HwCpp documentation

HwCpp (Hardware-C++) is a C++ close-to-the-hardware library for writing micro-controller applications. It uses modern C++ features (C++17 and concepts) to enable efficient code re-use. It relies heavily on unicorns.

The typical blink-a-LED application using HwCpp is:

#include "hwcpp.hpp"

using target = hwcpp::target<>;

int main(){

hwcpp::blink< target::led >();

}

(This assumes that the target has a default LED, which is the case for the usual suspects like the Arduino Uno, Arduino Due and Blue Pill boards.)

Conventions

Get() and set() operations should be idempotent: a get() operation has no observable side-effects, and in the absence of state-changing effects subsequent get() operations will return the same value as the first. After a set( x ) call, absence of state-changing effects, subsequent set( x ) calls have no observable side-effects.

Classes (in which all functions and variables are static) are used to represent static (compile-time known) objects. Hence :: instead of . is used to access functions and variables within such objects. 🡺 name for such things?

Initialization of static objects is done by calling their init() functions. This is the responsibility of the code that actually uses such objects.

Pins

A pin represents an output, input, input-output or open-collector pin, either on the target chip or on some other location that can be controlled by the target chip.

An output pin can drive an external component, for instance a LED. An output pin can be set to high (power voltage level) by set( true ), or to low (ground level) by set( false ). The initial level of an output pin is target-dependent.

An input pin can read the logic level on the pin, presumably set by external components, for instance a switch and a resistor. The get() function of an output pin will return true when the pin is high (at power voltage level), or false when it is low (ground level).

An input-output pin can be used either as an input pin or as an output pin, as determined by the last direction\_set( d ) call, where d can be either direction::input or direction::output. The output level of an input-output pin directly after a direction\_set(direction::output) call is target-dependent.

An open-collector pin supports both the output and input pin operations (set and get). Electrically, an open-collector pin will pull its output low (connect it to ground) after a set( false ) call, but will leave it floating after a set( true ) call. Such a pin is commonly used with a pull-up resistor.

The pin\_\*\_dummy classes provide pin implementations that do nothing, which can for instance be useful as a placeholders.

The pin\_\*\_value classes provide pin implementations that contain a variable value that reflects the pin value.

Beside the normal set() and direction\_set() functions there are the \*\_direct and \*\_buffered versions. The direct versions take direct effect, the buffered versions can delay their effect until a subsequent flush() or direction flush() call on the pin. Likewise, a get\_buffered() call can return a value that was read earlier, but no earlier than the latest refresh() call on the pin. By default, the normal operations (get(), set(), direction\_set()) are direct (unbuffered) and the flush() and refresh() calls are no-ops. The buffer<> decorator can be used to change the behaviour of those operations to buffered, which is appropriate in situations where multiple (possibly remote) pins are read or updated in an unspecified order. An example of such pins are the pins provided by I/O extender chips, like a PCF8547A.

Pin decorators

The pin\_out<>, pin\_in<>, pin\_in\_out<> and pin\_oc<> decorators change the behaviour to that of a pin conform their name. This serves both as a check that a pin provided by a user can be used as intended, and as a conversion to the intended behaviour.

The buffered<> decorator changes the behaviour of the normal operations to buffered.

The invert<> decorator inverts the value read from or written to a pin.

Arduino Due

The Arduino Due HAL provides static classes for the chip pins according to their Arduino names (a0 .. a11, d0 .. d53, dac0, dac1, cantx, canrx, scl, sda, sca1, sca1, tx, rx, led, sck, miso, mosi, cs0, cs1). Note that some of these names are aliased.

Arduino Uno

<idem>

Blue pill

<idem>

Gcc toolchains

I used the following gcc builds:

* For AVR8 : <http://blog.zakkemble.co.uk/avr-gcc-builds/>
* For Cortex : <http://gnutoolchains.com/arm-eabi/>
* For (windows) native : <https://mingw-w64.org/doku.php/download/mingw-builds>