HwCpp Primer

- Introduction
- Blink a led
- Kitt
- More fun with LEDs

Introduction

HwCpp is a library for writing micro-controller applications. This document provides a gentle introduction to using HwCpp. Basic C++ and hardware knowledge is assumed, but nothing too advanced.

Blink a led

Blinking a LED is the "Hello world!" equivalent for micro-controllers, so let's start with that.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::timing;
using led    = target::led;

int main(){
   led::init();
   timing::init();

   for(;;){
      led::set( 1 );
      timing::ms< 200 >::wait();
      led::set( 0 );
      timing::ms< 200 >::wait();
   }
}
```

A typical HwCpp application is a single main.cpp file that includes and combines the parts of the application.

```
#include "hwcpp.hpp"
```

The default way to specify which target is used is to put that information in the makefile, which passes it to the compiler as a command-line macro. The application just includes "hwcpp.hpp", which includes the appropriate target-specific parts of HwCpp. Alternatively, the application can include the target-specific HwCpp file directly.

```
using target = hwcpp::target<>;
```

All hwcpp stuff is inside the namespace hwcpp. This includes the target<> template, which is, through makefile/macro/hwcpp.hpp magic, the target micro-controller or board you are building your application for. By default, the target will use the highest clock speed possible, but in some cases you can specify a slower clock speed as template parameter. In this case we don't, so the target will run full-speed.

```
using timing = target::timing;
```

Most things in HwCpp are classes, not objects, so :: is used to select a thing within another thing. The line above selects the default timing service offered by our target as the timing we will use.

```
using led    = target::led;
```

This blinky application is written for a target board that has an on board default LED. This is the case for the currently supported targets boards (Arduino Uno, Arduino Due, Blue Pill).

```
led::init();
timing::init();
```

Most 'things' in HwCpp are classes, not objects, but the play roughly the same role objects do in standard OO style applications. Objects are initialized by their constructors, HwCpp classes, which are called Compile Time Objects (cto's), are initialized by calling their ::init() function. *All* cto's must be initialized in this way before they are used. Here we initialize the tow cto's we will use: the timing and the LED.

```
for(;;){
   . . .
}
```

A micro-controller application has no Operating System to return to, hence it typically contains a never-ending loop. We can probably bike-shed forever about the best way to write such a loop, but I prefer for(;;) so that is what you will see.

```
led::set( 1 );
. . .
led::set( 0 );
```

The LED cto has a ::set() function that accepts a single bool value and makes its pin high or low according to the parameter.

```
timing::ms< 200 >::wait();
```

The timing cto has a set of macro's (ns<>, us<>, ms<>, s<>) that are used to specify a duration of the specified amount of time. Such a duration cto has a ::wait() function that will wait for the appropriate amount of time.

The point of a library is to make writing applications easier, so HwCpp has a blink<> function template that can be used to write a blinky. It requires the

pin that must be blinked, and the duration of the on and off periods.

The HwCpp convention is that calling ::init() on cto's is the duty of the code that uses the cto's. Hence in this case it is the duty of the hwcpp::blink<> template to ::init() the led and duration, so we don't have to write those lines. And because the timing is now mentioned only once the using... line for that can be omitted.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;

int main(){
   hwcpp::blink<
      target::led,
      target::timing::ms< 200 >
   >();
}
```

The using... line for the target could be omitted too, but the target is mentioned twice, so in my taste omitting that line produces a blinky that is shorter, but slightly less pleasing to the eye.

```
#include "hwcpp.hpp"

int main(){
   hwcpp::blink<
      hwcpp::target<>::led,
      hwcpp::target<>::timing::ms< 200 >
   >();
}
```

Kitt

After blinking a single LED, the next step is to do something with a bunch of LEDs. The Kitt display (one LED back-and-forth, from the Knightrider series) is the standard example for this.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::port_out<
   target::d8,
   target::d9,
   target::d10,
   target::d11,
   target::d12,
```

```
   target::d13
>;

int main(){
   hwcpp::kitt< pins, timing::ms< 50 > >();
}
```

The supported target boards don't have a string of LEDs, so instead the pins that connect to the LEDs are specified. This application is for the Arduino Uno target, hence the Arduino pin names are used. (Alternatively, the pin names of the atMega328 chip could be used.) I used six pins are are conveniently located next to a ground pin.

```
using pins = hwcpp::port_out<
   target::d8,
   target::d9,
   target::d10,
   target::d11,
   target::d12,
   target::d13
>;
```

The 6 pins are combined into a port_out. A port is an (ordered) bundle of pins, and 'out' indicates that the port can be used only as output.

```
   hwcpp::kitt< pins, timing::ms< 50 > >();
```

We could write the kitt functionality ourselves, but HwCpp has a function template for that, which requires a port and a duration. We pass those parameters, call the function, and kitt is alive.

More fun with LEDs

Blinking can be made more interesting by blinking more than just a single LED. A bunch of pins can be combined into a something that walks and quacks like a single (output) pin with the hwcpp::fanout<> template. To blink the six LEDs of the kitt example in usinson, all we need is to combine them into a single 'pin', and pass that pin to the blink function.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::fanout<
   target::d8,
   target::d9,
   target::d10,
   target::d11,
   target::d12,
```

```
   target::d13
>;

int main(){
   hwcpp::blink< pins, timing::ms< 200 > >();
}
```

The hwcpp::invert<> template can be used to create a pin that inverses the behavior of the pin it decorates. If we invert the first three pins this way before passing them to fanout, the LEDs alternate left-right.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::fanout<
   hwcpp::invert< target::d8 >,
   hwcpp::invert< target::d9 >,
   hwcpp::invert< target::d10 >,
   target::d11,
   target::d12,
   target::d13
>;

int main(){
   hwcpp::blink< pins, timing::ms< 200 > >();
}
```

A different (but totally equivalent) way to get this effect is to first combine the two groups of three LEDs, then invert one, and finally combine the two groups.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::fanout<
   hwcpp::invert< hwcpp::fanout<
      target::d8,
      target::d9,
      target::d10 > >,
   hwcpp::fanout<
      target::d11,
      target::d12,
      target::d13 >
>;
```

```
int main(){
   hwcpp::blink< pins, timing::ms< 200 > >();
}
```

A simple variation alternates between the even and odd LEDs.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::fanout<
   hwcpp::invert< target::d8 >,
   target::d9,
   hwcpp::invert< target::d10 >,
   target::d11,
   hwcpp::invert< target::d12 >,
   target::d13
>;

int main(){
   hwcpp::blink< pins, timing::ms< 200 > >();
}
```

Another nice pattern is the inside-to-outside. The base for this is the walk<> function, which is like Kitt, but only forward.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::port_out<
   target::d8,
   target::d9,
   target::d10,
   target::d11,
   target::d12,
   target::d13
>;

int main(){
   hwcpp::walk< pins, timing::ms< 50 > >();
}
```

If you don't like the direction in which the pattern walks, you could of course change the order in which the pins are mentioned in the port_out constructor, but it is easier to use hwcpp::mirror<>.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::port_out<
   target::d8,
   target::d9,
   target::d10,
   target::d11,
   target::d12,
   target::d13
>;

int main(){
   hwcpp::walk< hwcpp::mirror< pins >, timing::ms< 50 > >();
}
```

To get inside-to-outside, we create two ports of three LEDs, one in the reverse order and the other in the normal order. These two ports are combined by fanout<> to get a single port. Running walk<> on this port creates the intended effect.

=> fanout doesn't work yet for ports

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::pfanout<
   hwcpp::port_out<
      target::d10,
      target::d9,
      target::d8 >,
   hwcpp::port_out<
      target::d11,
      target::d12,
      target::d13 >
>;

int main(){
   hwcpp::walk< pins, timing::ms< 200 > >();
}
```

An alternative is to create the two sub-ports both in the standard pin order, but apply mirror<> to one of them before the two ports are combined by fanout<>.

```
#include "hwcpp.hpp"

using target = hwcpp::target<>;
using timing = target::waiting;

using pins = hwcpp::pfanout<
   hwcpp::mirror< hwcpp::port_out<
      target::d8,
      target::d9,
      target::d10 > >,
   hwcpp::port_out<
      target::d11,
      target::d12,
      target::d13 >
>;

int main(){
   hwcpp::walk< pins, timing::ms< 200 > >();
}
```

- add dummy pins