The Sheep

TOM SCHOUTEN

August 4, 2008

Abstract

This document describes The Sheep, a simple 1–bit sound synth implemented in Staapl's PIC18 Forth dialect `staapl/pic18`, and can serve as a tutorial.

1 Introduction

This document presents Sheep, an application written in Staapl's PIC18 Forth dialect, henceforth called "the Forth". It serves as an introduction to language oriented programming through writing a *domain specific language* (DSL). The synth engine's output is a simple binary on/off switch, which allows the main focus to shift to dealing with time. The DSL will deal mostly with the construction of abstractions for control flow through the use of procedures and macros.

The intended audience consists of the two fields I'd like to bring together with Staapl: those familiar with a higher level language like Scheme and those familiar with Forth or low level assembly programming. If you're familar with both it should still be enjoyable. If for some reason the subject of this document interests you but it is too dense, please ask questions! I'm always interested in finding out how to write better documentation. I've included some more general comments in the footnotes that should help to provide some background information. It is my intention to make Staapl accessible to a wide audience of programmers, even if it at first sight it has a complex structure.

This document should be used as a complement to the PIC18 paper, which describes the main differences between Staapl's macro Forth approach

and standard Forth, and introduces the basic building blocks and the way macros are used.

2 Building a Language

The basic idea behind *bottom up* program design is to gradually increase the level of *abstraction* by writing a high level language in terms of a low level one. Most languages support some form of *procedures* as their basic means of abstraction. This gives a single name for a bunch of operations. The names of these procedures form an *interface* their functionality.[1]. In language like Forth (or Lisp) which have simple syntax, *macros* are available as an additional abstraction mechanism. They allow the construction of language constructs that cannot be expressed as procedures alone. A macro gives a single name for arbitrary source code constructs. In the Forth we use the term *language layer* to refer to a collection of definitions for high level *procedure* and *macro* words.

In the following I will illustrate this principle by talking about the implementation layers in the Sheep synth, gradially moving from a low to high abstraction.

[1] Abstraction makes programming simpler. Using an interface one does not need to worry about how the details are implemented. The details are conveniently hidden to make it possible to express oneself in terms of more abstract primitives. Next to conceptual clarity that comes from dividing a complex solution into a composition of smaller specific solutions, abstraction also provides a mechanism for program component *decoupling*. The usage of interfaces makes it possible to *change* one part of the program without having to change other parts. Because it is simply impossible to oversee all the details of any sufficiently useful program all the time, decoupling is the only workable strategy we have to write adaptable softare. Note that abstraction becomes interesting when the one that knows the details and the one that does not are the same person! Once you can trick yourself into switching between abstraction layers, programs becomes a lot easier to manage. When you have things worked out, it is usually a good idea to chop a solution in pieces: literally put them in different files, and force yourself to *only* use clearly defined interfaces.

3 The Sheep Primitives

The low level synth engine API[2] consists of the words below.

```
engine-on  \ --
engine-off \ --
synth      \ variable: containing the synth patch
posc0l     \ variable: low byte of OSC0 period
posc0h     \ variable: high byte ...
posc1l     \ same for OSC1
posc1h
posc2l     \ same for OSC2
posc2h
```

The synth patch is a byte with the following bit fields.

```
01    \ mixer: silence, xmod, reso, osc1
4     \ sync:0->1
5     \ sync:0->2
6     \ sync:1->2
7     \ osc1:noise
```

The synth consists of 3 square wave oscillators that can be synced together in different configurations, and an output mixer that combines the oscillators into a single output bit. The first two bits determine one of four mixing algorithms. The sync bits determine phase reset synchronization between the 3 oscillators, and the last can set oscillator 1 in noise generation mode.

This is our starting point. At the lowest level, the synth *patch language* is merely a couple of configuration variables, and two words implementing an on/off switch. For the interested, the hidden low level code can be found in `synth-core.f`[3]. On top of this primitive patch language we will build the

[2]Application Programmer Interface. The synth API is the collection of words to interact with the implementation of the synth engine which itself is hidden.

[3]This API succeeds in providing a simple way of producing stationary sound. In order to get this simplicity we give up the capability to perform arbitrary algorithms to make sound. By *abstracting* the solution (use a *synthesizer machine* to make some stationary sound parameterized *only* by this configuration data), we limit the number of possible sounds we can make, but greatly simplify the expression of a useful set of abstractions that will implement non-stationary sounds *on top* of this primitive API. The implementation that actually produces the stationary sound is hidden. It is a background task that periodically scans the variables to drive a couple of *timers* present as hardware on the PIC18 chip. These timers generate *interrupts* that lead to changes in the output state.

next level of abstraction. In the file `synth-control.f` there is a collection of words and macros that manipulate the configuration variables in a more high–level way.

I declined to mention a complication in the interface. Because the synth engine updates are implemented as an *interrupt service routine* or ISR which responds to timer interrupts, care has to be taken that the state update is performed in a way that ensures the ISR can see only a *consistent* state[4]. This can be done by disabling interrupts during the write.

This is called a *leaky* abstraction: the period variables are a nice way to represent the synth config, but using them is still cumbersome, because we need to know what rules to follow to change their value. So what do we do? We abstract the problem! In the `synth-control.f` file there are a couple of words that perform the correct operation for writing to period registers, without having to worry about this interrupt business:

```
_p0 \ lo hi --    | write lo an hi period bytes for OSC0
_p1 \ lo hi --
_p2 \ lo hi --
```

In the Forth the following convention is used. If a word starts with a *underscore* it will produce and/or consume *double* values[5]. For the 8-bit PIC18 Forth this means two bytes to represent 16-bit values. For ease of use, there are also 8–bit variants of these words:

```
p0 \ hi --    | write lo an hi period bytes for OSC0, OSC1
p1 \ hi --    | and OSC2, but compute those from a single byte
p2 \ hi --    | by multiplying it with 257
```

[4]A timer interrupt can happen at any time, so if some part of the program is changing the 2 state variables associated with the oscillator period, there is a possibility that an interrupt occurs *after* writing one of the bytes, but *before* writing the other. In other words: writing the 2 period values should be governed by a word that can ensure the operation happens *atomically*, meaning the intermediate state is never visible to the ISR. In general, issues of assuring consistent used of data shared between tasks that can interrupt each other arbitrarily is a difficult issue. It is OK to ignore the details here.

[5]This convention is there to make a Forth mapping to Harvard architectures easier. PIC Microcontrollers have separate code and data memories that moreover do not have the same wordlength. Standard Forth requires code and data cells to be the same size. In order to keep the road open to build a Forth with a 16–bit data cell on an 8–bit machine, in Staapl this prefix convention enables a simple identification of code that is 16–bit data cell compatible.

The number 257 is chosen so a single byte can represent the entire range of a 2–byte number. It's also simple to implemement: multiplication by `257` is the same as multiplication by `#x101` and thus a simple `dup` will do the trick.

New words can now be defined to facilitate the interaction with the words `_p0`, `_p1` and`_p2`. One extension uses a table internally to convert note and octave numbers to periods. See again the `synth-control.f` file for the implementation. Here I mention just the interface:

```
  octave \ o --   | set the current octave.

  note0  \ n --   | set OSCx to the frequency
  note1  \ n --   | of a note in the current octave,
  note2  \ n --   | counting from 0 -> C, 1 -> C#, ...
```

In addition to writing, some algorithms might find a need to update the current period instead, i.e. portamento. The following words can be used to retreive the double words from the period registers.

```
_p0@  \ -- lo hi  | return the contents of period register for
_p1@  \ -- lo hi  | OSC0, OSC1, and OSC2
_p2@  \ -- lo hi  |
```

The synth uses 4 different mixer algorithms, and they are named using the following constants

```
macro
: mix:silence 0 ;
: mix:xmod    1 ;
: mix:reso    2 ;
: mix:osc1    3 ;
forth
```

`silence` produces a zero output, `xmod` will cross–modulate the 3 oscillators, `reso` uses the oscillators to modulate a single carrier with an envelope built from the 2 remaining oscillators, and `osc1` outputs a single oscillator for pure noise or square wave sound. Next we build some abstractions that modify the variable `synth` and the period registers.

```
: reso \ --
    mix:reso
    sync:0->1 bit
    sync:0->2 bit synth !

    _p0@
    _>> _dup _p2     \ half max reso
    _>>      _p1 ;   \ reso freq 4x
```

This function creates a configuration value to store in the `synth` variable. Here the word `bit \ byte bit -- byte+` is used to set an individual bit in a byte and return the updated byte. The constant `mix:reso` representing the resonant mixer algorithm is combined with some bits that set the sync configuration. Additionally, we also configure the oscilattor periods. Note that underscores represent words that operate on double size cells. We take the period value of OSC0, divide it by 2 using a right shift operator and store it in OSC2's period register, then divide it by 2 again and store that result in OSC1's period register. The result is a single word `reso` that will put the synth engine into a state that produces a sound with a clear formant, using an algorithm called *hard sync*. In `synth-control.f` there are some more words that change the synth config in a similar manner, producing some well known synth sounds.

```
  silence \ --          | no sound
  reso    \ --          | fake resonant / formant wave
  noise   \ --          | noise
  square  \ --          | square wave
  xmod2   \ --          | 2 OSC cross modulation
  xmod3   \ --          | 3 OSC cross modulation
  rxmod   \ --          | random cross modulation
  pwm     \ period --   | pulse width modulation
  _pwm    \ lo hi --    |   same for 16 bits
```

4 Hierarchical Time

Next to the 3 on–chip timers used to to implement the sound oscillators, the 4th timer in the PIC18 is used to drive a fixed rate oscillator that provides a time base for highlevel code. It is implemented by incrementing a 32–bit counter on each timer tick. The counter value is stored in 4 variables `tick0` to `tick3`. Now, have a look at this table which reflects binary increment from 0 to 7.

```
000
001
010
011
100
101
110
111
```

Notice that in each row, the bits change value with a period that is the *double* of that of the right neighbouring column. This counter is a cheap source of events we can sync to, spanning an enormous range of frequencies.

This is exploited in the `sync-tick` macro word, which takes a single argument indicating which bit in the tick timer it will synchronize on: it simply waits until that particular bit exhibits zero to one transition. The counter is updated at a frequency of 7812 Hz = 2MHz / 256, which means bit 0 has a frequency of 3906Hz. For each bit to the left, the frequency halves. From this set of frequencies we choose 2:

```
: wait-note    9 sync-tick ;  \ 7.8 Hz
: wait-control 4 sync-tick ;  \ 244 Hz
```

In the abstractions we're about the create, control rate is the rate at which timbre modulations take place, while note rate is the duration of a 1/16th note at 117 BPM. These rates are rather arbitrary, but correspond roughly to the two basic time scales that can be used to structure sound and music. The former is useful for creating non–stationary *sounds* while the latter is useful as a basic unit for *rhythm*. This fixed relation can be a limitation, especially for the note rate, but it severely simplifies the implementation

while still giving a useful abstration[6]. To create non-stationary sounds, we need to change stationary synth parameters at the proper time instances. A way to do this is to simply put a time synchronization word in a loop:

```
: siren
    begin
        wait-note square
        wait-note noise
    again ;
```

The important thing to see here is that there is is a loop with an alternating *sync* and *action* part. The sync word waits until a certain time instance, and the action part changes the state of the sound engine to produce a different sound. When you want to create words that have synchronization built in, where do you put the sync part? Before or after the action? This depends on how you want to combine them. We'll see in the next section that the best place is *inbetween*, since that leads to easier composition of hierarchical time scales because it avoids *shared* synchronization points.

5 Sound Generators

Let's turn up the abstraction notch a bit more. In this section macros are used to create new control flow operators.

The Sheep is a binary output monosynth. Because mixing of sounds in the conventional way is not possible, the only thing we can do is to make the sound vary over time. The main problem is *sequencing* of state change events at particular time instances. Let's simplify the problem by splitting it up into two parts, corresponding to the two time scales identified in the previous section. We're going to build a collection of *sound generators* that implement evolution at the control time scale and a *trigger controller* that implements changes at the note time scale.

We'll implement a sound generator a an *infinite loop* that produces an evolving non-stationary sound. At each time instance, there is only one sound

[6]This document contains a lot of examples that illustrate a common programming technique. Change the problem specification so the implementation becomes simpler. When requirements are fixed, this is not always possible. However, in internal representations (self-imposed interfaces) it usually is a good trick. For low-level programming, *bit twiddling* hacks are a good source of such simplifications.

generator active. The trigger controller activates different sound generators. The difference between the two is that a sound generator can be *started* and *stopped* by trigger controller, and that the rate of change of a sound generator is *higher* than that of a trigger controller.

This approach requires a form of multitasking, since there are two separate threads of control. There is an infinite loop that drives the generation of non-stationary sound, and a hierarchical control loop that can influence the behaviour of the infinite loop. The multitasking is implemented in the `synth-soundgen.f` file and uses functionality from `pic18/task.f`.

The interface to the sound generator mechanism consists of variable `sound` which contains a *code vector* pointing to the current loop code, and a word `bang` which restarts the loop pointed to by `sound`. A code vector is a variable that contains a pointer to a code.

The following code uses this to create a particular sound generator instance that produces an alternation of a square wave and a noise burst:

```
: sync-mod
    7 sync ;
: wobble
    sound -> begin
                square 50 p0 sync-mod
                noise  sync-mod
              again
```

Here we define a word `sync-mod` which is used to synchronize to a clock of about 2 Hz. This is similar to the definition of the word `wait-control` in the previous section. However, the word `sync` implements some additional behaviour compared to the word `sync-tick` used before. In addition to make the code in which it occurs wait for the next event, it also *switches* to the other of the two tasks.

The implementation of this is fairly straightforward, but can be hard to understand at first so I will not go into much detail[7]. It will suffice to understand only the abstract meaning of the `sync` word. It waits for

[7]The multitasking works by continually switching between two tasks that are both waiting for the next event, and resuming operation of the first one that is allowed to continue. This is a simple form of *cooperative multitasking*. Each task has its own execution context, which consists of a return stack, a data stack, an auxiliary stack, and a separate copy of the array registers pointing to RAM and Flash memory. See the PIC18 Forth manual for more information.

synchronization events in the task in which it is called, and takes care of all the magic necessary to have 2 tasks run on a single machine in the first place.

The word `->` consumes a variable which contains a vector, in this case the code vector used to store a pointer to the current sound generator loop code, and sets it to point to the code immediately following the arrow. The word `wobble` in which `->` occurs is then aborted immideately, which means the code behind it will not run as part of the execution of `wobble`, but it is only used to define the meaning of the variable passed to `->`. The word `wobble` merely sets the *value* of the variable `sound`; it only changes the state of the sound generator. The next time `bang` is executed, this state is collected from the variable `sound` and the loop wil start running. Note that the generator loop and the trigger controller are tied: the loop only runs as long as a trigger controller is active, because it is activated during the time the controller word is waiting for a next event. This is a simple trigger controller:

```
: note-sync
    9 sync ;
: notes \ n --
    wobble                \ make the wobble sound current
    bang                  \ restart current sound
    for note-sync next    \ wait for n notes
    silence ;             \ turn off the sound
```

Running `10 notes` will produce a wobbly sound for a duration of 10 notes. Note that a sound generator acts as a *virtual sample player*. From the viewpoint of the controller, the sound is simply turned on and off. From the viewpoint of the sound generator, it will loop forever. This abstraction separates concerns at the the note–level time scale from concerns at the time scale at which sound modulations happens.

6 Pattern Programs

Let's recall the tower of abstractions we've used uptil now. The stationary sound synth is abstracted through a number of configuration variables. On top of this we've built an abstraction that allows the hierarchical sequencing of note rate events that trigger sounds implemented by loops that modulate the stationary sound synth at modulation rate.

Instead of writing words which perform an entire piece in the way explained above, where all the parts have to be expressed *statically* beforehand, it might be interesting to add the ability to change the behaviour of words at run time. This is called *dynamic binding*. Note that the `->` mechanism introduced above is already an example of this.

In this section we will explore the use of *byte codes*. A byte code is a number which represents a certain behaviour. The idea is that the relationship between a number and its associated behaviour (word) is fixed, but the numbers themselves can be stored in variables to implement a *state machine*. A state machine is a machine which has a variable behaviour that depends on its current state.

In the Forth this is implememented using the `route` macro word. This word can be used to implement jump tables that associate a number to an array of words implementing behaviour[8]. The words in a jump table are separated by `.,` while the last word is terminated using `;`

```
: step \ n --
    route
      left ., right ., up ., down ;
```

The word `step` accepts a number which it maps to behaviour. In this case 0 is mapped to `left`, 1 to `right`, ... The end result is that we can write code that does something based on a `variable` that can be changed at run time. For example

```
variable direction
: go  direction @ step ;
```

Now `go` will perform one of 4 behaviours depending on the value of the variable `direction`.

[8]This is implemented by skipping a number of machine instructions. It so happens that a word followed by a semicolon is exactly one machine instruction wide. It is a jump to the code that implements that word. An "upside down" semicolon behaves the same, but indicates to the compiler that the following code is still accessible.

We now create a language for expressing rhythmic patterns on top of the abstractions created thusfar. Suppose the word `drum` executes some synth reconfiguration event that changes the `sound` vector. A way to build a pattern sequencer is to create a map from a variable `time` to behaviour. Note that an upside down semicolon all by itself is just a RETURN instruction and thus does nothing.

```
variable time
: a-sequence time @
    route
        drum ., drum ., drum ., drum .,
        drum .,      .,      .,      .,
        drum ., drum .,      ., drum .,
        drum .,      ., drum .,      ;
```

This implements a state machine `a-sequence`. Its behaviour depends on the state variable `time`.

For convenience, let's limit the size of the patterns to 16. One thing I neglected to mention is that you can easily jump off the end of a jump table if the number is too big. Solve this by performing an `#x0F and` operation after fetching the byte. In that case one never has to worry about the value being out of range, and incrementing time is just `time 1+!`, resulting in a looping pattern.

Now, let's introduce some macros. Note that a macro is just an abbreviation for a sequence of words. This makes it similar to a procedure word, however macros can implement behaviour that simple procedure words cannot. In general, you need to use macros to build abstractions on top of other macros. So to compose *control words*, which are implemented as macros, one needs to use macros. Control words change the normal sequential order of execution. In the following we will build abstractions around the macros `.,` and `route`[9].

```
variable  time
macro
: o       drum ., ;
: .       ., ;
: pattern time @ #x0F and route ;
forth
```

With these macro definitions, each occurence of `o` will be replaced with `drum .,` and each occurence of `.` with `.,` and similarly each occurance of `pattern` will be replaced `time @ #x0F and route`. This means the word `a-sequence` is equivalently defined as:

```
: a-sequence
    pattern
      \ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
        o  o  o  o  o  .  .  .  o  o  .  o  o  .  o  . ;
```

That's an ASCII art GUI for creating sequencer patterns! Note that what we did is to simply identify reusable structure in the code and name it appropriately. Wheter this name is then implemented as a macro or a procedure depends on what functionality it abstracts, but the mechanism is largely the same.

Now, to complete the story, let's implement the word `drum` so we can reuse the word `a-pattern` for different sounds. We'll use the same vector mechanism that was used to implement `sound` in a previous secion. What I didn't mention before is that vectors can be executed using the `invoke` word. The complete implementation of `drum` and the associated macro `o` is:

[9]It might help to have a look at the PIC18 Forth paper and see how basic control flow operations are implemented. The fact that these constructs are programmable is one of the most powerful elements of Forth in general.

```
vector drum
: do-drum  drum invoke ;
macro
: o        do-drum ., ;
forth

\ some words that change the drum sound
: snare    drum -> init-drum-sound ;
: hihat    drum -> init-hihat-sound ;
```

Excuting the `do-drum` word sets the variable `drum` to point to the code sequence `init-drum-sound`. The code sequence `snare a-sequence` would execute the time–variant word `a-sequence` defined earlier. If the variable `time` contains a state number that corresponds to a `o` in the pattern, the word `init-drum-sound` will be executed. Alternatively, if the pattern contains a `.` word at that place, this word won't execute.

What we just did is to parameterize the behaviour of `a-sequence` using the word `snare`. Similarly, the word `hihat` can be used to do the same.

7 Conclusion

The examples in this tutorial illustrate the construction of a DSL for building a sound synthesizer from almost nothing. Compared to ordinary high level programming, where the abstraction level one starts from is already quite high, this is simply barbaric.

However, it is my opinion that the level of abstraction at which one starts doesn't really make much of a difference. The trick to writing good software is to learn how to use the primitives and the composition mechanisms effectively. This is not central to Forth, but Forth allows to do this in a very elegant way.

I hope to show with this tutorial that all you really need to build a powerful DSL from almost nothing is a composition mechanism that is powerful. For concatenative languages, this composition mechanism is string substitution: a word can refer to a sequence of words. This can then be implemented in two ways. Procedure words will be substituted at program run time using the chip's procedure call abstraction, while macro words are substituted at program compile time.