The Sheep

TOM SCHOUTEN

November 1, 2007

Abstract

This document talks about The Sheep, a simple 1–bit synth implemented in Purrr. Its main purpose is to illustrate the use of Purrr (and more generally Forth) in the construction of a domain specific language.

1 Bottom Up Programming

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. This might sound scary, but in a language like Forth (or Lisp) it is actually quite straightforward: a more abstract language layer *looks the same* as the lower level one, it just carries more highlevel meaning.

A *language layer* is a merely collection of high level words, which in Purrr are *procedure* and *macro* words. The basic idea is to *hide* the implementation of these words from the programmer using them. These words comprise an *interface*: programming is made simpler because the programmer using the interface does not need to worry about how the details are implemented, and uses *only* the words provided in the interface. Such an interface is sometimes called an Application Programmer Interface or API. A language layer is sometimes called a *code library.*

What bottom up programming provides is 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 hope we have at all to write working softare.

It becomes interesting part is when the programmers at the two sides of the API, 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, once you start making *contracts* with yourself, programs suddonly becomes a lot easier to manage. One way of doing this is to make interfaces *explicit*.

In Purrr, it is not necessary to use explicit interfaces. This is both good and bad: it takes away the red tape of being *obligated* to always draw the line between two sides of an interface. This can be an impediment to early stage code evolution. However, when you have things worked out, it is usually a good idea to chop them in pieces: literally put them in different files, and force yourself to *only* use clearly defined interfaces.

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.

2 The Sheep Primitives

I start with the low level synth engine as a given. The API 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 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 all there is: at the lowest level, the synth *patch language* is merely a couple of configuration variables, and two words implementing an on/off switch. As a programmer, you do not have access to the internals of the synth engine beyond these words without changing the low level code in `synth-core.f` and recompiling.

This API is still quite low level. However, it does succeed in providing a very concise view of the synth algorithm. We get a simple way of programming a *synthesizer machine*, but in order to get there we give up the capability to perform arbitrary algorithms. By *abstracting* the problem domain (make some noise), we limit the number of possible solutions we can use, but greatly simplify the expression of a useful set of solutions. We give up full control but instead simplify the problem to the configuration of a *task* that runs in the *background*. The API provided is a set of configuration variables, and a simple on/off switch. The great challenge in programming is to find the right kind of abstractions by simplifying and generalizing the problem.

So let's move on to the 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. 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. 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. 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 Purrr, if a word starts with a *underscore* it will produce and/or consume *double* values. I.e. in the 8–bit Purrr18, this means two bytes to represent 16-bit values. For ease of use, there are 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
```

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

Similarly words can be defined to facilitate the interaction with the words `_p0`, `_p1` and`_p2` themselves. One extension uses a table internally to convert note and octave numbers to periods

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

note0  \ n --   | set OSC0, OSC1 and OSC0 to a frequency
note1  \ n --   | corresponding to a note in the current octave,
note2  \ n --   | counting from 0 -> C, 1 -> C#, ...
```

Of course, it is not only possible to write to the period registers. Some algorithms might find a need to update the current period instead of setting it, i.e. portamento. The following words retreive the double words for each period. It is also possible to read from the period variables directly. Because the ISR never changes these registers, it is safe to read the period registers directly. However, for symmetry it might be nicer to use the following words:

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

For the variable `synth` we do the same: define a couple of words that set individual bits in the synth algorithm. For example

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

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

To create a value to store in the `synth` variable, start with the mixer value. Then use the word `bit` to set individual bits in the word on the top of the stack. When done, store the variable. For this particular synth config, we take the period value of OSC0, divide it by 2 and store it in OSC2's period register, then divide it by 2 again and store that result in OSC1's period register. As mentioned before, there are 4 different mixer algorithms, and they are named using

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

In `synth-control.f` there are some more words that change the synth config in a similar manner:

```
: 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
```

3 Hierarchical Time

Next to the 3 oscillator, there is something else that involves oscillating things: the 4th timer in the PIC18 is used to drive a fixed rate oscillator. On each tick a 32–bit counter is incremented. The counter value is stored in 4 variables `tick0` to `tick3`.

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. The `sync-tick` word 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
```

Control rate is the rate at which timbre modulations could take place, while note rate is the duration of a 1/16th note at 117 BPM. This fixed relation between frequencies can be a limitation, especially for the note rate, but it severely simplifies the implementation. Note that it is possible to vary the frequency that drives the timer network as a whole.

So, to play with time, we need to change parameters at the proper time instances. A way to do this is to simply put a synchronization word in a loop:

```
: siren
    begin
        wait-note square  \ ...
        wait-note siren   \ ...
    again ;
```

The important thing to see here is that there is is a loop with an alternating *sync* and *action* part. 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.

4 Sound Generators

[EDIT: Explain this better: disentangle tasks, syncrhonization and vectors.]

The Sheep is a binary output monosynth. Because mixing of sounds in the conventional way is as good as impossible, the only thing we can do to make the sound vary over time. Let's simplify the problem by splitting it up into two parts. We're going to build a collection of *sound generators* and a *trigger controller*.

A sound generator is an *infinite loop* that produces an evolving sound synchronized to events from the hierarchical timer. At each instance, there is only one sound generator active. The trigger controller activates different sound generators, also based on the hierarchical time. The difference between the two is that a virtual sample can be *started* and *stopped* by trigger controller.

This approach requires multitasking, since there are two separate threads of control: an infinite loop that generates sound, and control loop that can change the current sound task. This multitasking is implemented in the `synth-soundgen.f` file and uses functionality from `pic18/task.f`. The interface to the sound generator player consists of variable `sound` which contains a vector, and a word `bang` which restarts the sound generator stored in `sound`. A vector is variable that contains a code address. The code pointer stored in `sound` is a loop that changes the synth configuration while synchronizing on hierarchical timer events. For example, a sound consisting of an alternation of a square wave and a noise birst can be implemented as:

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

The word `sync` is like the word `sync-tick` mentioned before, but instead of just waiting for an event, it also passes control back to the bang controller task, which in turn will pass control back to the sound generator task when it is done. The word `->` (arrow) consumes a variable which contains a vector, in this case the vector used to point to the current sound generator, and sets it to point to the code after the arrow. Note that when `wobble` is executed, the loop code following the arrow does not execute. The word `wobble` merely sets the *value* of the variable `sound`. The next time when `bang` is executed, the loop wil start running (as long as the trigger controller runs).

To run this sound generator, you need to drive it with a controller loop. A very simple one could be:

```
: note-sync
    9 sync
: notes \ n --  | number of notes to run synth
    wobble bang
    for note-sync next
    silence ;
```

Then running `10 notes` or, to avoid timeout notices, `10` start notes—, gives the wobbly sound. The idea is that whenever a `sync` operation runs, the other task is activated to see if it needs to update something, so as long as you don't execute any `sync` operations from the primary task, which is commanded by the console, the sound generator task will not run.

To sum up, a small note about tasks. 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 `a` and `f` registers which point to RAM and Flash memory respectively. Usually, the word `yield` switches between tasks: it invokes a scheduler that saves the task that gives up control (the one calling `yield`) and wakes up another one. In our case, `yield` is very simple: it just switches between tasks. Now, for the sound generator, the magic happens in the `sync` word: this word calls `yield` before checking if the synchronization condition is true, and looping in case it is not. The idea is that while one task is waiting for an event, it passes control to another task. This is called *cooperative multitasking*.

5 Pattern Programs

[EDIT: already explained vectors at this point, so just build on top of it]

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 some *dynamic binding*: the ability to change behaviour of words (names) at run time. Note that Purrr is intentionally a strictly static bottom up language: once you compile code, you cannot change its meaning. You can write new code *on top* of old code, but changing a lower level inadvertently means recompiling the program.

This is problematic. However, it is straightforward to introduce dynamic behaviour by using *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 staticly fixed, but the numbers themselves can be stored in variables and thus modifed. This effectively creates a *virtual machine*. The numbers can be seen as instructions of a machine which is defined by the mapping from numbers to words.

Let's make this a bit more concrete. Suppose we want to create a language for expressing time sequence patterns to use in the Sheep synth. The objective is to have something that looks like a list of words. One way to implement this is to use a multitasking engine, however in this case that would be overkill since there is a simpler approach using the `route` word. This word performs a map from numbers to code by implementing a jump table.

```
: somewhere \ n --
    route
      left ; right ; up ; down ;
```

The word `somewhere` accepts a number which it maps to behaviour. In this case 0 is mapped to `left`, 1 to `right`, ...What `route` actually does is to skip a number of machine instructions. It so happens that a word followed by a semicolon is exactly one machine instruction: a jump to the code that implements that word. So `route` indicates the start of a jump table. 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 @ somewhere ;
```

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

Let's use this to build a pattern sequencer. Suppose the word `drum` executes some synth reconfiguration event. A way to build a pattern sequencer is to create a map from a variable `time` to behaviour. Like this:

```
variable time
: wicked time @
    route
        drum ; drum ; drum ; drum ;
        drum ;      ;      ;      ;
        drum ; drum ;      ; drum ;
        drum ;      ; drum ;      ;
```

Note that a semicolon all by itself is just a RETURN instruction and thus does nothing. Let's clean that up a little. This implements a word `wicked` with a time varying behaviour, assuming the variable `time` has the current time stored.

In general, when the words in a `route` table change the value of the variable on which is dispatch, this pattern is called a *state machine*. Here the state needs to be set by the caller of `wicked`, and will be a simple increment.

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 which is not instantiated as machine code, but always inlined. Macros can be used to factor out code that can not be abstracted in a procedure definition. In the following code the words `exit` and `route` interfere with straight line procedure control flow, and thus have to be abstracted in macros. Also note that `exit` is an alias for the semicolon word that does not have the special meaning of terminating a macro definition.

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

This leads to the definition of a pattern as:

```
: wicked
    pattern
        o o o o
        o . . .
        o o . o
        o . o .
```

That's an ASCII art GUI for creating wicked sequencer patterns!

Let's make this a bit more useful. The `drum` word above was only vaguely defined. Is it possible to change the meaning of that word at runtime? Sure, as long as we find a way to store its behaviour in RAM. We could create a byte code map for `drum` as we did for the `somewhere` word above, but let's try something different.

If you want to store the representative number of a small collection of words in a lot of different places, byte codes are a good idea. If you want to modify the behaviour of a small number of plug–in words, *vectors* are a better match. In Purrr a vector is represented by 2 bytes, making the variable a `2variable`. Vectors can be set using the `->` word. The code following the arrow is what the vector will point to when the word in which the arrow occurs is executed. Vectors can be run using the `invoke` word

```
2variable drum

: drum     drum -> stuff init-drum ;
: hihat    drum -> other-stuff init-hihat ;

: do-drum  drum invoke ;
macro
: o        do-drum exit ;
forth
```

Excuting the `drum` word would merely set the variable `drum` to point to the code sequence `stuff init-drum`. The sequence `drum wicked` would execute the time–variant word `wiked` defined earlier with `o` bound to the code following the arrow in the `drum` word. The the effect of `drum` is to change the *dynamic environment* in which the word `wicked` runs. This way we can combine two program elements: the current instrument, and the pattern used.