ATMEL PIN MACROS

Introduction

When writing simple digital I/O code on Atmel microcontrollers, the standard approach is to access the `PIN`, `PORT` and `DDR` registers directly.

Here is an example on Stack Overflow. As a concrete example, we might consider code like this:

```
#include <avr/io.h>
...
DDRB  |= _BV(PORTB2)
...
PORTB |= _BV(PORTB2)
...
PORTB &= ~(_BV(PORTB2))
```

The `_BV` macro here is provided by avr-libc and just wraps the necessary shifts.

That code works, but it's a pain to manage: typically in application code we want to refer to the pins in the language of the application, abstracting away from the pin which happens to perform the role.

Having chosen a pin, on the AVR mega, we need to keep track of both which 8-bit port is used, and which bit within that port. It would nice if we could say things like:

```
#define LIGHT_DDR  PORTB
#define LIGHT_PORT PORTB
#define LIGHT_BIT  PORTB2
...
LIGHT_DDR  |= _BV(LIGHT_BIT)
LIGHT_PORT |= _BV(LIGHT_BIT)
```

Nicer still might be to hide the bitwise operations entirely:

```
#include "mjo-pin.h"
...
SINGLE_PIN(light, B, 2)
...
light_init_write();
light_set_high();
```

Behind the scenes, mjo-pin.h defines the `SINGLE_PIN` macro which expands the name and definition into a series of inline functions. For example, `light_set_high()`:

```
static inline light_set_high(void) __attribute__((always_inline));
static inline light_set_high(void) { PORTB |= _BV(PORTB2); }
```

The functions compile down to compact machine code, so inlining them is both faster *and* smaller. As a bonus, if a function isn't used, it won't be included.

No composability

There are disadvantages to this approach though: because there's a 1:1 correspondence between pins and functions, you can't easily parameterize things, or clone them.

For example, in Arduino land, all the I/O pins have a single number and the I/O routines accept that number at runtime. So you can write code like this:

```
uint8_t pins[] = { 1,2,3,5,6,7,0 };
for(int i = 0; pins[i] != 0; i++)
  {
    digitalWrite(pins[i], HIGH);
  }
```

Most of the controller libraries also accept pin numbers in their constructors. Here's a stepper example:

```
#define STEPS 100
...
Stepper stepper(STEPS, 8, 9, 10, 11);
```

The natural way to write a `SIMPLE_PIN` stepper controller would be to define pins outside of the object, generating e.g. `stepA_set_high()` functions, and then call those functions explicitly in the controller code. This makes it hard to instantiate two controllers in the same code.

No fusability

Another disadvantage is that there's no scope for fusing separate operations. For example, consider:

```
SINGLE_PIN(front_led, B, 2);
SINGLE_PIN(back_led,  B, 3);
...
front_led_set_write();
back_led_set_write();
```

which will compile down to:

```
DDRB |= _BV(PORTB2);
DDRB |= _BV(PORTB3);
```

In an ideal world, we'd replace this with:

```
DDRB |= (_BV(PORTB2) | _BV(PORTB3))
```

Sometimes this is just a matter of efficiency, but in some cases it's important that the two bits are updated simultaneously.

Portability

On the plus side though, this scheme does hide all the AVR mega specific stuff in the macro definition. You could imagine writing analogous macros for the ARM say, and the only change to the application code would be to define the pins differently.

Or, if you were really wedded to the Arduino API you could target that. Given:

```
SINGLE_PIN(light, 13);
```

the macro would give us:

```
static inline light_set_high(void) __attribute__((always_inline));
static inline light_set_high(void) { digitalWrite(13, HIGH); }
```

Source: http://www.mjoldfield.com/atelier/2014/05/pin-macros.html