier to use. The use of `route` is best illustrated with an example of how it would appear in code:

```
: abcd \ bytecode --
    route
        aaa ., bbb ., ccc ., ddd ;
```

Here the word `.,` (a sideways semicolon) behaves as the `;` word, while telling the compiler that the code after it is reachable, so it won't be optimized away. Because there is really no other use, the word `.,` can be seen as a jump table separator. With this code the code `0 abcd` is equivalent to `aaa`, `1 abcd` is `bbb`, etc...

This works only when the words before the separators are procedure words. For a procedure `aaa`, the sequence `aaa .,` consists of a single jump instruction due to tail call optimization. The word `route` simply adds its argument to the base address of the table[8].

5.4 Cooperative Multitasking

The heavier approach to sequencing is to use *tasks*. State machines can be the right solution for some problems that do not require *recursion*. When procedure nesting is required but a piece of code does have some control state in isolation of other code, tasks are a good solution. A task has a separate *execution thread*.

Each task's state consists of a set of stacks. More specifically a return stack, a data stack and an auxiliary stack. Usually it is a good idea to also save a separate copy of the `a` and `f` registers per tasks. The Forth contains primitives to implement your own multitasker in the file `pic18/task.f`. It implements the words `suspend`, `resume` and `swaptask`.

```
suspend  \ -- task       | freeze current task context
resume   \ task --       | make task context current
swaptask \ task var --   | swap task with the task in var
```

[8]This is a low–level construct that is easily exploited for creative use. Every *instruction slot* in the jump table can be filled with anything that produces a single machine instruction. It is also possible to leave out the separators to just jump into a sequence of words skipping the firsts couple.

Usually the word that performs task switching is called `yield`. In the common case where there are only two separate tasks, this word simply switches between the two tasks, using a single variable to point to the representation of the *other* task:

```
variable other
: yield
    suspend
    other swaptask
    resume ;
```

The difficulty in using tasks on a low level is how to create them. This generally requires manually allocating resources for the tasks's stacks. For the 2–task case the `other` task can be booted by the word

```
: start-other-task
    suspend other !   \ suspend current task
    #x10 rp !         \ use half of the hardware return stack
    #x50 xp !         \ for rp, and use a region of RAM for
    #x60 dp !         \ the byte stacks dp, xp
    task-init-code ;  \ start the task's code body
```

This discards anything stored in the `other` variable, and performs manual context switching by changing the 3 stack pointers to a free memory location, before running the task's initialization code.

A more complicated scheduler can be implemented by replacing the code between `suspend` and `resume` in the `yield` code above. For example, code from `pic18/buffer.f` could be used to create a *round-robin* scheduler which executes a couple of tasks in a circular fashon[9].

5.5 Tasklets

If the number of tasks is small, and there is a clear hierarchy, *tasklets* can be used, which have the semantics of low priority interrupts. Whenever a machine interrupt handler finishes, it can *spawn* a tasklet by simply jumping

[9]For PIC18, the hardware return stack is a fairly limited resource. If a lot of tasks are required, explicit copying of the stack might be necessary. An alternative is to write a VM on top of the native Forth which doesn't use the hardware stack. There is a draft version of a 16–bit direct threaded interpreter available.

to some code after re–enabling interrupts. As long as the tasklet finishes before any other invokation of the ISR spawns a new one, this system is stable.

If the tasklet is allowed to pre–empt the main task, it doesn't need a set of stacks of its own, since it is never in a running state if the main task is active. Tasklets are thus more efficient than genuine tasks.

An example: you have a receiver consisting of a main application loop, a timer interrupt handler and a tasklet. The interrupt handler performs analog–to–digital (ADC) conversion and a single filter step. The tasklet recovers the next symbol from the accumulated filter state and possibly performs some synchronization operation after say 16 timer interrupts have elapsed. The main application loop reads full bytes from an input buffer.

5.6 Procedures or Macros?

At several points during the development of reusable library code I ran into the question: am I going to use macros or procedure words. To answer the question generally, it should be translated to: should this code be *fast* or *small*.

To understand the main reason why this question pops up it is necessary to look at the PIC architecture, where indirect addressing is quite expensive. It is obvious that the PIC has a bias toward *static* objects: it has quite some provisions to deal with memory addresses that are known at compile time, so they can be inlined in the code. However, dynamic access which is necessary for object abstractions requires the use of the FSR registers. Of these there are 3, and they are used as data stack pointer, auxiliary stack pointer and the `a` register. Whenever an indirect access occurs, the `a` register needs to be saved, set and restored[10]. As a consequence, dynamic objects are about an order of magnitute more expensive than static ones.

This bias toward static code eventually reflects in the design of the The `pic18/` library code: it has a lot of provisions for static objects in the form of macros, especially at points where speed might be an issue, for example the buffer code in `pic18/buffer.f`. These are somewhat harder to use because they often need to be *instantiated* explicitly.

[10]To get rid of save and restore it is possible to assume throughout the program that the register can get clobbered. However, this is a global constraint which makes it harder to enforce.

5.7 Compilation Unit

The Forth can use PLT Scheme's hierarchical module name management tools. However, for static low–level code it is sometimes convenient to link code components by loading them into a single name space. This is essentially an extra composition mechanism, on top of parameterized instantiation of macros.

I.e. The interpreter that supports target interaction is built like that. The code in `pic18/interpreter.f` references the words `receive` and `transmit`. In order for this code to work it needs to be included into a namespace using which has these words defined. For this the parsing word `load` can be used.

6 Essays

Some related articles.

6.1 Effective 8–bit Programming

Nothing limits Purrr to be implemented for larger word sizes. However, for PIC18 the language is organized in a way to make 8–bit data cells practical, while retaining a larger (machine specific) return stack size.

The ANS Standard explicitly prohibits an 8–bit cell size, setting the minimum size at 16 bits. It requires data stack elements, return stack elements, addresses, execution tokens, flags, and integers to be one cell wide. While the PIC18 Forth is non–standard for a lot of different reasons, this requirement really kills any hope for standard compliance. However, it is my opinion that an 8–bit Forth has a reason of existence, despite the limitations of different code and data cell sizes.

The Forth contains some 16–bit library routines, but using them can be cumbersome. The Staapl distribution contains a direct threaded 16–bit virtual machine written on top of the native 8–bit Forth which does enable a more standard approach. It comes with its own interaction system (currently broken in 0.5.x).

In the PIC18 Forth the 21 bit wide hardware return stack is used. Only the low 16 bits are used, leading to a representation of a procedure word as a two cell value. Because of its larger size and fixed depth (only 31 words), a separate byte stack called the `x` stack or auxilary stack is used. I.e. this

stack is used to store the loop counter in `for ...next` loops. It can be used as an alternative to the return stack for temporary value storage.

The problem points when working with 8 bit data words can be identified as limited precision for mathematical operations, limited practical data buffer sizes, limited loop size, and difficulty of representing code as data.

For math, you're basically out of luck and need to resort to tricks. The Forth has some 16–bit math routines, but math–intensive applications usually work better on larger word size (real or virtual) machines, and as such are not considered part of the application domain. Building a VM or macro language on top of the PIC18 Forth is the way to go here.

On the other hand, don't forget that logic is your friend! A lot of problems can be solved by creatively using `and`, `or`, `xor`, `-`, `+` and the shift and rotate operations together with the carry flag. The Forth exposes the these low level machine details to give you the means to create your own abstractions on top of them, using either procedure or macro words. Note that hexadecimal numbers are specified like `#xF0`, and decimal numbers like `#x11110000`. The Forth does not use a `base` word: all numbers are decimal, unless they are indicated as hexadecimal or binary. Also note that the PIC18 contains a hardware multiplier for $8 \times 8 \rightarrow 16$ unsigned multiplication, which can be used to build your own multiplication abstraction. The Forth18 contains some code for a 24 unsigned MAC operation to implement digital filters.

The problems caused by large data buffer sizes can usually be avoided by proper abstraction. In addition the intended target chips usually have small memory sizes, so large buffers are rare. When they do occur, it is usually easier to perform buffer management on a byte and a block level: adding hierarchy to a solution can often not only solve a word size problem, but also bring up solutions that are easier to write down. The same argument goes for limited loop sizes. If you need a `for ...next` loop that executes more than 256 times, just nest two of them. Even better: put the inner loop in a separate word and try to see if the code now tells you why you're better off using this hierarchical solution in the first place.

For the problem of effectively representing code as data, byte codes bring a simple solution. A byte code can represent up to 256 different words. The easiest way to do this is to use the word `route` to construct a jump table. Here is a code fragment from the boot interpreter taken from the `pic18/interpreter.f` file. It interprets numbers (tokens) ranging from 0 to 15 by mapping them to code.

```
: interpret \ token --
    #x0F and route
                .,  receive .,  transmit .,  jsr    .,
        lda     .,  ldf     .,  ack      .,  reset
        n@a+    .,  n@f+    .,  n!a+     .,  n!f+   .,
        chkblk  .,  preply  .,  ferase   .,  fprog  ;
```

The `route` here is used to perform something akin to procedure table lookup. The word takes a single argument n and jumps to the n–th machine word following itself. The table above contains 16 machine word entries. To make sure jumps remain inside the table, before `route` the top 4 bits of the token are chopped off using `and`. All words in the table, except the empty one and `reset`, revert to procedure words, for which which the idiom `receive .,` compiles to a single machine word jump instruction.

The first slot is empty: a `.,` word by itself compiles to the RETURN instruction, which in a route table acts as a no–op. The `reset` word is a macro that compiles to the RESET instruction, also taking a single machine word slot.

6.2 The E2 Bus

The idea of the E2 bus is to provide *data* and *power*, and a simple synchronization mechanism over just 2 wires, in a way that enables *bi–directional* communication. Together with the The Forth boot monitor this is intended as a programming and debugging network for developing multi–target applications. The connector for this network can be combined with the power connector in stand–alone mode, and is thus non–invasive. The circuit board requires a diode for power rectification, a capacitor large enough to span a single bus timing cycle, and a possibly interrupt enabled I/O pin.

The bus is defined in terms of a 4–phase state machine for receive and transmit modes. The logic encoding uses a 8N1 state machine, where the redundant stop bit can be used for other purposes. In total, a single byte transfer consists of $4 \times 10 = 40$ elementary clock pulses.

On the master side, operation is very straightforward. The only thing that is needed is an elementary clock event. The idea is to have a central hub that can drive multiple slaves, each on a separate pin.

Because the bus is *half–duplex*, synchronization is necessary in the data protocol to prevent bus collisions. This is intentionally left unspecified. For

the The Forth bootloader protocol this consists of a Remote Procedure Call (RPC) message protocol: master initiates transfer, slave acknowledges transfer with a status byte and/or data.