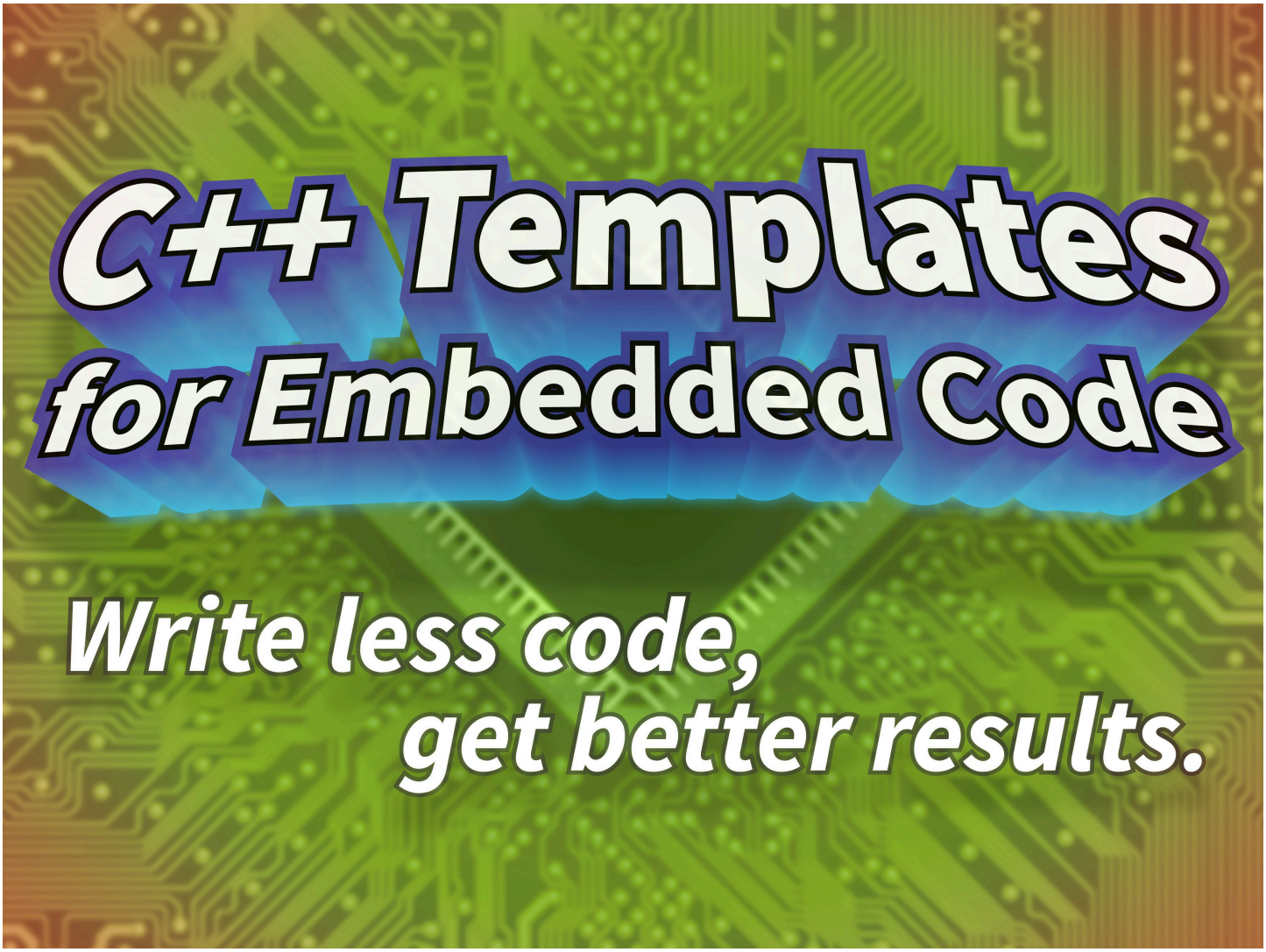
C++ Templates for Embedded Code

I will explain how templates work and how you may use them in your code.

Jul 27, 2019•30468 views•9 respects

template templates learn c++



Components and supplies

1 Arduino UNO

Apps and platforms

1 Arduino IDE 

Project description

Templates are a **powerful feature of the C++ language** but their syntax can be complex. This causes some developers to not use them, while others are concerned that templates might cause bloat to the compiled code.

I will explain *how templates work* and how you may use them in your code. Although the examples I provide are for the Arduino Uno, the concepts *will work with any platform and with all modern C++ compilers* .

The **first part** explains the **basics and function templates** . Then I will explain *template classes* and *template specialisation* .

What are Templates

Templates were introduced to allow a single function to **work with many different types** or to build **generic types** from classes. They also enable an efficient way for the compiler to generate many variants of the same code for you.

If you are experienced with Python or JavaScript you may be familiar with **dynamically typed** variables. In these languages, the same function can be called with parameters of various types. Be careful **not to confuse** dynamically typed variables with the concept of templates.

In C++ the types of *function templates* and *template classes* are defined at compile-time and thus, they are **type-safe** . In Python and JavaScript, however, the dynamic types of variables are **checked at runtime** and are *not type-safe* .

Here is a simple example written for the Arduino Uno:

```
template<typename Value>

void circularShift(Value &value) {

            const bool firstSet = (value & 0b1u);

            value >>= 1;

            if (firstSet) {

                value |= static_cast<Value>(0b1u) << (sizeof(Value)*8-1);

            }

}

uint32_t value1 = 0xabcd0123u;

uint16_t value2 = 0x4545u;

void setup() {

            Serial.begin(115200);

            while (!Serial) {}

}

void loop() {

            Serial.println(value1, BIN);

            Serial.println(value2, BIN);

            circularShift(value1);

            circularShift(value2);

            delay(1000);

}
```

To compile the code, make sure you use the latest version of the **Arduino IDE,** version 1.8.9 or later.

In this example, the `circularShift` function is similar to executing `>> 1` on a value, shifting all of the bits one position to the right. The difference is that `circularShift` "wraps" the bits, so the right-most bit (which would otherwise be shifted out of the value) moves to the left-most bit. For example:

```
0b11100101 >> 1  // results in 01110010
```

```
circularShift(0b11100101) // results in 11110010
```

To test the function, we use a 32-bit and a 16-bit value declared in lines 10 and 11 of the example code. In the `loop()` function the values are printed and then circular-shifted to the right in lines 21 and 22.

It is not necessary to understand the syntax of the function template for now, I will discuss this later on.

Compiling the Example

Processing our example the compiler will **treat the function template differently** than regular functions. It will keep the template function prototype in memory as it is but will not generate any code for it, yet.

To actually create code from the function prototype the function template needs to be **instantiated** . This process happens as soon the compiler sees a call to the function.

Therefore, in line 21 of the example code the compiler will first try to **determine the types** for the function template `circularShift` where there is one type `Value` that *we did not specify it explicitly* . The compiler now has to *get the type implicitly* by analysing the parameter `value1` . The parameter is declared as `uint32_t` and therefore Value has to be `uint32_t` .

After **resolving all template parameters** the compiler will check if it already generated a copy of this function prototype using the given type. In our case there was none made and therefore the compiler will use the prototype and try to compile a new variant of the function using the given type `uint32_t` .

The same happens in line 22 of the code, but this time with the `uint16_t` type. If the prototype function *could not be compiled using the given template parameters* the compiler would stop with an error message.

The Generated Code

The compiler **made two different variants** of the same function using two different types. Let’s look at the generated code:

```
642:  40 91 02 01   lds r20, 0x0102 ; 0x800102 <value1>
646:  50 91 03 01   lds r21, 0x0103 ; 0x800103 <value1+0x1>
64a:  60 91 04 01   lds r22, 0x0104  ; 0x800104 <value1+0x2>
64e:  70 91 05 01   lds r23, 0x0105  ; 0x800105 <value1+0x3>
652:  db 01         movw   r26, r22
654:  ca 01         movw   r24, r20
656:  b6 95         lsr r27
658:  a7 95         ror r26
65a:  97 95         ror r25
65c:  87 95         ror r24
65e:  40 fd         sbrc   r20, 0
660:  b0 68         ori r27, 0x80 ; 128
662:  80 93 02 01   sts 0x0102, r24  ; 0x800102 <value1>
666:  90 93 03 01   sts 0x0103, r25  ; 0x800103 <value1+0x1>
66a:  a0 93 04 01   sts 0x0104, r26  ; 0x800104 <value1+0x2>
66e:  b0 93 05 01   sts 0x0105, r27  ; 0x800105 <value1+0x3>
```

uint32_t

```
672:  20 91 00 01   lds r18, 0x0100  ; 0x800100 <__data_start>
676:  30 91 01 01   lds r19, 0x0101  ; 0x800101 <__data_start+0x1>
67a:  c9 01         movw   r24, r18
67c:  96 95         lsr r25
67e:  87 95         ror r24
680:  20 fd         sbrc   r18, 0
682:  90 68         ori r25, 0x80 ; 128
684:  90 93 01 01   sts 0x0101, r25  ; 0x800101 <__data_start+0x1>
688:  80 93 00 01   sts 0x0100, r24  ; 0x800100 <__data_start>
```

uint16_t

The Function Prototype

Here is what happens when our example code is compiled:

```
template<typename Value>
void circularShift(Value &value) {
    const bool firstSet = (value & 0b1u);
    value >>= 1;
    if (firstSet) {
        value |= static_cast<Value>(0b1u) << (sizeof(Value)*8-1);
    }
}
```

```
template<uint32_t>
void circularShift(uint32_t &value) {
    const bool firstSet = (value & 0b1u);
    value >>= 1;
    if (firstSet) {
        value |= static_cast<uint32_t>(0b1u) << (sizeof(uint32_t)*8-1);
    }
}
```

```
template<uint16_t>
void circularShift(uint16_t &value) {
    const bool firstSet = (value & 0b1u);
    value >>= 1;
    if (firstSet) {
        value |= static_cast<uint16_t>(0b1u) << (sizeof(uint16_t)*8-1);
    }
}
```

The compiler replaced the template parameter `Value` with the required type and compiled the resulting code. So while a template will allow you to use different types for the same function, **the process of compiling the code is type-safe** .

You can test this by adding code to the example: create a new `double` variable and try to call the `circularShift` function on it.

```
double value = 100.0;

circularShift(value);
```

This results in error messages from the compiler:

```
Templates_2.ino: In instantiation of 'void circularShift(Value&) [with Value =
double]':

Templates_2.ino:20:24:   required from here

Templates_2:3:34: error: invalid operands of types 'double' and 'unsigned int' to
binary 'operator&'

            const bool firstSet = (value & 0b1u);

Templates_2:4:11: error: invalid operands of types 'double' and 'int' to binary
'operator>>'

            value >>= 1;

Templates_2:4:11: error: in evaluation of 'operator>>=(double, int)'

Templates_2:6:43: error: invalid operands of types 'double' and 'unsigned int' to
binary 'operator<<'

                value |= static_cast<Value>(0b1u) << (sizeof(Value)*8-1);
```

“In instantiation” means that it happened **in the process where the compiler tried to generate new code for a call** . You can also see the function which caused the error together with the template parameters the compiler used in square brackets. This is the code the compiler tried to process:

```
template<double>

void circularShift(double &value) {

           const bool firstSet = (value & 0b1u);

           value >>= 1;

           if (firstSet) {

               value |= static_cast<double>(0b1u) << (sizeof(double)*8-1);
```

```
            }

}
```

After this initial line, you see all errors while compiling the code using the given template parameter. Shift operations are not available for floating-point types, neither is the bitwise OR operation. Therefore, the function template **will not compile** with a `double` parameter, which is the reason why templates are **type-safe** .

Compare Templates with Python Code

C++ detects type errors in templates at compile time while dynamically typed languages detect them at run-time.

Let us examine this Python example:

```
def circularShift(value):

            value = (value >> 1)



value1 = 1

value2 = 0.5

rollBits(value1)

rollBits(value2)
```

(Note that it is not a copy of the C++ template because the integer sizes in Python are dynamic.)

You can successfully **compile** the script into bytecode.

```
$ python3 -m compileall roll.py

Compiling 'roll.py'...
```

Starting it will give you these error messages:

```
$ python3 roll.py

Traceback (most recent call last):

          File "roll.py", line 8, in <module>

            rollBits(value2)
```

```
         File "roll.py", line 3, in rollBits

           value = (value >> 1)

TypeError: unsupported operand type(s) for >>: 'float' and 'int'
```

You can still have runtime errors in C++ template code, but *type related problems* are detected **at compile-time** . It is a benefit of type-safe languages like C++, C, Swift or Rust.

Finding errors as early as possible will save time and costs.

Writing unit-tests early is crucial for languages like Python or JavaScript. Only these provide all the necessary checks.

Function Template Syntax

There are **function templates** and **template classes** . For now, I will only discuss function templates.

A function template is created using the `template` keyword with a list of parameters in angled brackets `<...>` :

```
template<...> [function declaration]
```

Put the `template` part into its own line to make the code readable:

```
template<typename Type>

void myFunction(Type value) {

}
```

The **declaration and implementation** of a function template have to be in the **header** file. You can not put the body in an implementation file as with regular functions. There may be exceptions to this rule, but the compiler needs to know the *function prototype* to *generate* the implementation.

Template Parameters

Template parameters **declare types** or values and they look like regular function parameters. Types are declared using the `typename` keyword or `class` keyword. With minor exceptions, `typename` and `class` are interchangeable for a template

parameter. The `class` keyword was superseded by `typename`. If your compiler supports it, use the modern `typename` syntax.

Declaring value parameters with **primitive data types**, like `int`, `double`, or `char` is possible too. To use a value parameter, you have to pass a literal value or constant to the template function.

Write **type parameter names** as you would write class names, with an initial upper-case letter. Value parameters have no clear formatting rule, so I use the same format as for regular function parameters.

Have a look at these valid function templates:

```
template<typename T>

T createType() {

        // ...

}

template<typename T>

void process(T t) {

        // ...

}

template<typename Value, typename Key, typename SizeType = uint32_t>

void addValue(const Key &key, const Value &value) {

        // ...

}

template<typename Value, uint8_t count = 1>

Value shuffle(Value base) {

        // ...

}

template<uint32_t address, uint8_t delay>

void sendSignal() {

        // ...
```

```
}
```

Function Template Usage

We use function templates in two different ways: With **explicit template parameters** or without. If we are not explicit the compiler will try to deduce the template parameters from its context.

The next example reverses all bits in a given value. If we pass `0b00000011` to the function we get back `0b11000000` .

```
/// Reverse all bits in the given value.

///

/// @tparam Type The type of the value. Has to be a unsigned integer.

/// @param value The value to to reverse the bits.

/// @return The value with reversed bits.

///

template<typename Type>

Type reverseBits(Type value) {

         Type result = 0;

         for (uint8_t i = 0; i < (sizeof(Type)*8); ++i) {

           result <<= 1;

           if ((value & static_cast<Type>(0b1)) != 0) {

             result |= static_cast<Type>(0b1);

           }

           value >>= 1;

         }

         return result;

}
```

The function template works with `uint8_t` , `uint16_t` , `uint32_t` and `uint64_t` **with a single implementation** . `Type` is used as a function argument, therefore the

compiler can implicitly deduce the template parameter.

If you pass a `uint16_t` argument to the function the compiler assumes you did like to use `uint16_t` for the `Type` parameter.

The next example uses the implicit syntax to call the function templates, without `<...>` brackets:

```
void setup() {

        Serial.begin(115200);

        while (!Serial) {}

        uint32_t valueIn1 = 0b10000000000000000000000010110111u;

        auto valueOut1 = reverseBits(valueIn1);

        Serial.println(valueIn1, BIN);

        Serial.println(valueOut1, BIN);

        uint16_t valueIn2 = 0b1000000010110111u;

        auto valueOut2 = reverseBits(valueIn2);

        Serial.println(valueIn2, BIN);

        Serial.println(valueOut2, BIN);



        uint8_t valueIn3 = 0b10011011u;

        auto valueOut3 = reverseBits(valueIn3);

        Serial.println(valueIn3, BIN);

        Serial.println(valueOut3, BIN);

}
```

The result on the console is:

```
10000000000000000000000010110111

11101101000000000000000000000001

1000000010110111

1110110100000001
```

```
10011011

11011001
```

The function reverses the value of three different integer values correctly. If we change the code and pass the values as literals, however, we'll get **"unexpected" results** :

```
void setup() {

        Serial.begin(115200);

        while (!Serial) {}

        auto valueOut1 = reverseBits(0b10000000000000000000000010110111u);

        Serial.println(valueOut1, BIN);

        auto valueOut2 = reverseBits(0b10000000010110111u);

        Serial.println(valueOut2, BIN);

        auto valueOut3 = reverseBits(0b10011011u);

        Serial.println(valueOut3, BIN);

}
```

The new output on the console is:

```
11101101000000000000000000000001

1110110100000001

1101100100000000
```

The first two results are correct, but the last is a 16-bit integer instead of an 8-bit one.

This is how the compiler works. The smallest numerical integer value is `short` . Therefore, it will use `unsigned short` or `uint16_t` for the `0b10011011u` literal.

To solve this, we explicitly specify the template parameters for each call. While strictly only necessary for the first call, I use this syntax for all calls:

```
void setup() {

        Serial.begin(115200);

        while (!Serial) {}
```

```
        auto valueOut1 = reverseBits<uint32_t>(0b10000000000000000000000010110111u);

        Serial.println(valueOut1, BIN);

        auto valueOut2 = reverseBits<uint16_t>(0b1000000010110111u);

        Serial.println(valueOut2, BIN);

        auto valueOut3 = reverseBits<uint8_t>(0b10011011u);

        Serial.println(valueOut3, BIN);

}
```

Finally, the correct output on the console is:

```
11101101000000000000000000000001

1110110100000001

11011001
```

If you specify the template parameters explicitly they will take precedence over any other method.

Recap

```
// Implicit function template call

function(a, b, c)

// Explicit function template call

function<int, double, float>(a, b, c)
```

Practical Use Cases

Type Independent Libraries

Function templates can help you write platform-independent tool libraries **without macros** and without static configuration headers.

You may want to work with `uint16_t` on one platform and with `uint32_t` on another. The same is true for `float` or `double` or a custom fixed-point number type.

Without function templates, you have to use overloaded functions to get the same results, but this will produce repetitive code.

```
void reverseBits(uint32_t &value) {

         auto bits = value;

         uint32_t result = 0;

         for (uint8_t i = 0; i < 32; ++i) {

           result <<= 1;

           if ((bits & static_cast<uint32_t>(0b1)) != 0) {

             result |= static_cast<uint32_t>(0b1);

           }

           bits >>= 1;

         }

         value = result;

}

void reverseBits(uint16_t &value) {

         auto bits = value;

         uint16_t result = 0;

         for (uint8_t i = 0; i < 16; ++i) {

           result <<= 1;

           if ((bits & static_cast<uint16_t>(0b1)) != 0) {

             result |= static_cast<uint16_t>(0b1);

           }

           bits >>= 1;

         }

         value = result;

}

void reverseBits(uint8_t &value) {

         auto bits = value;
```

```
        uint8_t result = 0;

        for (uint8_t i = 0; i < 8; ++i) {

          result <<= 1;

          if ((bits & static_cast<uint8_t>(0b1)) != 0) {

            result |= static_cast<uint8_t>(0b1);

          }

          bits >>= 1;

        }

        value = result;

}
```

Working with Multiple Values

To reverse bitmasks in your application you probably use a library with an interface like this:

```
void reverseBits(uint32_t &value);

void reverseBits(uint16_t &value);

void reverseBits(uint8_t &value);
```

You may come from C, and use a different interface approach:

```
void reverseBits64(uint64_t &value);

void reverseBits32(uint32_t &value);

void reverseBits16(uint16_t &value);

void reverseBits8(uint8_t &value);
```

For the first interface, or if you start from scratch, replace all functions with a function template. It has to be placed in the header file to make the function prototype available for the compiler.

```
template<typename Type>

void reverseBits(Type &value) {

        auto bits = value;
```

```
        Type result = 0;
        for (uint8_t i = 0; i < (sizeof(Type)*8); ++i) {
          result <<= 1;
          if ((bits & static_cast<Type>(0b1)) != 0) {
            result |= static_cast<Type>(0b1);
          }
          bits >>= 1;
        }
        value = result;
}
```

If you work with the old C interface, or if you like to encapsulate the implementation, you wrap the function templates as shown in the next three examples:

A: ReverseBits.hpp

```
#pragma once
#include <Arduino.h>
template<typename Type>
void reverseBitsT(Type &value) {
        auto bits = value;
        Type result = 0;
        for (uint8_t i = 0; i < (sizeof(Type)*8); ++i) {
          result <<= 1;
          if ((bits & static_cast<Type>(0b1)) != 0) {
            result |= static_cast<Type>(0b1);
          }
          bits >>= 1;
        }
```

```
        value = result;
}
inline void reverseBits(uint32_t &value) {
        reverseBitsT(value);
}
inline void reverseBits(uint16_t &value) {
        reverseBitsT(value);
}
inline void reverseBits(uint8_t &value) {
        reverseBitsT(value);
}
```

This example limits the `reverseBits` function to three integer types and provides the template function with the name `reverseBitsT`. It is backwards compatible but you have access to the new template. `inline` is used as a suggestion for the compiler to resolve the function calls and use the function template directly.

B: ReverseBits.hpp

```
#pragma once
#include <Arduino.h>
template<typename Type>
void reverseBits(Type &value) {
        auto bits = value;
        Type result = 0;
        for (uint8_t i = 0; i < (sizeof(Type)*8); ++i) {
          result <<= 1;
          if ((bits & static_cast<Type>(0b1)) != 0) {
            result |= static_cast<Type>(0b1);
          }
          bits >>= 1;
```

```
        }

        value = result;

}

inline void reverseBits64(uint64_t &value) {

        reverseBits(value);

}

inline void reverseBits32(uint32_t &value) {

        reverseBits(value);

}

inline void reverseBits16(uint16_t &value) {

        reverseBits(value);

}

inline void reverseBits8(uint8_t &value) {

        reverseBits(value);

}
```

This example is a progressive extension of the C interface. Old code will compile perfectly using the known function names, and new code can call the function template `reverseBits` .

C: ReverseBits.hpp

```
#pragma once

#include <Arduino.h>

void reverseBits(uint64_t &value);

void reverseBits(uint32_t &value);

void reverseBits(uint16_t &value);

void reverseBits(uint8_t &value);
```

C: ReverseBits.cpp

```
#include "ReverseBits.hpp"
```

```
template<typename Type>

void reverseBitsT(Type &value) {

         auto bits = value;

         Type result = 0;

         for (uint8_t i = 0; i < (sizeof(Type)*8); ++i) {

           result <<= 1;

           if ((bits & static_cast<Type>(0b1)) != 0) {

             result |= static_cast<Type>(0b1);

           }

           bits >>= 1;

         }

         value = result;

}

void reverseBits(uint64_t &value) {

         reverseBitsT(value);

}

void reverseBits(uint32_t &value) {

         reverseBitsT(value);

}

void reverseBits(uint16_t &value) {

         reverseBitsT(value);

}

void reverseBits(uint8_t &value) {

         reverseBitsT(value);

}
```

The last example encapsulated the implementation. You can only use a limited set of four types and there is no way accessing the function template directly. A user of the library can only see the interface, which allows you to rewrite the

implementation at any time. In this case, the function template is only a clever way to avoid repetitive code.

Self Optimizing Code

In embedded applications, you often use **specialized code** to access registers in a certain order. For speed reasons or for implementing a special communication protocol.

This results in either **inefficient code** with many nested `if` and `else` statements or you get a **repetitive** list of functions with minimal differences.

Function templates solve this problem with a unified interface for all calls. Each time you use the function with different template parameters, the compiler generates new code. These generated functions are compiled separately, which produces highly optimised code.

Encode a Really Fast Signal

To encode a very fast signal we need to use direct register access. We like to send the signal over multiple lines, therefore the solution has to be flexible and the port and pins have to be configurable.

As reference we will use a slow example. Some Arduino developers would implement the protocol like this:

SenderSlow.ino

```
#include "Sender.hpp"

Sender sender;

const char data[] = "Hello World!";

void setup() {

         sender.begin();

}

void loop() {

         for (uint8_t dataByte : data ) {

           sender.sendByte(dataByte);
```

```
        }

}
```

Sender.hpp

```
#pragma once

class Sender

{

public:

        Sender(uint8_t clockPin = 12, uint8_t dataPin = 13)

          : _clockPin(clockPin), _dataPin(dataPin) {

        }

        void begin() {

          pinMode(_clockPin, OUTPUT);

          pinMode(_dataPin, OUTPUT);

          digitalWrite(_clockPin, LOW);

          digitalWrite(_dataPin, LOW);

        }

        void sendByte(uint8_t data) {

          sendBit((data & (0b1u<<0)) != 0);

          sendBit((data & (0b1u<<1)) != 0);

          sendBit((data & (0b1u<<2)) != 0);

          sendBit((data & (0b1u<<3)) != 0);

          sendBit((data & (0b1u<<4)) != 0);

          sendBit((data & (0b1u<<5)) != 0);

          sendBit((data & (0b1u<<6)) != 0);

          sendBit((data & (0b1u<<7)) != 0);

        }

private:
```

```
        void sendBit(bool oneBit) {
          digitalWrite(_dataPin, (oneBit ? HIGH : LOW));
          digitalWrite(_clockPin, HIGH);
          digitalWrite(_clockPin, LOW);
        }
private:
        uint8_t _clockPin;
        uint8_t _dataPin;
};
```

This is a simple implementation of the SPI protocol. The analysis with the Saleae logic analyzer produces this result:

It takes 0.48ms for the reference implementation to send the initial `Hello `, which equals ~10.4 KB per second.

This is way too slow, therefore, some desperate Arduino developers will rewrite the code like this:

SenderBadIdea.ino

```
#include "Sender.hpp"
Sender sender;
const char data[] = "Hello World!";
void setup() {
        sender.begin();
}
void loop() {
        for (uint8_t dataByte : data ) {
          sender.sendByte(dataByte);
        }
}
```

The main file is unchanged.

Sender.hpp

```
#pragma once

// BAD EXAMPLE CODE - DO NOT USE!

#define SENDER_PORT PORTB

#define SENDER_DIR DDRB

#define SENDER_DATA (0b00100000u)

#define SENDER_CLOCK (0b00010000u)

class Sender

{

public:

         void begin() {

           SENDER_DIR |= (SENDER_DATA|SENDER_CLOCK);

           SENDER_PORT &= ~(SENDER_DATA|SENDER_CLOCK);

         }

         void sendByte(uint8_t data) {

           for (uint8_t i = 0; i < 8; ++i) {

             sendBit((data & 0b1u) != 0);

             data >>= 1;

           }

         }

private:

         void sendBit(bool oneBit) {

           if (oneBit) {

             SENDER_PORT |= SENDER_DATA;

           } else {

             SENDER_PORT &= ~SENDER_DATA;
```

```
          }

          SENDER_PORT |= SENDER_CLOCK;

          SENDER_PORT &= ~SENDER_CLOCK;

        }

};
```

Oh no! Macros...

Only one configuration can be used, despite the fact this implementation uses a class. To reconfigure the sender, the implementation file has to be changed. You can not just include this library into your project, you have to copy these files.

Performance-wise, this is a great improvement. It only takes 35.6µs to send the `Hello` bytes, which equals 140.4 KB/s. This is 14 times faster than the previous implementation.

Now, we use function templates to improve this code. They will make the sender configuration flexible, without macros.

SenderTemplateFunctions.ino

```
#include "Sender.hpp"

const uint8_t cSpiPortAddr = _SFR_IO_ADDR(PORTB);

const uint8_t cSpiDirAddr = _SFR_IO_ADDR(DDRB);

const uint8_t cSpiDataMask = 0b00100000u;

const uint8_t cSpiClockMask = 0b00010000u;

const char data[] = "Hello World!";

void setup() {

        Sender::initialize<cSpiPortAddr, cSpiDirAddr, cSpiDataMask, cSpiClockMask>
();

}

void loop() {

          Sender::sendByte<cSpiPortAddr, cSpiDataMask, cSpiClockMask>(dataByte);

        }
```

```
}
```

Sender.hpp

```
#pragma once

// Intermediate Example -> This will get better, do not use!

namespace Sender {

template<uint8_t ioPortAddr, uint8_t ioDirAddr, uint8_t dataMask, uint8_t clockMask>

void initialize() {

          _SFR_IO8(ioDirAddr) |= (dataMask|clockMask);

          _SFR_IO8(ioPortAddr) &= ~(dataMask|clockMask);

}

template<uint8_t ioPortAddr, uint8_t dataMask, uint8_t clockMask>

void sendBit(bool oneBit) {

         if (oneBit) {

           _SFR_IO8(ioPortAddr) |= dataMask;

         } else {

           _SFR_IO8(ioPortAddr) &= ~dataMask;

         }

         _SFR_IO8(ioPortAddr) |= clockMask;

         _SFR_IO8(ioPortAddr) &= ~clockMask;

}

template<uint8_t ioPortAddr, uint8_t dataMask, uint8_t clockMask>

void sendByte(uint8_t data) {

         for (uint8_t i = 0; i < 8; ++i) {

           sendBit<ioPortAddr, dataMask, clockMask>((data & 0b1u) != 0);

           data >>= 1;

         }

}
```

```
}
```

This is just an intermediate example, using what we learned so far. It will get better, I promise!

The last example uses function templates to let the compiler generate code for every configuration. **The performance is equal** to the previous solution.

The size of the binary is 532 bytes and *both programs produce the **exact** same firmware.*

The main difference is the flexibility of the template solution.

To send data on multiple channels, you just add additional calls to the main file, as shown in the next example:

```
#include "Sender.hpp"

const uint8_t cSpiPortAddr = _SFR_IO_ADDR(PORTB);

const uint8_t cSpiDirAddr = _SFR_IO_ADDR(DDRB);

const uint8_t cSpiDataMaskA = 0b00100000u;

const uint8_t cSpiClockMaskA = 0b00010000u;

const uint8_t cSpiDataMaskB = 0b00001000u;

const uint8_t cSpiClockMaskB = 0b00000100u;

const uint8_t cSpiDataMaskC = 0b00000010u;

const uint8_t cSpiClockMaskC = 0b00000001u;

const uint8_t cDataSize = 16;

const char cDataA[cDataSize] = "Hello World!   ";

const char cDataB[cDataSize] = "Templates for  ";

const char cDataC[cDataSize] = "flexible code. ";

void setup() {

        Sender::initialize<cSpiPortAddr, cSpiDirAddr, cSpiDataMaskA, cSpiClockMaskA>
();

        Sender::initialize<cSpiPortAddr, cSpiDirAddr, cSpiDataMaskB, cSpiClockMaskB>
();
```

```
        Sender::initialize<cSpiPortAddr, cSpiDirAddr, cSpiDataMaskC, cSpiClockMaskC>
();

}

void loop() {

        for (uint8_t i = 0; i < cDataSize-1; ++i ) {

          Sender::sendByte<cSpiPortAddr, cSpiDataMaskA, cSpiClockMaskA>(cDataA[i]);

          Sender::sendByte<cSpiPortAddr, cSpiDataMaskB, cSpiClockMaskB>(cDataB[i]);

          Sender::sendByte<cSpiPortAddr, cSpiDataMaskC, cSpiClockMaskC>(cDataC[i]);

        }

}
```

Suddenly, the same `Sender` sends on three different SPI channels. The binary is just 654 bytes with the additional generated code.

This is not the End ♫

Do you already feel the template power at your fingertips? This is just the beginning.

The last implementation has several issues:

The code is ugly and hard to use. All these parameters have to be passed for each `sendByte` call. **The code is not portable.** The `_SRF_XXX` macros and IO addresses will just work for a small range of Atmel chips. To use the same code on a SAM D21 chip, we need to rewrite parts of the sender code.

In the next article, I will introduce **template classes** that allow even more powerful things but with less code. We will also fix the sender example and make it easy to use.

Read the next part and many other other articles on my website. They are completely free, without ads.

Code

Repository with All Examples

This is the repository with all example code as Arduino IDE project files.

LuckyResistor / templates-for-embedded 4 1

no description / Read More

Latest commit to the master branch on 28-07-2019 Download as a zip

Repository with All Examples

This is the repository with all example code as Arduino IDE project files.

LuckyResistor / templates-for-embedded 4 1

no description / Read More

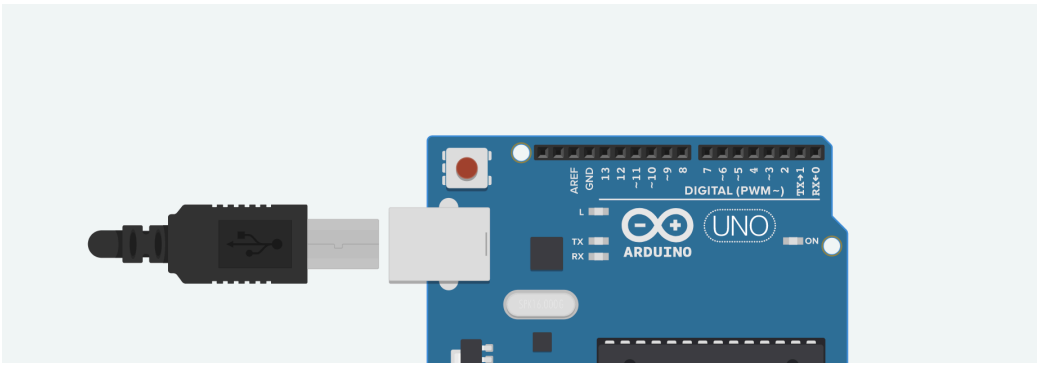
Latest commit to the master branch on 28-07-2019 Download as a zip

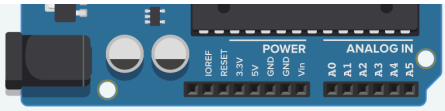
Downloadable files

Test Setup

This is a simple test setup. Just an Arduino Uno, no components required. The tutorial does not require you use an actual Arduino Uno, but it helps to understand the example code.

Test Setup





Comments

Only logged in users can leave comments

LOGIN

Anonymous user

a year ago

Thanks for sharing this information. I work with presentations and would be grateful for any tools or sources that might be helpful in this. I recently found a way to improve my work, and this source helped me https://masterbundles.com/ And it greatly influenced the final result of my work.

Anonymous user

a year ago

Guys, I'm looking for good template for creating my website, so does anybody know something about it? This is really important.

Anonymous user

a year ago

Hello, I would like to understand why an unused variable in the template generates warning messages when compiling, but the implementation code needs the variable, so if

the template variable is partially removed the warning messages no longer appear. Please see the discussion here on the forum: https://forum.arduino.cc/t/how-to-resolve-this-notification-in-instantiation-of-xxxx-required-from-here/1014853/10

A **Anonymous user**

a year ago

Actually, this thing is quite necessary for everyone who works using laptop. Besides, I would like to mention the fact that Use templates in order to create quality and attractive website, and once I made a decision to get more instruments. For instance, I accidentally found this source https://flatsome.info/reviews/11510/swatches where there was information about Swatches' template. Have you ever thought about using it? I think you should try.

A **Anonymous user**

2 years ago

Actually, this thing is quite necessary for everyone who works using laptop. Besides, I would like to mention the fact that Use templates in order to create quality and attractive website, and once I made a decision to get more instruments. For instance, I accidentally found this source https://flatsome.info/reviews/11510/swatches where there was information about Swatches' template. Have you ever thought about using it? I think you should try.

A **Anonymous user**

2 years ago

Guys, I'm looking for good template for creating my website, so does anybody know something about it? This is really important.

A **Anonymous user**

2 years ago

Hello, I would like to understand why an unused variable in the template generates warning messages when compiling, but the implementation code needs the variable, so if the template variable is partially removed the warning messages no longer appear. Please see the discussion here on the forum: https://forum.arduino.cc/t/how-to-resolve-this-notification-in-instantiation-of-xxxx-required-from-here/1014853/10

A **Anonymous user**

2 years ago

Thanks for sharing this information. I work with presentations and would be grateful for any tools or sources that might be helpful in this. I recently found a way to improve my work, and this source helped me https://masterbundles.com/ And it greatly influenced the final result of my work.

MOHAMMEDSHAREEF

5 years ago

good