Godafoss reference

Contents

1 passing a const parameter

from: ../basics/gf-passing.hpp

```
// use by_const< T > when passing a T
template< typename passed_type >
using by_const = ...
```

[>by_const<]

The by_const< type > template is the preferred way to pass a const value of the type passed_type. This will be either a plain (by copy) const, or a const reference, depending (among other things) on the size of the type compared to a the size of a reference.

```
#include "array"
#include "godafoss.hpp"
namespace gf  = godafoss;

template< typename T >
void function( gf::by_const< T > p ){ (void) p; }

int main(){

   // should probably be passed by value (copy)
   function( 'x' );

   // should be probably be passed by reference
   function( std::array< int, 100 >{ 0 } );

};
```

2 random

from: ../basics/gf-random.hpp

This is simple 32-bit LCG random function, for demos and games. The random facilities of the standard library are not used because they eat up too much RAM. Do NOT use this for crypto work.

The LCG used is the Microsoft Visual/Quick C/C++ variant as explained on https://en.wikipedia.org/wiki/Linear_congruential_generator, but using bits 8..23 rather than 16..30.

[>random16<]

```
uint16_t random16(){ ... }
```

This function returns a 16-bit non-negative pseudo-random number.

[>random32<]

```
uint32_t random32(){ ... }
```

This function returns a 32-bit non-negative pseudo-random number.

[>random_in_range<]

```
template< typename int_type >
int_type random_in_range( by_const< int_type >first, by_const< int_type > last ){ ... }
```

This function returns a non-negative pseudo-random number in the range [first .. last]. This number is calculated from a number generated by random32 by modulo arithmetic. This is simple and fast, but the distribution is not ideal: the higher values in the range will be somewhat underrepresented. When the width of the range is much smaller than 2^32 this effect will be small.

[>random_seed<]

```
void random_seed( by_const< uint32_t > x ){ ... }
```

This function sets the start for the value returned by subsequent random calls. It can be used to re-start a random sequence, or (when you have a truely random source) to start a truely random random sequence.

3 background processing

from: ../basics/gf-background.hpp

[>background<]

The background class provides a hook for run-to-completion style background processing.

```
struct background : public not_copyable {
   // This function will be called to do background work for its object.
   virtual void work() = 0;
};
```

A class that needs background processing must inherit from background and implement the work function. This work function will be called when plain wait functions (the ones that allow background processing) are called.

When an application contains background work, all plain wait functions can take longer than the specified time, up to the run time of the longest runtime of the work() functions.

No background work will be done from wait calls made while a work() function is running.

For all background jobs: be careful to preserve the object, or your servicing will end. This is not UB: the background destructor removes itself from the list of background jobs.

When the application would terminate (exit from its main()), background::run() can be called instead, which will serve the background processing (it will never return).

```
#include "godafoss.hpp"
namespace gf  = godafoss;

struct background_work: gf::background {

   auto last = gf::target::now_ms();

   void work() override(){
      now = gf::target::now_ms();
      if( now > last + 1'000 ){
         gf::cout << "Another second has passed\n";
      }
   }

};

int main(){

   {
      background_work annnoying;
      for( int i = 0; i < 10; ++i ){
         gf::wait_ms( 2'800 );
         gf::cout << "[" << i << "] 2.8 seconds passed\n"
      }
      // annnoying is destructed here, so it will finally shut up
   }

   for( int i = 0; i < 10; ++i ){
```

```
      gf::wait_ms( 2'100 );
      gf::cout << "[" << i << "] 2.1 seconds passed\n"
   }

};
```

4 ints specified by number of bits

from: ../basics/gf-ints.hpp

[>uint_bits<]

```
template< uint64_t n > struct uint_bits {
   typedef typename ...
      fast;
   typedef typename ...
      least;
};
```

uint_bits< N >::fast is the smallest 'fast' unsigned integer type that stores (at least) N bits.

uint_bits< N >::least is the smallest (but not necesarrily fast) unsigned integer type that stores (at least) N bits.

As both are unsigned they should be used for bit patterns, not for amounts.

Note that both can be larger than requested, so they should not be used for modulo arithmetic (at least not without masking out excess bits).

Use uint_bits< N >::fast for variables and parameters, use uint_bits< N >::least for arrays.

5 function and class attributes

from: ../basics/gf-attributes.hpp

[>GODAFOSS_INLINE<]

```
#define GODAFOSS_INLINE  ...
```

GODAFOSS_INLINE forces a function to be inline. It is used when the function body is very simple, for instance when it calls only one deeper function. This serves (only) to reduce code size and execution time.

[>GODAFOSS_NO_INLINE<]

```
#define GODAFOSS_NO_INLINE  ...
```

GODAFOSS_NO_INLINE forces a function to be not inline. This is used to preserve low-level properties of a function, like the number of cylces taken by the function preable and postamble. This can be important to get predictable timing.

[>GODAFOSS_NO_RETURN<]

```
#define GODAFOSS_NO_RETURN  ...
```

GODAFOSS_NORETURN indicates that a function will not return. It is used for functions that contain a never-ending loop. This can reduce code size.

[>GODAFOSS_IN_RAM<]

```
#define GODAFOSS_IN_RAM  ...
```

GODAFOSS_IN_RAM places the function body in RAM (instead of FLASH). On some targets, this is necesarry to get predicatable timing, or faster execution.

[>GODAFOSS_RUN_ONCE<]

```
#define GODAFOSS_RUN_ONCE                ...
```

GODAFOSS_RUN_ONCE causes the remainder of the function (the part after the macro) to be executed only once.

[>not_constructible<]

```
struct not_constructible {
```

Inheriting from not_constructible makes it impossible to create objects of that class.

[>not_copyable<]

```
struct not_copyable {
```

Inheriting from not_copyable makes it impossible to copy an object of that class.

6 hd44780

from: ../chips/gf-hd44780.hpp

[>hd44780_rs_e_d_s_timing<]

```
template<
   pin_out_compatible   rs,
   pin_out_compatible   e,
   port_out_compatible  port,
   xy<>                 size,
   typename             timing
> using hd44780_rs_e_d_s_timing = ...
```

This template implements a terminal on an hd44780 character lcd.

The rs, e and port must connect to the corresponding pins of the lcd. The lcd is used in 4-bit mode, so the port must connect to the d0..d3 of the lcd, the d4..d7 can be left unconnected. Only writes to the lcd are used. The _r/w pin must be connected to ground.

The size of the lcd must be specified in characters in the x and y direction. Common sizes are 16x1, 16x2, 20x2 and 20x4.

The timing is used for the waits as required by the hd44780 datasheet.

7 hx711

from: ../chips/gf-hx711.hpp

[>hx711<]

This template implements an interface to the hx711 24-Bit Analog-to-Digital Converter (ADC). This chip is intended to interface to a load cell (force sensor).

```
template<
   pin_out_compatible  _sck,
   pin_in_compatible   _dout,
   typename            timing
>
struct hx711 {
   enum class mode {
      a_128  = 1,  // A inputs, gain 128
      b_32   = 2,  // B inputs, gain 32
      a_64   = 3   // A inputs, gain 64
   };
   static void init( mode m = mode::a_128 ){ ... }
   static int32_t read(){ ... }
   static void power_down(){ ... }
   static void mode_set( mode m ){ ... }
};
```

The chip interface consist of a master-to-slave clock pin (sck), and a slave-to-master data pin (dout).

The timing is used for the waits as required by the hx711 datasheet.

The mode offers a choice between the A differential inputs with a gain of 128 or 64, and the B inputs with a gain of 32. The A inputs are meant to be used with a load cell. The datasheet suggest that the B inputs could be used to monitior the battery voltage. The mode is set at the initialization (the defauylt is a_128), and can be changed by the mode_set() function.

The chip can be powered down. When a read is done the chip is first (automatically) powered up.