Simulate Memory Based Device Control By Using Policy Based Design

2012-09-25

Andreas Hűnnebeck <ah@bruker.de>
Bruker BioSpin, Department NMRENS
76287 Rheinstetten, Germany
Phone: +49 (721) 51 61-6444

In this document I demonstrate how policy based design eases the implementation of memory based device control. The resulting control code can be used without performance penalties both on the embedded hardware and in a simulation environment. Several policies for simulation are shown which allow debugging and testing on hardware level within the simulation environment.

Contents

1. Introduction

Memory based device control is a typical feature of embedded systems. So called *device codes* are addresses of memory locations where a read or write executes an action in the connected hardware, like in the following example:

```
const uint32_t SEQUENCER = 0x60200004;          // device code for sequencer control
...
*((volatile uint16_t*)SEQUENCER) = 0x1;         // start sequencer
...
uint16_t tmp = *(volatile uint16_t*)SEQUENCER;  // stop sequencer
```

Testing such code within a simulation environment is impossible, because one cannot access the memory at those hard coded addresses. It is either none existent or belongs to other processes. Typical solutions use conditional compilation and insert different code for debugging and testing, or none at all, as in this example:

```
const uint32_t SEQUENCER = 0x60200004;          // device code for sequencer control
...
#ifndef SIMUL
*((volatile uint16_t*)SEQUENCER) = 0x1;         // start sequencer
#endif
...
#ifndef SIMUL
uint16_t tmp = *(volatile uint16_t*)SEQUENCER;  // stop sequencer
#endif
```

2. Use a Template Class With A Policy

Policy Based Design uses template classes and functions to specify at compile time how a software component is built. A common approach is to create a template class with member functions which forward the real task to member functions of so called *policy classes*.

2.1 Create a Template Class Using a Policy For Memory Access

In our case we need a template class with `read()` and `write()` member functions which forward the actual memory read and write tasks to a policy class. Let's call the template class `MemAccess` and the policy class `AccessPolicy:`

```
// MemAccess uses an AccessPolicy to read and write to a specific memory address:
// it implements both a read() and a write() function inline which forward the task
// to AccessPolicy.
template <typename AccessPolicy>
class MemAccess : public AccessPolicy
{
  public:
     MemAccess(): AccessPolicy() {}
     uint16_t read(uint32_t adr)                { return AccessPolicy::template read(adr); }
     void     write(uint32_t adr, uint16_t val) { AccessPolicy::template write(adr, val); }
};
```

`AccessPolicy` can be any `class` or `struct` which implements the matching `read()` and `write()` calls.

2.2 Create a Policy For Physical Access Of Memory

A policy which implements physical access (as needed on the embedded hardware) just reads and writes at the specified address:

```
struct PhysicalMemAccess
{
     static uint16_t read(uint32_t adr)                { return *(volatile uint16_t*)adr; }
     static void     write(uint32_t adr, uint16_t val) { *(volatile uint16_t*)adr = val; }
};
```

2.3 Create A Policy Which Does Nothing

A different AccessPolicy is needed for the simulation environment. The simplest of these policies does nothing at all:

```
struct NoMemAccess
{
     static uint16_t read(uint32_t /* adr */) { return 0xffff; }
     static void     write(uint32_t /* adr */, uint16_t /* val */) {}
};
```

2.4 Use The Template Class With Policies

Now we can rewrite the example code from chapter 1 for both embedded hardware and the simulation with only one #ifdef clause:

```
#ifdef SIMUL
    typedef NoMemAccess        MemAccessPolicy;
#else
    typedef PhysicalMemAccess  MemAccessPolicy;
#endif

MemAccess<MemAccessPolicy> memory;
const uint32_t SEQUENCER = 0x60200004;
...
memory.write(SEQUENCER, 0x1);            // start sequencer
...
uint16_t tmp = memory.read(SEQUENCER);   // stop sequencer
```

A very important point to notice is that there is no performance penalty involved because all functions are inline. You can verify this by checking the assembler output of the code for the embedded device.

2.5 Other Policies for Simulation

2.5.1 A Policy To Simulate Embedded Memory

An AccessPolicy which emulates the memory of the embedded device could use an array of the size of the real memory where the read() and write() calls operate on. A more memory conservative approach is to use an associative array like std::map supplied by the STL. The memory address serves as key and the memory content is the value stored in the map.

```
// simulate a memory with hard coded addresses
class EmulateMemory
{
  public:
    EmulateMemory() : memMap() {}
    uint16_t read(uint32_t adr)                { return memMap[adr]; }
    void     write(uint32_t adr, uint16_t val) { memMap[adr] = val; }
 private:
    std::map<uint32_t, uint16_t> memMap;
};
```

If necessary the constructor of EmulateMemory could preload the map with specific values at specific addresses so that read() calls return useful values.

It is tempting to add debugging code to EmulateMemory, e.g. a printf() call in read() and write(). However, in policy based design you do not mix functionalities which are independent of each other. Creating debug information is independent of accessing memory therefore it is implemented by its own policy.

3. Use a Template Class With Two Policies

To get debug information we add a second policy class called LogPolicy which logs the read() and write() calls:

```
// MemAccess uses an AccessPolicy to read and write to a specific memory address,
// and a LogPolicy to log those read and write accesses.
template <typename AccessPolicy, typename LogPolicy>
class MemAccess : public AccessPolicy, LogPolicy
{
  public:
    MemAccess(): AccessPolicy(), LogPolicy {}
    uint16_t read(uint32_t adr)
    {
        uint16_t val = AccessPolicy::template read(adr);
        LogPolicy::template logRead(adr, val);
        return val;
    }
    void write(uint32_t adr, uint16_t val)
    {
        AccessPolicy::template write(adr, val);
        LogPolicy::template logWrite(adr, val);
    }
};
```

LogPolicy can be any class or struct which implements the matching logRead() and logWrite() calls.

3.1 Create Policies for Logging

3.1.1 A Policy Which Does Nothing

For the embedded device we need a LogPolicy which does nothing:

```
struct LogNothing
{
    static void logRead(uint32_t /* adr */, uint16_t /* val */) {}
    static void logWrite(uint32_t /* adr */, uint16_t /* val */) {}
};
```

3.1.2 A Policy Which Logs All Accesses

If you want to know precisely which device codes have been accessed in which order you can log all read and write accesses in a dynamic array, e.g. the std::vector supplied by the STL:

```
struct MemLogAll
{
    enum Type { READ, WRITE };                                     // type of access
    struct LogEntry { Type type; uint32_t adr; uint16_t val; }     // access information

    MemLogAll(): logs() {}
    void logRead(uint32_t adr, uint16_t val)  { logs.push_back(LogEntry(READ, adr, val)); }
    void logWrite(uint32_t adr, uint16_t val) { logs.push_back(LogEntry(WRITE, adr, val)); }

    std::vector<LogEntry> logs;
};
```

After the task to be debugged is finished you can print out the whole content of logs and check it for correctness. You can also use it for unit tests:

1. create a vector of LogEntry and fill it with access entries expected by the test
2. clear logs in MemLogAll
3. run the test
4. compare the contents of logs and the vector created in step 1.

3.2 Use The Template Class With Both Policy Types

To use both policy types we extend the example code from chapter 2:

```
#ifdef SIMUL
    typedef EmulateMemory      MemAccessPolicy;
    typedef MemLogAll          LogPolicy;
#else
    typedef PhysicalMemAccess  MemAccessPolicy;
    typedef LogNothing         LogPolicy;
#endif

MemAccess<MemAccessPolicy, LogPolicy> memory;
const uint32_t SEQUENCER = 0x60200004;
...
memory.write(SEQUENCER, 0x1);            // start sequencer
...
uint16_t tmp = memory.read(SEQUENCER);   // stop sequencer
```

Again there is no performance penalty involved because all functions are inline. The really important advantages are:

1. You can change the behaviour of the program by simply changing some `typedef`s.
2. We have implemented 3 access policies and 2 log policies, so we can create 3 * 2 = 6 different implementations of `MemAccess.`
3. Even if `MemAccess` and the policies would be defined in a library you can still control its behaviour by using your own locally defined policies.

   You could for example use a specially crafted `LogPolicy` on the embedded hardware, e.g. one which logs the last read and write access into two specific memory addresses which can be accessed from outside. This is not possible if you put the logging code into the `AccessPolicy` for simulation.
4. A policy is usually fairly simple and can be tested separately.

This is the power of policy based design.

4. Support Other Types

4.1 Support Any Value Type

The implementation of `read()` and `write()` in the template class and the policies shown above has very limited usability, as it supports only 16 Bit value types (`uint16_t`). You could overwrite the `write()` member function for other types as well, but this does not work for the `read()` member function because you cannot overwrite functions which differ by their return type only. Again, templates come to the rescue:

```
template <typename AccessPolicy, typename LogPolicy>
class MemAccess : public AccessPolicy, public LogPolicy
{
  public:
    MemAccess(): AccessPolicy(), LogPolicy() {}

    template <typename ValueT> ValueT read(uint32_t adr)
    {
        ValueT val = AccessPolicy::template read<ValueT>(adr);
        LogPolicy::template logRead<ValueT>(adr, val);
        return val;
    }
    template <typename ValueT> void write(uint32_t adr, ValueT val)
    {
        AccessPolicy::template write<ValueT> (adr, val);
        LogPolicy::template logWrite<ValueT> (adr, val);
    }
};
```

The `read()` and `write()` member functions in `MemAccess` are now template functions. Therefore the corresponding member function in the policies `AccessPolicy` and `LogPolicy` must be template functions as well, like in these examples:

```
struct PhysicalMemAccess
{
    template <typename ValueT>
    static void write(uint32_t adr, ValueT val) { *(volatile ValueT*)adr = val; }

    template <typename ValueT>
    static ValueT read(uint32_t adr) { return *(volatile ValueT*)adr; }
};

struct LogNothing
{
    template <typename ValueT>
    static void logWrite(uint32_t /* adr */, const ValueT& /* val */) {}

    template <typename ValueT>
    static void logRead(uint32_t /* adr */, const ValueT& /* val */) {}
};
```

We must also change `EmulateMemory` because the `std::map` must support read and write with arbitrary value size:

```
class EmulateMemory
{
  public:
     EmulateMemory() : memMap() {}
     template <typename ValueT> ValueT read(uint32_t adr);
     template <typename ValueT> void write(uint32_t adr, const ValueT& val);
  private:
     std::map<uint32_t, uint32_t> memMap; // hardware is 32 bit wide
};
```

The implementation of `read()` and `write()` in `EmulateMemory` is fairly complex so refer to the sources in the appendix 6.1 on page 9, please.

4.2 Support Any Memory Type

The implementation of `read()` and `write()` in the template class and the policies of chapter 4.1 is limited to a memory where the width of the address is 32 bits (`uint32_t`). Since embedded devices may have other memory widths (8, 16 or even 64 bits) we need another template parameter `PointerT` for the type of pointer[1], which replaces `uint32_t` in the template class `MemAccess`:

```
template <typename PointerT, typename AccessPolicy, typename LogPolicy>
class MemAccess : public AccessPolicy, public LogPolicy
{
  public:
    MemAccess(): AccessPolicy(), LogPolicy() {}
    template <typename ValueT> ValueT read(PointerT adr)
    {
        ValueT val = AccessPolicy::template read<ValueT>(adr);
        LogPolicy::template logRead<ValueT>(adr, val);
        return val;
    }
    template <typename ValueT> void write(PointerT adr, ValueT val)
    {
        AccessPolicy::template write<ValueT> (adr, val);
        LogPolicy::template logWrite<ValueT> (adr, val);
    }
};
```

The corresponding member function in the policies `AccessPolicy` and `LogPolicy` must be template functions as well, like in these examples:

```
struct PhysicalMemAccess
{
    // no PointerT template parameter needed since uintptr_t is always correct
    template <typename ValueT>
    static void write(uintptr_t adr, ValueT val) { *(volatile ValueT*)adr = val; }

    // no PointerT template parameter needed since uintptr_t is always correct
    template <typename ValueT>
    static ValueT read(uintptr_t adr) { return *(volatile ValueT*)adr; }
};

struct LogNothing
{
    template <typename ValueT, typename PointerT>
    static void logWrite(PointerT /* adr */, const ValueT& /* val */) {}

    template <typename ValueT, typename PointerT>
    static void logRead(PointerT /* adr */, const ValueT& /* val */) {}
};
```

Note that in `PhysicalMemAccess` we need not specify the `PointerT` type because type `uintptr_t` is always the correct type for a native pointer.

Again we must also change `EmulateMemory` because key type and value type of the `std::map` need to be chosen depending on the bit width of the memory:

```
template <typename PointerT, typename PointerValueT>
class EmulateMemory
{
 public:
    EmulateMemory() : memMap() {}
    template <typename ValueT> ValueT read(PointerT adr);
    template <typename ValueT> void write(PointerT adr, const ValueT& val);
 private:
    std::map<PointerT, PointerValueT> memMap;
};
```

The implementation of `read()` and `write()` in `EmulateMemory` is fairly complex so refer to the sources in the appendix 6.2 on page 9, please.

1 `PointerT` is the type of integer which can hold a pointer.

If our embedded hardware has 32 bit memory and can access it with 16 bit read/write we can rewrite the example from chapter 3 as follows:

```
#ifdef SIMUL
    typedef EmulateMemory<uint32_t, uint16_t>  MemAccessPolicy;
    typedef MemLogAll                          LogPolicy;
#else
    typedef PhysicalMemAccess                  MemAccessPolicy;
    typedef LogNothing                         LogPolicy;
#endif

MemAccess<uint32_t, MemAccessPolicy, LogPolicy> memory;
const uint32_t SEQUENCER = 0x60200004;
...
memory.write<uint16_t>(SEQUENCER, 0x1);            // start sequencer
...
uint16_t tmp = memory.read<uint16_t>(SEQUENCER);   // stop sequencer
```

5. Conclusion

Policy Based Design is an important technique for testing and debugging code which controls hardware via memory based device control:

- Different policies allow fine control of the data to be logged for debugging and test purposes, and how precise the hardware is emulated. These policies are fully controlled by the user, compared to typical libraries where the functionality is frozen.
- The code compiled for the embedded hardware does not suffer from code bloat or performance penalties.
- The code is not cluttered by conditional compilation.

6. Appendix

6.1 Source Code of EmulateMemory for PointerT

Source code of `EmulateMemory::read()` for `PointerT`:

```
template <typename ValueT>
ValueT EmulateMemory::read(uint32_t adr)
{
    // create a buffer which is filled by the read:
    uint8_t val_array[sizeof(ValueT) + sizeof(uintptr_t)];

    // fill the buffer by the read:
    uint32_t* cur_val_ptr = reinterpret_cast<uint32_t*>(val_array);
    uint32_t cur_adr = adr;
    for (uint32_t i = 0; i < sizeof(ValueT); i += 4, cur_adr += 4, ++cur_val_ptr)
        *cur_val_ptr = memMap[cur_adr];

    // copy the buffer into the real value:
    ValueT* valp = reinterpret_cast<ValueT*>(&val_array[0]);
    return *valp;
}
```

Source code of `EmulateMemory::write()` for `PointerT`:

```
template <typename ValueT>
void EmulateMemory::write(uint32_t adr, const ValueT& val)
{
    // simple implementation, should read buffer, manipulate it and write back
    const uint32_t* cur_val_ptr = reinterpret_cast<const uint32_t*>(&val);
    uint32_t cur_adr = adr;
    for (uint32_t i = 0; i < sizeof(val); i += 4, cur_adr += 4, ++cur_val_ptr)
        memMap[cur_adr] = *cur_val_ptr;
}
```

6.2 Source Code of EmulateMemory for PointerT and PointerValueT

Source code of `EmulateMemory::read()` for `PointerT` and `PointerValueT`:

```
template <typename PointerT, typename PointerValueT>
template <typename ValueT>
ValueT EmulateMemory::read(PointerT adr)
{
    if (adr & (sizeof(PointerValueT) - 1)) throw "parameter 'adr' is not aligned";

    // create a buffer which is filled by the read:
    // - reserve up to 4/8 more bytes because we read always 4/8 bytes
    uint8_t val_array[sizeof(ValueT) + sizeof(uintptr_t)];

    // fill the buffer by the read:
    PointerValueT* cur_val_ptr = reinterpret_cast<PointerValueT*>(val_array);
    PointerT cur_adr = adr;
    for (uint32_t i = 0;
         i < sizeof(ValueT);
         i += sizeof(PointerValueT), cur_adr += sizeof(PointerValueT), ++cur_val_ptr)
    {
        *cur_val_ptr = memMap[cur_adr];
    }

    // copy the buffer into the real value
    // - casting a ValueT pointer to the begin of val_array does not work!
    ValueT val;
    uint8_t* val8_ptr = reinterpret_cast<uint8_t*>(&val);
    for (uint32_t i = 0; i < sizeof(val); ++i)
        *val8_ptr++ = val_array[i];
    return val;
}
```

Source code of `EmulateMemory::write()` for `PointerT` and `PointerValueT`:

```
template <typename PointerT, typename PointerValueT>
template <typename ValueT>
void EmulateMemory::write(PointerT adr, const ValueT& val)
{
    if (adr & (sizeof(PointerValueT) - 1)) throw "parameter 'adr' is not aligned";

    const PointerValueT* cur_val_ptr = reinterpret_cast<const PointerValueT*>(&val);
    PointerT cur_adr = adr;
    for (uint32_t i = 0;
         i < sizeof(val);
         i += sizeof(PointerValueT), cur_adr += sizeof(PointerValueT), ++cur_val_ptr)
    {
        memMap[cur_adr] = *cur_val_ptr;
    }
}
```