**HWCPP: a C++ library for**

**close-to-the-hardware**

**programming**



**Manual**

|  |  |
| --- | --- |
|  | Wouter van Ooijen  wouter@voti.nl  2015  Version 0.1, last modified 2015-06-25  <http://www.voti.nl/hwcpp> |

# Summary

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.

# Table of Contents

[Summary 1](#_Toc422918310)

[Table of Contents 2](#_Toc422918311)

[1. Introduction 4](#_Toc422918312)

[2. Library architecture and mechanisms 5](#_Toc422918313)

[2.1. Interfaces 5](#_Toc422918314)

[2.2. Objects 6](#_Toc422918315)

[2.3. Wrappers 6](#_Toc422918316)

[2.4. Optimization 7](#_Toc422918317)

[3. Timing 9](#_Toc422918318)

[3.1. Duration ADT 10](#_Toc422918319)

[3.2. Frequency ADT 10](#_Toc422918320)

[3.3. Moment ADT 11](#_Toc422918321)

[3.4. Delay service 12](#_Toc422918322)

[3.5. Clock service 12](#_Toc422918323)

[3.6. Scheduler service 12](#_Toc422918324)

[4. I/O pins and ports 13](#_Toc422918325)

[4.1. pin\_\* 13](#_Toc422918326)

[4.2. pin\_pullup, pin\_pulldown 15](#_Toc422918327)

[4.3. pin\_ad, pin\_da 15](#_Toc422918328)

[4.4. pin\_\*\_from 16](#_Toc422918329)

[4.5. port\_\* 16](#_Toc422918330)

[4.6. port\_\*\_from 18](#_Toc422918331)

[4.7. invert 18](#_Toc422918332)

[4.8. mirror 19](#_Toc422918333)

[4.9. dummies 19](#_Toc422918334)

[5. Graphics 20](#_Toc422918335)

[6. External chips 21](#_Toc422918336)

[6.1. I2C 21](#_Toc422918337)

[6.2. SPI 21](#_Toc422918338)

[6.3. D1W 21](#_Toc422918339)

[6.4. ADS7843 21](#_Toc422918340)

[6.5. DS18x20 21](#_Toc422918341)

[6.6. HC595 21](#_Toc422918342)

[6.7. HD44780 21](#_Toc422918343)

[6.8. MCP23xxx 21](#_Toc422918344)

[6.9. PCF8574(A) 21](#_Toc422918345)

[6.10. PCF8591 21](#_Toc422918346)

[6.11. PCD8544 21](#_Toc422918347)

[7. Miscellaneous 22](#_Toc422918348)

[7.1. default\_baudrate 22](#_Toc422918349)

[7.1. HWCPP\_INLINE 22](#_Toc422918350)

[7.2. Interface 22](#_Toc422918351)

[8. Naming conventions 23](#_Toc422918352)

[8.1. \*\_from 23](#_Toc422918353)

[8.2. get, set 23](#_Toc422918354)

[8.3. has\_\* 23](#_Toc422918355)

[8.4. interface\_type 23](#_Toc422918356)

[9. References 24](#_Toc422918357)

[10. ToDo 25](#_Toc422918358)

# Introduction

https://xkcd.com/license.html

====================== this whole document is work-in-progress ========================

HwCpp (HardWare library for C++) is a library for efficient and re-usable programming of (small) micro-controllers. It contains abstractions and implementations of for instance GPIO pins, timing (including a cooperative multitasking scheduler), and interfaces to external hardware like SPI and I2C busses, IO extenders, A/D converters, and LCDs. The library is provided under the Boost license, which basically means that you can do everything you want with this software, except that when you re-distribute the source, it must be under that same license.

The library aims to enable target-agnostic programming, and at the same time to be as efficient as C code written for a specific situation. To this end hwcpp makes extensive use of C++ 17 features, compile-time polymorphism, whole-program (link-time) optimization, and some gcc-specific extensions[[1]](#footnote-1), hence a recent gcc compiler is required (gcc 7.2 and later should work, clang will probably work too).

Hwcpp is not a complete development environment, but it can be used with bmptk, which provides a simple make-based build and download environment.

For the situations hwcpp is intended for, some C++ features are often inappropriate. Hence the hwcpp obeys some restrictions:

* No dynamic (heap) memory is used.
* No exceptions are used.
* No RTTI is used.
* No floating-point operations are used directly (but a hwcpp template can be instantiated with a floating point type)

Additionally, to suit my personal taste:

* No global objects that require run-time initialization are used.
* With very few exceptions, the library code compiles without warnings at the highest warning level.

Note that these limitations are *met* by hwcpp; this does not imply that an application that uses hwcpp must also meet these limitations.

Hwcpp is 'work in progress'. The latest version can be found at [github.com/wovo/hwcpp](http://github.com/wovo/hwcpp). If you somehow found and used hwcpp I am very interested to hear your experiences and comments.

The comics in this document are from [www.xkcd.com](http://www.xkcd.com), a definite must-see for everyone involved in computer programming, or otherwise thinks (too) logically.

Wouter van Ooijen (wouter@voti.nl)

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

#include "hwcpp.hpp"

using target = hwcpp::target<>;

using timing = target::waiting;

int main(){

hwcpp::blink< target::led, timing::ms< 200 > >();

}

(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>

# Library architecture and mechanisms



This section explains the terms and mechanisms used in the library.

## Interfaces

The library defines a number of interfaces. These interfaces are defined as abstract classes. An abstract class defines a number of pure virtual functions that, in standard OO fashion, must be implemented by each concrete class that implements the interface. A typical interface is shown below: pin\_in\_out abstracts a pin that can be used (configured) by the system either as input or as output.

struct pin\_in\_out {

typedef void has\_pin\_in\_out;

typedef pin\_in\_out interface\_type;

virtual void init() = 0;

virtual void direction\_set\_input() = 0;

virtual void direction\_set\_output() = 0;

virtual bool get() = 0;

virtual void set( bool value ) = 0;

};

Each interface X has a void member type has\_X, which can be used to select the appropriate template specializations when an object that implements the interface is passed as a parameter, and to produce an appropriate error message when an object is passed that the callee can’t accept (using the HWCPP\_REQUIRE\_INTERFACE() macro).

Each primary interface has a member type interface\_type that is the type of the interface. This type is used to by the object() function to cast an object of a derived type to its base type. This is used when knowledge of the actual (derived) type must be erased (using the interface() function).

Primary interfaces are mutually exclusive: an object can implement only one primary interface. In addition an object can implement a number of secondary interfaces.

Each primary interface has an init() function, which must be called before any other function of the interface is used.[[2]](#footnote-2) Likewise, a secondary interface X can have an X\_init() function that must be called before any of its functions is used.

## Objects

The library provides basic objects that implement some of its basic interfaces for a number of target micro-controllers. Each such object is a distinct type. An example is shown below: the gpio\_0\_4 and gpio\_0\_5 objects implement the pin\_oc interface on a LPC1114 micro-controller. Note that the pin and port numbers are passed to the constructor as template arguments, hence the two objects have different types.

struct lpc1114 {

template< int port, int pin >

struct pin\_oc final : public hwcpp::pin\_oc {

void init() override { . . . }

void set( bool x ) override { . . .}

bool get() override { . . . }

};

};

struct lpc1114fn28 {

. . .

auto gpio\_0\_4 = lpc1114::pin\_oc< 0, 4 >();

auto gpio\_0\_5 = lpc1114::pin\_oc< 0, 5 >();

}

An interface implementation publicly inherits the interface definition (in this case hwcpp:pin\_oc), and implements the required virtual functions. An interface implementation itself not meant to be sub-classed[[3]](#footnote-3), hence it is marked as final.

## Wrappers

The bulk of the library is coded as wrappers. A wrapper is a function template that takes one or more objects (that each implement an interface), and returns an object that implements an interface. This interface can be the same as one of the argument interfaces or a different one.[[4]](#footnote-4)

The class of the object returned by a wrapper inherits from (and hence implements) one of the libraries interfaces. A wrapper never inherits from one of its arguments: it uses delegation rather than inheritance.

Wrappers that are called X\_from take some arguments, and return an object that implements interface X. Such wrappers are often used to either convert an argument objects into a suitable form, or give a decent compile-time error message if this is not possible.

Run-time duck typing[[5]](#footnote-5) is used by many interpreted ‘scripting’ languages. It implies that a piece of code doesn’t care what type of object it gets passed to work on, as long as that object supports all the operations it wants to do on it during execution. Note *during execution*: if an execution path that is never activated would attempt an operation that is not supported by the object, no one will ever know.

Most hwcpp wrappers use compile-time duck typing: they are templatized on the type of each parameter, hence any type is acceptable, provided that it provides all the operations that the body of the wrapper does on the object. To avoid deep error messages, a wrapper will start with a check that it gets a parameter that is suitable. This is often realized as a conversion of the argument to a more suitable type, using a \*\_from() wrapper.

The blink function shows below is not a wrapper in the normal sense (because it does something instead of returning something), but is shows the way a wrapper accepts an argument. The type of the p argument is a template parameter, hence any (non-const) object is acceptable. The first thing done inside the function is to pass the p to the pin\_out\_from wrapper, which either converts it to a pin\_out (so blink() does not have to bother with for instance setting the direction when it gets passed a pin\_in\_out), or fails with a suitable compile time error message.

template< typename Pin >

void function blink( Pin & p ){

auto pin = pin\_out\_from( p );

. . .

}

## Optimization

Hwcpp objects can be passed as classic OO objects that are used via their virtual functions, but in most cases it preferable to use each object as a distinct type, which causes the compiler to instantiate and optimize the callee for the use of the specific object. This mechanism is essential to get compact and fast code.

The example below shows how a blink function accepts an object as a distinct type. Because all pin objects are unique types, each call to this blink function (with a different pin object) instantiates the template for that object type, which enables the compiler to optimize it for that type, which (with the appropriate compiler settings) eliminates the virtual calls.

template< typename Pin >

void function blink( Pin & p ){

auto pin = pin\_out\_from( p );

. . .

}

// both calls are instantiated and optimized for their pin

blink( target.gpio\_0\_5 );

blink( target.gpio\_0\_6 );

When a function is large and it is called with objects of many different type, a separate instantiation for each type can lead to code bloat. This can be avoided by erasing the type of the passed object down to its interface type, which results in one instantiation of the function being created per interface type. The template function interface() does this, by casting its argument to its interface\_type. This mechanism puts the choice between per-argument instantiation and optimization, or one-function-to-serve-them-all, on the user.

// instantiated once for the interface type of gpio\_0\_5/gpio\_0\_6 (pin\_oc)

// and hence shared by all other pins that are passed as pin\_oc

blink( interface( target.gpio\_0\_5 ));

blink( interface( target.gpio\_0\_6 ));

Functions that are not expected to benefit from being instantiated separately for each argument can be written in the classical way, accepting an object of the interface type by reference. A disadvantage is that such a function accepts only the exact type.

void function foo( pin\_in\_out & pin ){ . . . }

// OK with or without wrapper: gpio\_0\_0 is a pin\_in\_out

foo( target.gpio\_0\_0 );

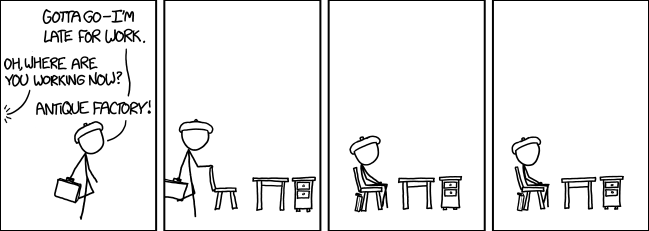
foo( pin\_in\_out\_from( target.gpio\_0\_0 ));

// wrapper needed: gpio\_0\_5 is a pin\_oc

foo( pin\_in\_out\_from( target.gpio\_0\_5 ));

A disadvantage is that such a function accepts only the exact type.

# Timing



This section explains how the library handles time (relative and absolute) and frequency values, and timing (waiting and scheduling).

Relative time is represented by the duration ADT[[6]](#footnote-6), absolute time by the moment ADT, and frequency by the frequency ADT. Time is built into the library because waiting is an essential part of embedded applications, frequency because it is needed to configure a target for a specific clock speed.

Time-related values are stored as long long (signed 64 bit int). [[7]](#footnote-7) Duration and moment are in units of a nanosecond, frequency in units of a mHz. Hence the maximum duration or moment after startup that can be expressed is 292 year.[[8]](#footnote-8)

Each time-related type has an associated set of subtypes that are used to express a compile-time literal value as a unique type. Such subtypes make it possible to force a wrapper that gets such an argument, to be optimized for its specific value. This can be especially important for very small durations, for which a delay() call can be implemented as a few inline instructions instead of a function call.

Timing is provided by a timing service, of which there are three versions: delay, clock, and scheduler. It is up to a target implementation which service(s) it supports and with which accuracy and resolution.

Code, especially library code, should accept a timing service as an argument and be written using only services that provided by the most primitive acceptable timing service. For instance, code that interfaces to an external chip typically must satisfy only some minimum timing requirements; hence it should use only the wait() service as provided by delay.

## Duration ADT

The duration ADT represents an amount of time (a time interval). The ’factory’ functions, member functions and operators related to this type are show in the next table. A duration value is constructed by a call to the ns(), us(), ms() or s() functions. The const\_ns<>(), const\_us<>(),const\_ms<>() and const\_s<>() template functions taken the amount as template argument and create, for each amount, a duration literal: an object of a unique type, derived from duration.

|  |  |  |  |
| --- | --- | --- | --- |
| Duration | | | |
| ’factory’ function | return type | Meaning | |
| ns( x ) | duration | a duration of x nanoseconds | |
| us( x ) | duration | a duration of x microseconds | |
| ms( x ) | duration | a duration of x milliseconds | |
| s( x ) | duration | a duration of x seconds | |
| ns\_literal< x >() | unique subtype of duration | a ’duration’ of x nanoseconds | |
| us\_literal< x >() | unique subtype of duration | a ’duration’ of x microseconds | |
| ms\_literal< x >() | unique subtype of duration | a ’duration’ of x milliseconds | |
| s\_literal< x >() | unique subtype of duration | a ’duration’ of x seconds | |
| member function | return type | meaning | |
| d.ns() | long long | the duration d expressed in nanoseconds | |
| d.us() | long long | the duration d expressed in microseconds | |
| d.ms() | long long | the duration d expressed in milliseconds | |
| d.s() | long long | the duration d expressed in seconds | |
| operator | left argument | right argument | result |
| + | none | duration | duration |
| - | none | duration | duration |
| + | duration | duration | duration |
| - | duration | duration | duration |
| \* | duration | int, long long | duration |
| \* | int, long long | duration | duration |
| / | duration | int, long long | duration |
| / | duration | duration | long long |
| ==, !=, >, <, >=, <= | duration | duration | bool |

## Frequency ADT

The frequency ADT is used to store and manipulate values that represent a frequency. A frequency is stored as a 1/1000s of a Hz. A frequency value is constructed by a call to the mHz(), Hz(), kHz(), MHz() or GHz() functions. The const\_ns<>(), const\_us<>(),const\_ms<>() and const\_s<>() template functions taken the amount as template argument and create, for each amount, a duration literal: an object of a unique type, derived from duration.

|  |  |  |  |
| --- | --- | --- | --- |
| frequency | | | |
| ’factory’ function | return type | meaning | |
| mHz( x ) | frequency | a frequency of x mHz | |
| Hz( x ) | frequency | a frequency of x Hz | |
| kHz( x ) | frequency | a frequency of x kHz | |
| MHz( x ) | frequency | a frequency of x MHz | |
| GHz( x ) | frequency | a frequency of x GHz | |
| mHz\_literal< x >() | unique subtype of frequency | a ’frequency’ of x mHz | |
| Hz\_literal < x >() | unique subtype of frequency | a ’frequency’ of x Hz | |
| kHz\_literal < x >() | unique subtype of frequency | a ’frequency’ of x kHz | |
| MHz\_literal < x >() | unique subtype of frequency | a ’frequency’ of x MHz | |
| GHz\_literal < x >() | unique subtype of frequency | a ’frequency’ of x GHz | |
| member function | return type | meaning | |
| d.mHz() | long long | the frequency d expressed in mHz | |
| d.Hz() | long long | the frequency d expressed in Hz | |
| d.kHz() | long long | the frequency d expressed in kHz | |
| d.MHz() | long long | the frequency d expressed in MHz | |
| d.GHz() | long long | the frequency d expressed in GHz | |
| d.period() | period[[9]](#footnote-9) | the period of the frequency | |
| operator | left argument | right argument | result |
| + | none | frequency | frequency |
| - | none | frequency | frequency |
| + | frequency | frequency | frequency |
| - | frequency | frequency | frequency |
| \* | frequency | int, long long | frequency |
| \* | int, long long | frequency | frequency |
| / | frequency | int, long long | frequency |
| / | frequency | frequency | long long |
| / | long long | period | frequency |
| ==, !=,  >, <, >=, <= | frequency | frequency | bool |

Like for frequencies, there are functions that return a frequency, and function templates that, for each frequency value, return an object of a unique type that is a subtype of frequency.

## Moment ADT

A particular point in time is represented by the moment ADT. It is expressed in nanoseconds stored in a long long (signed 64-bit integer). The zero point will in most cases be the moment the timing service of the target is initialized. The table below shows that a moment can be obtained by calling the now() function provided by a timing service, and that calculations can be done on moments and durations.

|  |  |  |  |
| --- | --- | --- | --- |
| moment | | | |
| function | return type | meaning | |
| now() | moment | returns the current moment in time | |
| operator | left argument | right argument | result |
| + | moment | duration | moment |
| + | duration | moment | moment |
| - | moment | duration | moment |
| - | moment | moment | duration |
| ==, !=,  >, <, >=, <= | moment | moment | bool |

## Delay service

struct delay {

typedef void has\_delay;

void init();

template< typename D > void wait( D d );

};

Delay is the most primitive timing service. The delay struct shows the interface provided by a delay service. The init() function must be called first. Next wait() can be called, which accepts a duration value and (busy) waits for that amount of time. A target can implement waiting for a literal amount of time in a more efficient way, for instance by inserting a few inline instructions. On targets where execution time can’t be easily predicted and that don’t have a timer wait() might not be very accurate, but it will always err towards waiting longer. Delay will probably have the smallest memory footprint of the timing services, especially when only literal durations are used.

## Clock service

struct clock : public delay {

typedef void has\_clock;

moment now();

template< typename D > void wait( D d );

};

The interface of the clock timing service inherits from delay, hence it provided the services offered by delay. Additionally it supports a notion of absolute time in the form of a now() function that returns a moment, and a wait() function that also accepts (besides the duration that is accepted by the wait() provided by a duration service) a moment. As for a delay service, init() must be called first.

A clock service is typically implemented by a free-running 32-bit or 64-bit timer. It is expected to have a modest footprint, somewhat larger than delay. When it is implemented by a timer smaller than 64 bits (most targets don’t have a 64 bit timer) it might be required that, in order for the service to work properly, a call to now(), or wait() with a moment or a non-literal duration argument, must be made every so often, to allow the service to check for and properly handle a timer overflow.

## Scheduler service

A scheduling service offers the services provided by clock, and additionally supports cooperative multithreading.

TBW

# I/O pins and ports



The pin\_\* interfaces define a standard interface to basic IO pins. The secondary pin interfaces define additional things that can be done with such a pin: enabling or disabling (weak) pull-up or pull-down resistors, or using the pin as an analog input or output. The port\_\* interfaces define what can be done with a port (a group of pins). The pin\_\*\_from and port\_\*\_from wrappers provide a way to convert one type pin or port to another one.

## pin\_\*

These interfaces define what can be done with a basic IO pin. There are four: pin\_in, pin\_out, pin\_in\_out and pin\_oc. The library has wrapper functions pin\_\*\_from that convert a pin into a different pin, for the cases where that makes sense.

struct pin\_in {

typedef void has\_pin\_in;

typedef pin\_in interface\_type;

virtual void init() = 0;

virtual bool get() = 0;

};

A pin\_in can be used for input only (that is: reading the logical value that is presented to the pin by the outside world). It has (beside the mandatory interface\_type and has\_pin\_in markers) the functions init() and get(). The init() function must be called before get() is called.

struct pin\_out {

typedef void has\_pin\_out;

typedef pin\_out interface\_type;

virtual void init() = 0;

virtual void set( bool value ) = 0;

};

A pin\_out can be used for output only (that is: setting the logical value that the pin presents to the outside world). It has (beside the mandatory interface\_type and has\_pin\_out markers) the functions init() and put(). The init() function must be called before set() is called.

struct pin\_in\_out {

typedef void has\_pin\_in\_out;

typedef pin\_in\_out interface\_type;

virtual void init() = 0;

virtual void direction\_set\_input() = 0;

virtual void direction\_set\_output() = 0;

virtual bool get() = 0;

virtual void set( bool value ) = 0;

};

A pin\_in\_out can be used for both for input and for output. It has (beside the mandatory interface\_type and has\_pin\_in\_out markers) the functions init(), direction\_set\_input(), direction\_set\_output, get() and set(bool). The init() function must be called before any of the other functions is called. Before set() is called direction\_set\_output() must be called. Before get() is called the direction\_set\_input() should probably be called, otherwise the pin is not an input and level that is read will likely depend only on the value that was last passed to set().

struct pin\_oc {

typedef void has\_pin\_oc;

typedef pin\_oc interface\_type;

virtual void init() = 0;

virtual bool get() = 0;

virtual void set( bool value ) = 0;

};

A pin\_oc can be used for both for input and for output, but as output it will only drive the pin low, a set(1) call will make the pin high-impedance (input) rather than driving it high. It has (beside the mandatory interface\_type and has\_pin\_oc markers) the functions init(), get() and set(bool). The init function must be called before any of the other functions is called. The before get() is called set(1) should probably be called, otherwise the level that is read will likely be 0 because the pin is driven low.

## pin\_pullup, pin\_pulldown

struct pin\_pullup {

typedef void has\_pin\_pullup;

virtual void pullup\_enable() = 0;

virtual void pullup\_disable() = 0;

};

struct pin\_pulldown {

typedef void has\_pin\_pulldown;

virtual void pulldown\_enable() = 0;

virtual void pulldown\_disable() = 0;

};

The pin\_pullup and pin\_pulldown secondary interfaces provide operations to enable and disable pullup and pulldown resistors.

These interfaces are generally provided by objects that also provide one of the pin\_in, pin\_out, pin\_in\_out, or pin\_oc interfaces, or one of the corresponding port interfaces.

## pin\_ad, pin\_da

The pin\_ad and pin\_da secondary interfaces define how a pin can provide access to its analog-to-digital and digital-to-analog features.

template< unsigned int n\_bits >

struct pin\_ad {

typedef void has\_pin\_ad;

static constexpr int ad\_bits = n\_bits;

typedef typename uint\_t< n\_bits >::fast ad\_value\_type;

static constexpr ad\_value\_type ad\_maximum =

int\_info< ad\_value\_type >::maximum;

virtual void ad\_init() = 0;

virtual void ad\_start() = 0;

virtual bool ad\_get\_will\_block()= 0;

virtual ad\_value\_type ad\_get() = 0;

virtual ad\_value\_type ad\_start\_get(){ . . . }

};

The pin\_ad interface interface provides the means to use a pin as an analog-to-digital converter. The interface is templated on the number of bits (accuracy) of the converter. The interface provides the has\_pin\_ad type that identifies the interface, three declarations that are derived from the number of bits, and a number of functions.

The function ad\_init() must be called first. After that a conversion can be started by calling ad\_start(). Now ad\_get() can be called, which will busy wait until the conversion is finished and then return the conversion result. The function ad\_get\_will\_block() can be called to check whether ad\_get() will block. The default function ad\_start\_get() simply calls ad\_start() and ad\_get().

template< unsigned int n\_bits >

struct pin\_da {

typedef void has\_pin\_da;

static constexpr int da\_bits = n\_bits;

typedef typename uint\_t< n\_bits >::fast da\_value\_type;

static constexpr da\_value\_type da\_maximum =

int\_info< da\_value\_type >::maximum;

virtual void da\_init() = 0;

virtual void da\_set( da\_value\_type n ) = 0;

};

The pin\_da interface provides the means to use a pin as a digital-to-analog converter. The interface is templated on the number of bits (accuracy) of the converter. The interface provides the has\_pin\_da type that identifies the interface, three declarations that are derived from the number of bits, and the two functions: da\_init() must be called first, and after that da\_set() can be used to set the voltage output by the pin, relative to the maximum value. What happens when a value is set that is larger than the maximum is not defined.

These interface is generally provided by an object that also provide one of the pin\_in, pin\_out, pin\_in\_out, or pin\_oc interfaces.

## pin\_\*\_from

The pin\_\*\_from wrappers convert a pin to a pin\_\*.

template< typename P > auto pin\_in\_from( P && pin ){ . . . }

template< typename P > auto pin\_out\_from( P && pin ){ . . . }

template< typename P > auto pin\_in\_out\_from( P && pin ){ . . . }

template< typename P > auto pin\_oc\_from( P && pin ){ . . . }

The pin\_in\_from() wrapper function will, if possible, create a pin\_in from the object you pass to it. A pin\_in, pin\_in\_out or pin\_oc is acceptable, a pin\_out (obviously) is not. When the object is not acceptable you will get a compilation error.

The pin\_out\_from() wrapper function will, if possible, create a pin\_out from the object you pass to it. A pin\_out, pin\_in\_out or pin\_oc is acceptable, a pin\_in (obviously) is not. When the object is not acceptable you will get a compilation error.

The pin\_in\_out\_from() wrapper function will, if possible, create a pin\_in\_out from the object you pass to it. A pin\_in\_out or pin\_oc is acceptable, a pin\_out or pin\_in (obviously) is not. When the object is not acceptable you will get a compilation error.

The pin\_in\_oc\_from() wrapper function will, if possible, create a pin\_oc from the object you pass to it. A pin\_in\_out or pin\_oc is acceptable, a pin\_out or pin\_in (obviously) is not. When the object is not acceptable you will get a compilation error.

## port\_\*

A port is a list of pins, all of the same type. As for pins, there are input, output, input-output and open-collector ports. The port interfaces are templated on the number of pins in the port. Each port interface has a declaration n\_pins that is the number of pins in the port, and a value\_type that is the type accepted by set() and/or returned by get(). The number of pins that can be part of a port is limited by the number of bits in the largest unsigned integer type that is available.

template< int n >

struct port\_in {

typedef void has\_port\_in;

typedef port\_in interface\_type;

static constexpr int n\_pins = n;

typedef typename uint\_t< n >::fast value\_type;

virtual void init() = 0;

virtual value\_type get() = 0;

};

A port\_in can be used to read from the pins that make up the port. It has (beside the mandatory interface\_type and has\_port\_in markers, and the n\_pins and value\_type declarations) the functions init() and get(). The init() function must be called before get() is called. Get returns a value in which each bit represents the value read from one pin.

template< int n >

struct port\_out {

typedef void has\_port\_out;

typedef port\_out interface\_type;

static constexpr int n\_pins = n;

typedef typename uint\_t< n >::fast value\_type;

virtual void init() = 0;

virtual void set( value\_type x ) = 0;

};

A port\_out can be used to write to the pins that make up the port. It has (beside the mandatory interface\_type and has\_port\_out markers, and the n\_pins and value\_type declarations) the functions init() and set(). The init() function must be called before set() is called. Each bit in the value provided to set() is written to a pin that is part of the port.

template< int n >

struct port\_in\_out {

typedef void has\_port\_in\_out;

typedef port\_in\_out interface\_type;

static constexpr int n\_pins = n;

typedef typename uint\_t< n >::fast value\_type;

virtual void init() = 0;

virtual void direction\_set\_input() = 0;

virtual void direction\_set\_output() = 0;

virtual value\_type get() = 0;

virtual void set( value\_type value ) = 0;

};

A port\_in\_out can be used as port\_out (after direction\_set\_output() has been called), or as port\_in (after direction\_set\_input() has been called. There is no way to switch the direction of individual pins.

template< int n >

struct port\_oc {

typedef void has\_port\_oc;

typedef port\_oc interface\_type;

static constexpr int n\_pins = n;

typedef typename uint\_t< n >::fast value\_type;

virtual void init() = 0;

virtual value\_type get() = 0;

virtual void set( value\_type value ) = 0;

};

A port\_oc supports reading and writing to the set of pins that make up the port. Each pin behaves as an open-collector in: writing a 0 to it makes it low, writing a 1 to it makes it high-impedance (input).

## port\_\*\_from\_pins

|  |
| --- |
|  |

TBW

The port\_\*\_from wrapper functions convert a list of pins into a port of the requested type. An individual pin can be retrieved from a port. For use by the port, all pins are converted to the requested type (in, out, in\_out, or oc) using the appropriate pin\_\*\_from wrapper. When a pin is not acceptable to that wrapper a compiler error is generated.

The individual pins of a port can be read (get() function) or written (set() function) in one action. For a port\_in\_out, the direction of all pins can be set, but only collectively.

## port\_\*\_from

TBW

## invert

template< typename P > auto invert( P && p ){ . . . }

The invert wrapper function returns an object that inverts the operation of the object that it gets passed. For digital interfaces (pin\_\* and port\_\*) a 0 will be read or written as a 1, and a 1 as a 0. For pullups and pulldowns all actions on a pullup will result in actions on a pulldown, and vice versa. For proportional interfaces a value x will be read or written as ( max – x ), where max is the maximum value of the interface. When the object is a port, the actions for all pins are inverted.

## mirror

TBW

## dummies

Occasionally a dummy pin or port parameter is needed because an interface requires an object, but as far as the user is concerned the operations on the object don’t need to have any effect. For this purpose some dummy objects are provided.

Needs gcc 5 ☹

TBW: tee, both, all, … ?

# Graphics

This section describes how the library handles graphics.

# External chips

This section documents the library components that interface to external chips.

## I2C

## SPI

## D1W

## ADS7843

## DS18x20

## HC595

## HD44780

## MCP23xxx

## PCF8574(A)

## PCF8591

## PCD8544

# Miscellaneous



## default\_baudrate

constexpr int default\_baudrate = …;

This constant is the default baudrate used for downloading and for communication with the host computer. When used from bmptk, it is the BMPTK\_BAUDRATE, otherwise it defaults to 19200.

## HWCPP\_INLINE

#define HWCPP\_INLINE …

The function decorator HWCPP\_INLINE requests the compiler to always inline the decorated function.

## HWCPP\_REQUIRE\_INTERFACE

#define HWCPP\_REQUIRE\_INTERFACE( P, T ) ...

The macro invocation HWCPP\_REQUIRE\_INTERFACE( P, T ) can be placed in a template to assure that the parameter P has a static void type T in it, which by convention indicates that it implements the interface with that name.

## interface

template< typename T >

typename T::interface\_type & HWCPP\_INLINE interface( T & x ) …

template< typename T >

typename T::interface\_type HWCPP\_INLINE interface( const T & x ) …

The interface template function casts an object or value to its interface\_type. It is used to erase the type of a parameter down to its interface type, to avoid instantiating a different callee for each parameter type.

# Naming conventions



## \*\_from

A wrapper function X\_from returns an object that implements the X interface from another (or the same!) type of object. Such a wrapper can be used at the start of a duck-typed wrapper, to check that a parameter is acceptable, and at the same time convert it to the most convenient type.

## get, set

The get() and set() names are used for functions that are expected to be idempotent: calling get() twice without any intervening event that changes the ’stored value’ should return the same value, and calling set() twice with the same value should have the same effect as calling it once.

## has\_\*

Each interface X has a nested void type with the name has\_X to indicate that the interface is provided. This is used by many wrappers to select the appropriate wrapper for the type of object that is passed.

## interface\_type

Each interface has an interface that is defined as the interface type itself. This is used by the interface() function template to erase the type of its parameter down to its interface type.

# References

# ToDo

* Header number must have space instead of tab
* Index
* Re-visit the introduction
* Example, gentle introduction
* Explain ‘no offset’ timing loop with now()
* Loop<>
* Targets
* Getting started, way of use
* logical\_or( a, b, … ) \_and \_xor on pins or ports
* filter/debounce
* once, clear\_on\_read
* toggle
* most targets don’t support

1. The extensions used are \_\_attribute\_\_((always\_inline)) and \_\_builtin\_constant\_p(). [↑](#footnote-ref-1)
2. This init() mechanism is used rather than the more user-friendly do-it-in-the-constructor approach to make sure that constructors can be written without side effects. This makes it feasible for the toolchain to eliminate unused objects, which would be impossible if the constructors had to use peripheral registers, which by their nature must be declared volatile. [↑](#footnote-ref-2)
3. When the functionality of a concrete class is to be enhanced or otherwise modified by a decorator, delegation is used (rather than inheritance) to access the functionality of the underlying class object. [↑](#footnote-ref-3)
4. Hence a wrapper is a generalization of the decorator (returns same interface) and adapter (returns different interface) patterns. Often the same name is used to for a number of wrappers, selected by the interface implemented by the argument object. Hence one of these wrappers can be a decorator, and the others are adapters, but that only confuses their purpose. The essence (in that case) is that they all return an object that implements the same interface. [↑](#footnote-ref-4)
5. Duck typing refers to the phrase ”If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” or something similar. [↑](#footnote-ref-5)
6. Abstract Data Type: something that behaves more or less like an integer. [↑](#footnote-ref-6)
7. I don’t know what a negative frequency is, but you can express one. The reason is that I wanted to save code size by avoiding using both 64-signed and 64-bit unsigned libraries. The cost is one wasted bit. [↑](#footnote-ref-7)
8. This might cause a problem for a system that runs for more than 292 year without a restart. Don’t expect me to be around when this causes a problem. [↑](#footnote-ref-8)
9. The types returned by the frequency literal\_\* function templates have a period() function that returns a period subtype as it would be returned by duration literal\_\* function templates. [↑](#footnote-ref-9)