HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

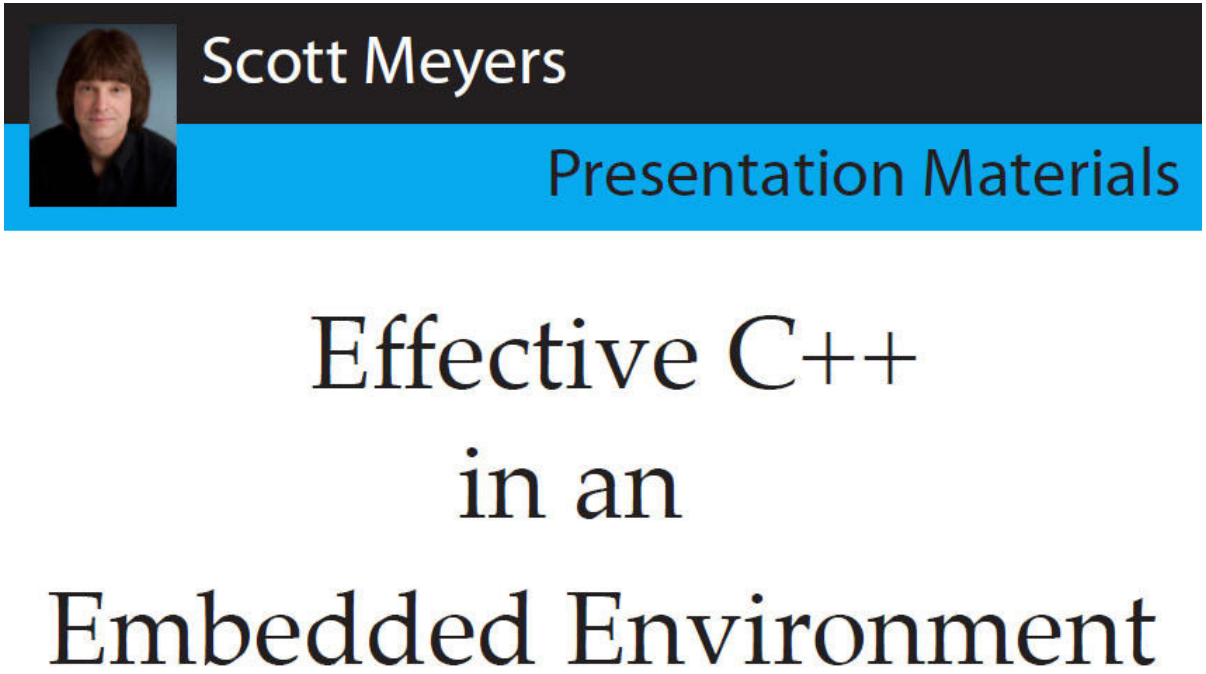
Embedded Software Engineering 2
Modeling Memory Mapped IO (MMIO)

Prof. Reto Bonderer
HSR Hochschule für Technik Rapperswil
reto.bonderer@hsr.ch

Mai 2020

Effective C++ in an Embedded Environment

Die meisten der folgenden Informationen stammen aus einem Seminar von Scott Meyers



Modeling Memory-Mapped IO (MMIO)

Many systems map IO devices to fixed parts of a program's address space.

- Input registers are often separate from output registers.
- Control/status registers are often separate from data registers.
  - Different status register bits convey information such as readiness or whether device interrupts are enabled.

C++ makes it easy to make memory-mapped IO devices look like objects with natural interfaces.

- **At zero cost.**
  - Provided you have a decent compiler :-)

Modeling Memory-Mapped IO

Memory-mapped devices may require special handling, e.g.,

- Atomic reads/writes may require explicit synchronization.
- Individual bits may sometimes be read-only, other times write-only.
- Clearing a bit may require assigning a *1* to it (positive vs. negative logic).
- One control register may control more than one data register.
  - E.g., bits 0-3 are for one data register, bits 4-7 for another.

What follows is a *framework* for modeling memory-mapped IO, not a prescription.

- The framework tells you where to put whatever special handling your devices require.
- It's an elegant and efficient way to define a Hardware Abstraction Layer (HAL)

Grundsätzliches zu MMIO (siehe HAL in EmbSW1)

- Embedded Systems werden sehr intensiv mittels Registern programmiert, d.h. durch Setzen oder Löschen einzelner Bits
- Register könnten mittels Bitfeldern modelliert werden.
  - Nicht portabel
  - Können sehr ineffizient sein (manchmal aber auch sehr effizient)
  - Erlauben nur sehr eingeschränkte Abstraktion
  - Sollten generell vermieden werden
- Register müssen auf eine bestimmte Adresse gemappt werden können
- MMIO Register **müssen** mit **`volatile`** gekennzeichnet werden

Modeling a Control Register

Suppose we have a four-byte control register such that:

- Bit 0 indicates readiness: 1 ⇒ ready, 0 ⇒ not ready.
- Bit 2 indicates whether interrupts are enabled: 1 ⇒ enabled, 0 ⇒ not.

```
enum {bit0 = 0x1, bit1 = 0x2, ... , bit31 = 0x80000000};
class ControlReg
{
public:
  bool ready() const {return (regValue & bit0) == bit0;}
  bool interruptsEnabled() const {return (regValue & bit2) == bit2;}
  void enableInterrupts() {regValue |= bit2;}
  void disableInterrupts() {regValue &= ~bit2;}
private:
  volatile uint32_t regValue; // register data may change outside prog control
};
```

Modeling a Control Register: Notes

- All functions are inline, so their existence should incur no cost.
- `enableInterrupts()` and `disableInterrupts()` use read/modify/write instructions, so they may be subject to race conditions in multithreaded systems.
- We assume that the register is both readable and writable.
  - If it's write-only, you'll need to cache the most recently written value.

Placement new

- Die Aufgabe des Operators new ist nicht primär, Speicher zu allozieren, sondern eine Adresse zurückzugeben auf einen bestimmten Speicherbereich
- Üblicherweise wird new zusammen mit dem Allozieren von Speicher verwendet

- **Placement new** ist eine Variante des new-Operators, die ein Objekt an eine bestimmte Stelle im Speicher legt. Eine Speicherallozierung ist damit nicht verbunden

- Ein durch placement new plaziertes Objekt muss und darf nicht mit `delete` gelöscht werden!

Place a Register at Address Using Placement new

```
// create a ControlReg object at address 0xFFFF0000 and make pcr point to it
ControlReg* const pcr =
  new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;

ControlReg& cr =            // avoid pointer syntax
  *new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;

while (!cr.ready())         // wait until the ready bit is on
{}
cr.enableInterrupts();      // enable device interrupts
if (cr.interruptsEnabled()) ... // if interrupts are enabled
```

Placement new vs. Raw Casts

Our use of placement new calls the default `ControlReg` constructor.

- It's implicit, inline, and empty.
  - It should optimize away (i.e., to zero instructions).
  - **If it doesn't and you care** (or if your compiler doesn't support placement new), consider a reinterpret_cast instead of placement new:

```
ControlReg* const pcr =                          // pointer version
  reinterpret_cast<ControlReg*>(0xFFFF0000);

ControlReg& cr =                                 // reference version
  *reinterpret_cast<ControlReg*>(0xFFFF0000);
```

Placement new vs. Raw Casts (cont'd)

Placement new is typically preferable to a raw `reinterpret_cast`:

- It works if the device object's constructor has work to do.
- A raw cast will behave improperly in that case.

But there are times when `reinterpret_cast` can be superior:

- If placement new isn't optimized to zero instructions (and you care).
- If you want to ROM the address of an IO register *and*
  - Your compiler will ROM the result of a `reinterpret_cast` *and*
  - It won't ROM the result of a use of placement new.

Placing Objects via Compiler Extensions

With some compilers/linkers, there is another alternative:

- Use a compiler extension to place an object at a specific address.

Examples:

- Altium Tasking compilers offer this kind of syntax:

```
ControlReg cr __at(0xFFFF0000);
```

- The Wind River Diab compiler offers this:

```
#pragma section MMIO address=0xFFFF0000
#pragma use_section MMIO cr
ControlReg cr;
```

Such extensions may impose restrictions:

- E.g., such manually-placed objects may have to be POD types.
- It's not portable

Payoffs:

- No need to access MMIO objects indirectly through a pointer.
- No need to allocate space for a pointer to each MMIO object.

Placing Objects via Linker Commands

Linkers may support specification of where objects are to be placed:

- C++ source code is “normal.”
  - No object placement information is present.

```
ControlReg cr;
```

- Linker scripts map C++ objects to memory locations, often by:
  - Mapping objects to sections.
    - The linker sees only *mangled* names.
  - Mapping sections to address ranges.

Result is more portable C++ code.

- Platform-specific addresses mentioned only in linker scripts.

Modeling an Output Device, Take 1

An `uint32_t`-sized output device register can be modeled as a `ControlReg` object bundled with the `uint32_t`-sized data it controls:

```
class OutputDevice1
{
public:
  OutputDevice1(uint32_t controlAddr,
                uint32_t dataAddr);          // see next page
  ControlReg& control() {return *pcr;}       // get ControlReg
  void write(uint32_t value) {*pd = value;}  // write data to device
private:
  ControlReg* const pcr;                  // ptr to ControlReg; note const
  volatile uint32_t* const pd;            // ptr to data; note const and volatile
};
```

Modeling an Output Device, Take 1

The constructor makes `pcr` and `pd` point to the correct addresses:

```
OutputDevice1::OutputDevice1(uint32_t controlAddr,
                             uint32_t dataAddr)
  : pcr(new (reinterpret_cast<void*>(controlAddr)) ControlReg),
    pd(new (reinterpret_cast<void*>(dataAddr)) uint32_t)
{}
```

Clients use the class like this:

```
OutputDevice1 od(0xFFFF0000,        // ctrl reg addr
                 0xFFFF0004);       // data reg addr
uint32_t x;
...
while (!od.control().ready())       // wait until the ready bit is on
{}
od.write(x);                        // write x to od
...
```

Modeling an Output Device, Take 2

MMIO addresses are compile-time constants, so it shouldn't be necessary to store them as data members like `pcr` and pd.

A template with non-type parameters makes it easy not to:

```
template<uint32_t controlAddr, uint32_t dataAddr>
class OutputDevice2
{
public:
                                    // ctor now gone
  static ControlReg& control()
  {return *reinterpret_cast<ControlReg*>(controlAddr);}
  static void write(uint32_t value)
  {*reinterpret_cast<volatile uint32_t*>(dataAddr) = value;}
                                    // data members now gone
};
```

OutputDevice2 uses static member functions to avoid passing *`this`* pointers.

Modeling an Output Device, Take 2 (cont'd)

This example assumes the `ControlReg` constructor/destructor do nothing.

- Otherwise `OutputDevice2` will need a constructor/destructor that call them (e.g., via placement `new`).

```
template<uint32_t controlAddr, uint32_t dataAddr>
class OutputDevice2
{
public:
  OutputDevice2()
  {new (reinterpret_cast<ControlReg*>(controlAddr)) ControlReg;}
  ...
};
```

Such initialization/cleanup must occur only once!

- Problematic if multiple OutputDevice2 objects exist for a single hardware device.
  - Use Singleton to prevent multiple instantiations?
  - Use static alreadyInitialized/alreadyCleanedup flags?

Modeling an Output Device, Take 2 (cont'd)

Client code looks almost the same as before:

```
OutputDevice2<0xFFFF0000, 0xFFFF0004> od;
uint32_t x;
...
while (!od.control().ready())  // wait until the ready bit is on
{}
od.write(x);                   // write x to od
...
```

Advantages of this approach:

- OutputDevice2 objects are smaller than OutputDevice1 objects.
- OutputDevice2 code may also be smaller/faster than OutputDevice1 code.
  - No need to go indirect via a this pointer.

Modeling an Output Device, Take 3

If, as in this case, the ***control and data registers are in contiguous memory***, you can use a third design:

- Create objects *directly* on the MMIO locations:

```
class OutputDevice3
{
public:
  ControlReg& control() {return cr;}
  void write(uint32_t value) {data = value;}
private:
  OutputDevice3(const OutputDevice3&); // prevent copying
  ControlReg cr;
  volatile uint32_t data;
};
```

- Have clients use placement new (or bare reinterpret_cast) themselves:

```
// create OutputDevice3 object at address 0xFFFF0000
OutputDevice3& od = *new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;
```

Modeling Hardware Directly

There are thus two kinds of classes for modeling memory-mapped IO.

One kind models hardware *directly*.

- Objects of such classes are created by clients at specific addresses.
  - Via placement new or reinterpret_cast.
- They contain only non-static data that maps to MMIO registers:

```
class OutputDevice3
{                          // class designed to be instantiated at MMIO addresses
public:
...
private:
  OutputDevice3(const OutputDevice3&);
  ControlReg cr;           // data members map directly to MMIO device registers
  volatile uint32_t data;
};
```

- Static data is okay.
- They never contain virtual functions (virtual functions leads to a vptr *somewhere* within each object)

Modeling Hardware Indirectly

The other kind models hardware *indirectly*.

- Objects of such classes are *not* created at specific addresses.
  - Clients pass MMIO addresses as template or constructor arguments.
- They may contain “extra” data members.
  - I.e., that don’t correspond to MMIO device registers.

```
class OutputDevice1
{
...
private:
  ControlReg* const pcr;          // data members that don’t map
  volatile uint32_t* const pd;    // to MMIO device registers
  uint32_t lastValueWritten;      // useful for write-only regs
};
```

Modeling Hardware Indirectly (cont'd)

They may contain virtual functions:

```
class DeviceBase {
public:
  virtual void reset() = 0;
  ...
};
class OutputDevice1: public DeviceBase {
public:
  void reset() override;
  ...
};
template<uint32_t controlAddr, uint32_t dataAddr>
class OutputDevice2: public DeviceBase {
public:
  void reset() override;
  ...
};
...                          // continued on next slide…
```

Modeling Hardware Indirectly (cont'd)

```
OutputDevice1 od1a(0xFFFF0000, 0xFFFF0004);
OutputDevice1 od1b(0xFFFF0010, 0xFFFF0014);
...
OutputDevice2<0xEEEE0000, 0xEEEE0010> od2a;
OutputDevice2<0xEEEE0020, 0xEEEE0040> od2b;
...
DeviceBase* registers[] = {&od1a, &od1b, …, &od2a, &od2b, …};
const std::size_t numRegisters = sizeof(registers)/sizeof(registers[0]);
...
for (std::size_t i = 0; i < numRegisters; ++i)
{                          // reset all registers in system
  registers[i]->reset();
}
```

Modeling Hardware Indirectly (cont'd)

Indirect modeling more expensive than direct modeling:

- Memory for “extra” data members (if present).
- Indirection to get from the object to the register(s) (if needed).

It’s also more flexible:

- May add other data members.
- May declare virtual functions.

Preventing Likely Client Errors

Classes that model hardware directly can easily be misused, e.g.:

- Clients might instantiate them at non-MMIO addresses.
- Clients who use placement new might think they need to call delete.
  - They don’t, though they may need to manually call the destructor.

To prevent such errors, consider declaring the destructor private.

Preventing Likely Client Errors (cont'd)

```
class OutputDevice3    // class modeling hardware directly
{                      // prevents some kinds of client errors
  ...
private:
  ~OutputDevice3() {}
  ControlReg cr;
  volatile uint32_t data;
};

OutputDevice3 d; // error! implicit destructor invocation. (We want to prevent
                 // MMIO objects from being placed on the stack.)
OutputDevice3* pd =
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3; // fine
...
delete pd;       // error! another implicit destructor invocation
```

Preventing Likely Client Errors: Basic Idea

A fundamental design goal is that *design violations should not compile*.

Generalizing via Templates

Devices are likely to vary in several ways:

- Number of bits in each register.
- Which bits correspond to ready and interrupt status, etc.

Templates make it easy to handle such variability:

```
template<typename RegType, RegType readyBitMask, RegType interruptBitMask>
class ControlReg {
public:
  bool ready() const {
    return (regValue & readyBitMask) == readyBitMask;}
  bool interruptsEnabled() const {
    return (regValue & interruptBitMask) == interruptBitMask;}
  void enableInterrupts() {regValue |= interruptBitMask;}
  void disableInterrupts() {regValue &= ~interruptBitMask;}
private:
  volatile RegType regValue;
};
```

Generalizing via Templates: Usage

```
// CRui03 is a control register the size of an uint32_t where
// bit 0 is the ready bit and bit 3 is the interrupt bit

typedef ControlReg<uint32_t, bit0, bit3> CRui03;

CRui03& cr1 = *new (reinterpret_cast<void*>(0xFFFF0000)) CRui03;
... // use cr1


// CRuc15 is a control register the size of an uint8_t where
// bit 1 is the ready bit and bit 5 is the interrupt bit

typedef ControlReg<uint8_t, bit1, bit5> CRuc15;

CRuc15& cr2 = *new (reinterpret_cast<void*>(0xFFFF0010)) CRuc15;
... // use cr2
```

Summary: Modeling Memory-Mapped IO

C++ tools you'll probably want to use:

- Classes
- Class templates (with both type and non-type parameters)
- Inline functions
- Placement new and reinterpret_cast
- const pointers
- volatile memory
- References
- Private member functions, e.g., copy constructor, destructor.