One approach to using hardware registers in C++

Martin Moene has been programming professionally since 1983, mostly in C++. He has a background in electronics engineering, and most programming revolves around instrument control, image processing, crunching numbers and sometimes administrivia. Martin can be contacted at m.j.moene@eld.physics.LeidenUniv.nl.

Software that accesses hardware registers is not always written as clearly as one would like. A cause for this may be the assumption that using an abstraction for the register degrades performance too much. Also such code often lacks good support for testing, which is aggravated by the write-only property of many registers that complicates verifying if the software operates correctly. This article presents the approach that we use to addresses these issues in our software for scanning probe microscopy.

Scanning Probe Microscopy

To form an idea of a scanning probe microscope [SPM], you may recall the old stereo vinyl-record player with its needle picking up small height variations of the record groove. Now imagine the needle out the groove, making a scanning movement in two directions over a part of the record's surface, somewhat like the head of an inkjet printer scanning over a sheet of paper. This resembles the *scanning probe* that scans a sample and probes its height variations with a tip. Shrink it to the atomic scale of nanometres and it's called a *microscope*. Figure 1 illustrates the scanning movements.

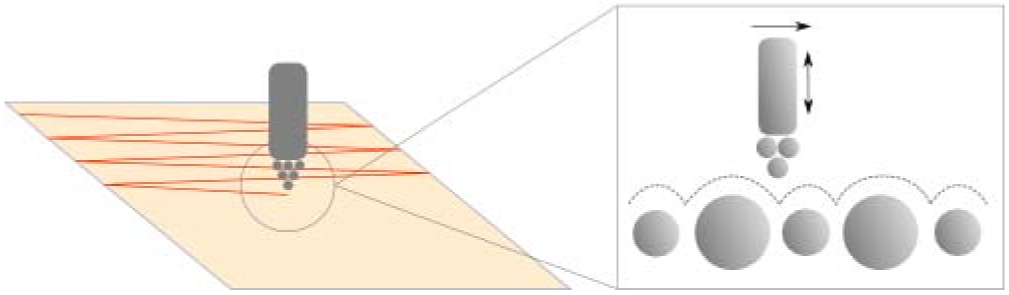

Figure 1: Scanning a surface at atomic scale.

The seeing actually is more like feeling and one of the sensing methods uses the tunnel current [TC] that flows between tip and sample for that purpose. This is called scanning tunneling microscopy (STM). Another type of scanning probe microscopy is atomic force microscopy (AFM), where the sensing method relies on interaction forces between tip and sample if they are close. Before scanning a material's surface, the tiny tip is brought towards the area of interest in a delicate process that's called *approach*.

The strength of the measured interaction between tip and surface is plotted against the scan coordinates in an intensity graph that then represents a view of the surface's topography.

Further, the Leiden Interface Physics [IP] group has designed and built custom electronics to perform scanning probe microscopy measurements as fast as possible. Material with a very even surface allows the recording of movies with up to 80 images of 128x128 pixels squared per second, while still obtaining atomic resolution. With these and several other techniques in place, changes on the surface of a material can be 'videoed' while for example different kinds of gas are flowed over the surface in succession, leading to new discoveries [Frenken05].

In control

The video-rate SPM controller used for these measurements consists of a rack with several modules or cards. There are two computer buses in the rack, one called STM-bus and the other ADC-bus. STM-bus is a bit of a misnomer, as the controller is not limited to STM measurements. The cards connected to the STM-bus are used to generate the signals to perform the tip (sample) scanning movement and the timing of the measurements. The ADC-bus handles the signals measured by several Analog to Digital Converters (ADCs) on that bus. While scanning, the measured values are transfered to the computer under [DMA].

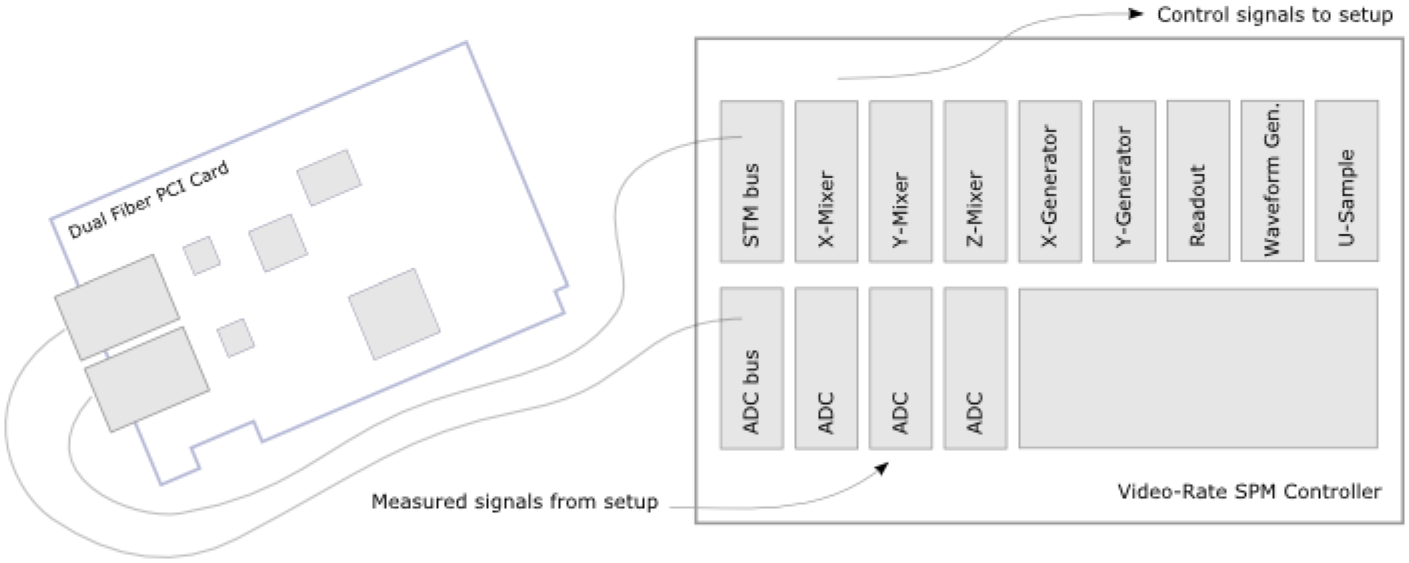

Figure 2: Computer interface card with SPM contoller rack.

Each bus has a bus controller card that is connected via a pair of glass fibers to a PCI interface card in the computer that controls the measurement (Figure 2). A spin-off called Leiden Probe Microscopy [LPM] now produces and markets the SPM controller.

At the same time and as part of the project, C++ software has been developed to interface with the electronics and to perform and analyse STM and AFM measurements.

Control fades

As research objectives change, and new insights in research methods arise, the software has to change. However the software is showing its age and it is becoming increasingly difficult to adapt: it incurred technical debt [Fowler09]. In its current form, the software is also not very well suited to use the SPM controller for measurements outside the field of scanning probe microscopy. Eventually we decided to redevelop the parts of the software that interface with the electronics and that provide the basic scanning probe microscopy functionality.

Regaining weight

As the existing software has no supporting unit tests --[Feathers04] calls this legacy software-- and the gap between it and the desired situation was quite large, we decided not to refactor [Fowler99] the existing code but instead completely redevelop it. We also took the chance to move from Microsoft's Visual C++ version 6 to version 8, which later can be replaced more easily with an even newer version. Where it supports our needs well, we decided to prefer to use [Boost] libraries over other possible libraries. This guideline helped us to choose Boost.Test over for example [GTest] for unit testing.

In the remainder of the article we'll look at our approach to development and testing of the software that controls the electronics with an emphasis on accessing hardware registers and testing the bits and bytes that eventually will flow through them.

While writing this article I encountered "A Technique for Register Access in C++", by Pete Goodliffe, which has the following introductory sentence: *This article discusses a C++ scheme for accessing hardware registers in an optimal way* [Goodliffe05]. It contains a nice introduction to hardware registers and how they are accessed. It's emphasis on the limitations of embedded software is not a concern here.

Register diversity

A hardware register is a kind of memory element, although its implementation may differ from memory used for temporary storage in a computer [HR]. The key properties of a register in this discussion are its data and address width and the method used to access it. Common register access methods are memory-mapped I/O, port-mapped I/O and bus-separated access. Further it is quite common that information written to a register cannot be read-back from it, complicating read/modify/write operations and testing [Roberts]. And for most registers there isn't an easy way to electronically check the result of how we configure it.

A rough analysis was done of the functions of the various cards involved and of their hardware registers' properties. We counted 34 registers, of which 13 use simple word-wide access. The remaining 21 registers are control and status registers and have multiple functions. As much as possible, we would like to be able to use non-related functions within a single register separately from each other. On the other hand sometimes the microscope or domain behaviour requires that what may be disparate register functions or even disparate functions in separate registers, must change in concert.

There are memory-mapped registers on the PCI interface card and registers on the cards in the SPM rack that are accessed via the PCI interface card. The registers on the PCI card are 32-bit wide and have a 32-bit address, whereas the registers on the SPM cards have a 16-bit data type and an 8-bit address. It would be nice if we can use the same register abstraction for the registers on the PCI interface card and the registers on the cards that are accessed via that PCI interface card.

In our search for an approach that works, these are the guiding ideas [Flexible]:

- Use: provide a register abstraction with useful (bitwise) operations such as bittest, bitset and masked variants thereof and use it for all register types.
- Test: make register access testable and actually test it via automated unit tests.
- Access policy: separate the actual register access method from the register abstraction.
- Performance: ensure register access speed that is comparable to pointer-based register access for memory-mapped registers.

Show, don't (just) tell

What the test environment should provide is simple. Initially, while we develop the code, the test environment is allowed to relax [TeX] and to just report the register access that it sees. Thus we can compare register access with the manual description and existing code, reason about it and easily spot and understand any errors we make. Gaining trust in what we wrote, we add register read and write expectations that specify the programmed behaviour and ensure that it is tested (not relaxing anymore). With development completed, visual feedback on the register operations is turned off and only a failing test case may draw our attention.

Although the way of working resembles test driven development [TDD], I merely regard it as visual inspection driven implementation (VIDI) or to overload the term, probe driven development (PDD). Seeing what actually happens to the register first helps to implement the required behaviour correctly.

Listing 1 presents some successfully verified test output that shows the register interaction for a call to the function `read(address)` of the PCI interface card in the computer to obtain a value from a card in the SPM controller.

Listing 1

```
prompt>test --log_level=message --run_test=*/*Stm*/*Read*
Running 1 test case...
Inspect> testThatReadUsesRightRegisterAccessSequence:
Inspect>   1: 0x0044 <-- 0x0001  0b0000000000000001  1  | clear fifo data
Inspect>   2: 0x0044 <-- 0x0000  0b0000000000000000  0  |   is available flag
Inspect>   3: 0x0044 --> 0x0002  0b0000000000000010  2  | wait for write
Inspect>   4: 0x0044 --> 0x0000  0b0000000000000000  0  |   fifo not full
Inspect>   5: 0x0060 <-- 0x82aa0000  0b10000010101010100000000000000000  2192179200     |
write read command
Inspect>   6: 0x0044 --> 0x0002  0b0000000000000010  2  | wait for address
Inspect>   7: 0x0044 --> 0x0002  0b0000000000000010  2  |   to become
Inspect>   8: 0x0044 --> 0x0000  0b0000000000000000  0  |     accepted
Inspect>   9: 0x0044 --> 0x0000  0b0000000000000000  0  | wait for data to
Inspect>  10: 0x0044 --> 0x0001  0b0000000000000001  1  |   become available
Inspect>  11: 0x0060 --> 0x3355  0b0011001101010101  13141      | note host to fiber byte
swap
*** No errors detected
```

At the left of the arrow is the address to read from (-->) or write to (<--), at its right is the register content in hexadecimal, binary and decimal notation. At item 5, in 0x82aa0000, 0x82 is the STM-bus' read command, 0xaa the card address to read from and 0x5533 at item 11 is the value obtained from the card by the simulated read. At the far right, remarks explain what should be happening. Note the explicative name of the test: testThatReadUsesRightRegisterAccessSequence [Henney09].

Listing 2 shows a failing test where the value written to the register (0x246) is different from the expected one (0x123).

Listing 2

```
prompt>test
Running 3 test cases...
Test-fail.cpp(52): error in "testThatRegisterAssignmentWritesProperValueToRegister": check {
spy.getAllExpectations().begin(), spy.getAllExpectations().end() } == {
spy.getAllOccurrences().begin(), spy.getAllOccurrences().end() } failed.
Mismatch in a position 0: [write,0x17,0x123] != [write,0x17,0x246]

*** 1 failure detected in test suite "Master Test Suite"
```

Register class declaration

Ah, finally we get to see some bits of code! Listing 3 shows the Register class declaration.

Listing 3

```
/**
 * register type.
 */
template
< typename D                         // data type
, typename A = D*                    // address type
, typename CP = MemoryChannel<D,A>   // channel policy
>
class Register : public boost::noncopyable;
```

A Register object is defined by its data type (D), address type (A) and the concept (CP) that defines how the register is accessed, e.g. as memory or in another way.

To test registers access, the actual reading from and writing to the hardware registers must be intercepted. The user-provided access policy class make this possible [Henney06, Henney08]. Another approach would be to make the register access an abstract interface. However, the policy approach potentially provides better performance as the compiler can optimise away function calls, whereas it may not do that with the virtual function calls of an abstract interface.

As a register object represents a single hardware register, we prevent copying by inheriting the register class from `boost::noncopyable`.

Channel concept

How exactly registers are accessed is governed by the channel concept. It prescribes that the policy class provides a read() method and a write() method with proper data and address types (Listing 4). That's practically all there's to it, the type of the policy class itself is not relevant. This kind of polymorphism --compile-time polymorphism in case of C++-- is called duck typing [DT]: *if it walks, quacks and swims like a duck, it’s probably a duck.*

Listing 4

```
/**
 * the duck in the channel.
 */
struct ChannelConcept
{
   typedef sometype DataType;
   typedef sometype AddressType;

   DataType read ( AddressType );
   void     write( AddressType, DataType );
};
```

Memory-mapped register access

Some hardware registers are accessed in the same way as memory. These memory-mapped registers need only a simple access policy, such as the one shown in Listing 5.

Listing 5

```
/**
 * transport for memory mapped registers.
 */
template
< typename D    // data type
, typename      // A  address type unused
>
class MemoryChannel
{
public:
   typedef D DataType;
   typedef volatile DataType RegisterType;
   typedef RegisterType* AddressType;

   static DataType read( AddressType address )
   {
      return *address;
   }

   static void write( AddressType address, DataType data )
   {
      *address = data;
   }
};
```

Here, the address type is derived from the data type. With `volatile` in the intermediate RegisterType declaration we inform the compiler that the value of the memory location may change without the compiler being aware of it. The effect is that the compiler will not optimise away any reads from the location that it may consider redundant.

As we'll see shortly, the Register class can use channel policy class objects that are created either internally or externally. Some policy classes have no need for object member data and can use static member functions. The MemoryChannel class is an example of this: there's no need for an externally initialized object of it.

Intermezzo: Template type parameter or template template parameter

Early in development, I made the channel policy a template template parameter, so that the register class governs the channel's data and address types. For memory-mapped registers this seems a natural choice. However is it also a good choice for a channel that has its own fixed data and address types? Then when I wanted to use a spied-upon fiber interface card class as the channel policy of a card class, I was in trouble, because we are no longer feeding the card's class a template class but a type instead.

It was time to consult the ACCU-general mailing list and ask for reasons and consequences of choosing between a template type parameter and template template parameter. James Dennett's answer exactly mentioned what I was experiencing: a template type parameter gives extensibility/flexibility, or put otherwise the choice for a template template parameter results in non-extensibility and inflexibility [Dennett09]. Thus the channel policy template parameter became a type.

Now that data and address types for the register and the channel can be specified separately, this could introduce a conversion. However, for simplicity it is just checked at compile-time if the types are equivalent using for example `BOOST_STATIC_ASSERT( boost::is_same<D, typename CP::D>::value );`

Register class implementation

Listing 6 presents the implementation of the Register class. There are a couple of things to note.

- there are two constructors: one without channel object, one with channel object parameter
- the destructor contains a call to method checkTypes that statically asserts that data and address type of register and channel policy match.
- the class normally caches the value written to the hardware register to compensate for the fact that the registers are write-only; if a registers can also be read it usually has a different meaning, e.g. it is a status value instead of the written control value.

And of course, there are the methods to read and write the register as a whole, or test, set and clear bits, or groups of bits.

Listing 6

```
template
< typename D
, typename A = volatile D*
, typename CP = MemoryChannel<D,A>
>
class Register : public boost::noncopyable
{
public:
   typedef D DataType;
   typedef A AddressType;
   typedef CP ChannelType;

   Register( AddressType address, int offset, DataType data = 0 )
```

```
: m_channel_smartptr()   // Note: must be declared before m_channel
, m_channel( createChannel() )
, m_address( computeAddress( address, offset ) )
, m_cache( data )
{
   ; // do nothing
}

Register( ChannelType& channel, AddressType address, int offset, DataType data = 0 )
: m_channel_smartptr()
, m_channel( channel )
, m_address( computeAddress( address, offset ) )
, m_cache  ( data    )
{
   ; // do nothing
}

~Register() {
   checkTypes();
}

static void checkTypes() {
   BOOST_STATIC_ASSERT( ( boost::is_same<D, typename CP::DataType>::value ) );
   BOOST_STATIC_ASSERT( ( boost::is_same<A, typename CP::AddressType>::value ) );
}

operator DataType() {
   return read();
}

Register& operator= ( DataType data ) {
   write( data );
   return *this;
}

AddressType address() const {
   return m_address;
}

DataType cache() const {
   return m_cache;
}

DataType read() {
   return m_channel.read( m_address );
}

void write() {
   m_channel.write( m_address, m_cache );
}

void write( DataType value ) {
   m_channel.write( m_address, m_cache = value );
}

void write_nc( DataType value ) {
   m_channel.write( m_address, value );
}

bool bittest( int bit ) {
   return 0 != ( read() & bitmask( bit ) );
}

void bitclear( int bit ) {
   write( m_cache & ~bitmask( bit ) );
}
```

```
      void bitset( int bit ) {
         write( m_cache | bitmask( bit ) );
      }

      bool masktest( DataType mask ) {
         return mask == ( read() & mask );
      }

      void maskclear( DataType mask ) {
         write( m_cache & ~mask );
      }

      void maskset( DataType mask ) {
         write( m_cache | mask );
      }

      void maskset( DataType clearmask, DataType setmask ) {
         m_cache &= ~clearmask;
         maskset( setmask );
      }

      static DataType bitmask( int bit ) {
         return 1 << bit;
      }

      static AddressType computeAddress( AddressType base, int offset ) {
         return base + offset;
      }

   private:
      ChannelType& createChannel() {
         m_channel_smartptr.reset( new ChannelType() );
         return *m_channel_smartptr;
      }

   private:
      // to be replaced by std::unique_ptr:
      std::auto_ptr<ChannelType> m_channel_smartptr;
      ChannelType& m_channel;
      AddressType m_address;
      DataType m_cache;
   };
```

Although a register just has a single address, the constructor takes both an address and an offset. The rationale behind it is to concentrate address computations at a single point in the register class and not spread it over the classes that use the register class.

All in all, not too exciting a class. It's the combination of register operations, registers access and its testing that makes it interesting.

Using class Register

Now where do all these preparations bring us to? Listing 7 presents a small part of class DualFiberLinkImpl for the computer interface PCI card that connects the computer to the video-rate SPM controller.

Listing 7

```
template < typename CP >
class DualFiberLinkImpl
{
   typedef CP ChannelType;
   typedef Register< pci_data_t, pci_address_t, CP > RegisterType;

   class BusStatusRegister : private RegisterType
   {
   private:
      enum EStatusRegister
      {
         eBit_DataIsAvailable       = 0,
         eBit_WriteFifoIsFull       = 1,
         eBit_ErrorHasOccurred      = 2,

         eBit_ClearDataIsAvailable  = 0,
         eBit_ClearErrorHasOccurred = 2,
         ...
      };

   public:
      BusStatusRegister( ChannelType& channel, pci_address_t address, int offset )
      : RegisterType( channel, address, offset ) {;}

      bool fiberDataIsAvailable() const {
         return bittest( eBit_DataIsAvailable );
      }

      ...
      void clearFiberDataIsAvailable() {
         bitset  ( eBit_ClearDataIsAvailable );
         bitclear( eBit_ClearDataIsAvailable );
      }
   };
   ...
   enum ERegisterOffset
   {
      eRegOff_BusStatus = 1,
   };

public:
   DualFiberLinkImpl( ChannelType& channel, const pci_address_t address )
   : m_regBusStatus ( channel, address, eRegOff_BusStatus  )
   ...
   {
   }

   bool fiberDataIsAvailable() const
   {
      return m_regBusStatus.fiberDataIsAvailable();
   }

   void clearFiberDataIsAvailable()
   {
      m_regBusStatus.clearFiberDataIsAvailable();
   }

private:
   BusStatusRegister m_regBusStatus;
   ...
};
```

In normal use, the class will be instantiated with the `MemoryChannel` policy class to provide access to the memory-mapped PCI registers.

Configurable register spy

The channel concept not only allows for different ways to access real registers, it also provides the means to bring the register access into our test framework. A register spy class is a channel policy and besides that it can contain and provide the expected register access operations as well as the actually occurred register access operations [Meszaros07]. The macro call `SPM_CHECK_REGISTER_SPY_EXPECTATIONS(spy)` checks if the programmer-defined expectations are met (Listing 8).

Listing 8

```
/**
 * the spy we love.
 */
template < typename D, typename A >
class RegisterSpy;

// main spy operations:
void relax( bool relax = true );
void addReadExpectation ( AddressType address, DataType data, std::string remark = "" );
void addWriteExpectation( AddressType address, DataType data, std::string remark = "" );

// macros:

// the name of the current test case.
#define SPM_TEST_CASE_NAME ...

// issue test message; streams its argument.
#define SPM_TEST_MESSAGE( arg ) \
   BOOST_TEST_MESSAGE( "" << arg )

// issue test message: "'prefix' testcase_name" << 'postfix'.
#define SPM_TESTCASE_MESSAGE( prefix, postfix ) \
   SPM_TEST_MESSAGE( prefix << SPM_TEST_CASE_NAME << postfix )

// match expectations and occurrences.
#define SPM_CHECK_REGISTER_SPY_EXPECTATIONS( spy ) ...
```

Boost.Test

As already mentioned, we're using Boost.Test as the test framework. It nicely supports the described way of working through its message and test macros and its command line options. While developing, we use option --log_level=message, later on when used as regression test we use --log_level=error, which is the default. With option --run_test=*spec* we can select one or more tests to run instead of running all tests. For example, option --run_test=*/*Stm*/*Read* selects the tests with Read in their name from the (sub) test suites that have Stm in their name.

Sometimes the tests specified initially were wrong and failed, whereas the code to test was correct. I don't think this is a bad thing, it just makes you all the more conscious of what the code does. One thing I was able to spot immediately occurred when we moved from a Windows-API-based lock to the Boost-based lock. The read operation for the fiber interface PCI card halted where it previously had no problem. It appeared that I had inadvertently chosen a non-re-entrant mutex from Boost.Thread, whereas the CRITICAL_SECTION previously used for the mutex *is* re-entrant.

A test example

The Boost.Test main program (Test-main.cpp) is joyfully simple. The actual groups of related tests (test suites) are located in separate source files, such as Test-simple.cpp in Listing 9. Compile and link the source files with Test-main.cpp as the first to obtain the test program with all test suites.

Listing 9

```
// File: Test-main.cpp

#define BOOST_TEST_MAIN Master Test Suite
#include <boost/test/unit_test.hpp>


// File: Test-minimal.cpp

#include <iostream>        // std::cout

#include "Register.h"      // class Register
#include "RegisterSpy.h"   // class RegisterSpy
```

```
#include "Test-common.h"   // SPM_TEST_INSPECT_MESSAGE

#include <boost/test/unit_test.hpp> // Boost.Test

typedef int DataType;
typedef int AddressType;

typedef spm::tdd::RegisterSpy< DataType, AddressType > RegisterSpyType;
typedef      spm::Register   < DataType, AddressType , RegisterSpyType > RegisterType;

const AddressType base   ( 0x10 );
const int         offset ( 0x07 );
const DataType    initial( 0xe5 );
const AddressType address( RegisterType::computeAddress( base, offset ) );

BOOST_AUTO_TEST_SUITE( Register )
BOOST_AUTO_TEST_SUITE( Minimal )

struct Fixture
{
   Fixture() : spy( "Inspect>" ) , reg( spy, base, offset, initial ){;}
   ~Fixture() { SPM_TEST_MESSAGE( spy ); }

   RegisterSpyType spy;
   RegisterType    reg;
};

BOOST_FIXTURE_TEST_CASE( testThatRegisterAssignmentWritesCorrectValueToRegister, Fixture )
{
   SPM_TESTCASE_MESSAGE( "Inspect> ", " (Pass):" );
   const DataType value( 0x123 );
   spy.addWriteExpectation( address, value, "assign value to register" );
   reg = value;
   SPM_CHECK_REGISTER_SPY_EXPECTATIONS( spy );
}

BOOST_AUTO_TEST_SUITE_END() // Minimal
BOOST_AUTO_TEST_SUITE_END() // Register
```

In its most basic usage `spy.relax()` is called before the register is used and the register spy just records the access to one or more registers. However, here the spy is provided with a sequence of expectations of addresses and values that should be written and read and at the end of the test these expectations are matched with the actual register accesses to report any discrepancy (Listing 9).

Space and time efficiency

The design of the register class assumes that calls to the channel policy class are optimised away by the compiler. This results in code that's both smaller and faster because of the absence of a function call. Processor cache size limits also reward smaller code with faster execution.

Space efficiency

Each object of the Register class contains a smart pointer used for internally created channel policy objects, a reference to the channel, the register's address and the cache for the value written to the register. Would another approach, for example one where less information is stored in the register objects, lead to overall smaller code? I don't know. I didn't look into it because size per se is not a big concern to me. The chosen approach leads to quite simple code for register manipulation, as much is abstracted into functions to build on, so my impression is that it is size-efficient.

Register performance

Register access is at the heart of many operations, so performance may be an issue. Moving a slider that controls a voltage in the setup to pan (or zoom, rotate etc) the scanned area should be a smooth operation. Say run-time performance and the very next word is: measure. I timed the various operations the register class provides as well as comparable pointer-based memory access statements. Table 1 lists these measurements. Note that the tests access conventional memory as opposed to registers on a 66 MHz PCI bus (15 ns period). However our prime interest is to compare the performance of the Register approach with the presumed efficient way it is done in the legacy code.

Table 1: Operation memory access times on computer running Windows-XP SP3 with a 2.3 GHz AMD Athlon(tm) 64 X2 Dual Core Processor 4400+ and 2 GByte RAM. The program was compiled with MS Visual C++ 8, with options -O2 -EHs. Times are in [ns].

#	Median[ns]	Operation
1	1.31	`*p = x`
2	1.31	`reg.write_nc( x )`
3	1.31	`*p = cache = x`
4	1.595	`reg.write( x )`
5	1.6	`reg = x`
6	1.36	`cache = *p`
7	1.75	`cache = reg.read()`
8	1.75	`cache = reg`
9	1.36	`b = 0 != (*p & 0x01)`
10	1.57	`b = reg.bittest( 0 )`
11	1.75	`*p = (cache | 0x02)`
12	1.75	`reg.write_nc( reg.cache() | 0x02 )`
13	2.29	`*p = cache |= 0x04`
14	2.41	`reg.bitset( 3 )`
15	2.31	`*p = cache |= 0x0f`
16	2.41	`reg.maskset( 0x0f )`
17	2.9	`*p = cache = ((cache & ~0xf0) | 0x30`
18	2.98	`reg.maskset( 0xf0, 0x30 )`

Note that the timing also depends on other tasks running on the computer and therefore the test program was run 100 times to spread out the measurements in time. The register class performs quite well compared to the pointer-based access.

Table 2 presents the resulting assembly code. It appears that often the code generated for the Register class is the same or almost the same as for the equivalent pointer based statements.

Table 2: Assembly code when compiled with VC8, options -O2 -EHa.

```
 1  *p = x
    mov    ecx, DWORD PTR [esi+12]
    mov    DWORD PTR [ecx], eax

 2  reg.write_nc( x )
    mov    ecx, DWORD PTR [esi+28]
    mov    DWORD PTR [ecx], eax


 3  *p = cache = x
    mov    ecx, DWORD PTR [esi+12]
    mov    DWORD PTR [esi+4], eax
    mov    DWORD PTR [ecx], eax

 4  reg.write( x )
    mov    ecx, DWORD PTR [esi+28]
    mov    DWORD PTR [esi+32], eax
    mov    DWORD PTR [ecx], eax

 5  reg = x
    mov    ecx, DWORD PTR [esi+28]
    mov    DWORD PTR [esi+32], eax
    mov    DWORD PTR [ecx], eax

 6  cache = *p
    mov    ecx, DWORD PTR [esi+12]
    mov    edx, DWORD PTR [ecx]
    mov    ecx, DWORD PTR [esi+12]
    mov    DWORD PTR [esi+4], edx
 7  cache = reg.read()
    mov    eax, DWORD PTR [esi+28]
    mov    eax, DWORD PTR [eax]
    mov    DWORD PTR [esi+4], eax

 8  cache = reg
    mov    eax, DWORD PTR [esi+28]
    mov    eax, DWORD PTR [eax]
    mov    DWORD PTR [esi+4], eax

 9  b = 0 != (*p & 0x01)
    mov    ecx, DWORD PTR [esi+12]
    mov    edx, DWORD PTR [ecx]
    mov    ecx, DWORD PTR [esi+12]
    and    dl, 1
    mov    BYTE PTR [esi], dl

10  b = reg.bittest( 0 )
    mov    eax, DWORD PTR [esi+28]
    mov    eax, DWORD PTR [eax]
    and    eax, 1
    mov    BYTE PTR [esi], al

11  *p = (cache | 0x02)
    mov    ecx, DWORD PTR [esi+4]
    mov    edx, DWORD PTR [esi+12]
    or    ecx, 2
    mov    DWORD PTR [edx], ecx

12  reg.write_nc( reg.cache() | 0x02 )
    mov    eax, DWORD PTR [esi+32]
    mov    ecx, DWORD PTR [esi+28]
    or    eax, 2
    mov    DWORD PTR [ecx], eax

13  *p = cache |= 0x04
    mov    ebx, 4
    or     DWORD PTR [esi+4], ebx
    mov    eax, DWORD PTR [esi+4]
    mov    edx, DWORD PTR [esi+12]
    mov    DWORD PTR [edx], eax

14  reg.bitset( 3 )
    mov    eax, DWORD PTR [esi+32]
    mov    ecx, DWORD PTR [esi+28]
    or    eax, 8
    mov    DWORD PTR [esi+32], eax
    mov    DWORD PTR [ecx], eax

15  *p = cache |= 0x0f
    mov    edi, 15 ; 0000000fH
    or    DWORD PTR [esi+4], edi
    mov    eax, DWORD PTR [esi+4]
    mov    edx, DWORD PTR [esi+12]
    mov    DWORD PTR [edx], eax

16  reg.maskset( 0x0f )
    mov    eax, DWORD PTR [esi+32]
    mov    ecx, DWORD PTR [esi+28]
    or    eax, 15 ; 0000000fH
    mov    DWORD PTR [esi+32], eax
    mov    DWORD PTR [ecx], eax

17  *p = cache = ((cache & ~0xf0) | 0x30 )
    mov    eax, DWORD PTR [esi+4]
    and    eax, -193  ; ffffff3fH
    or     eax, 48    ; 00000030H
    mov    DWORD PTR [esi+4], eax
    mov    edx, DWORD PTR [esi+12]
    mov    DWORD PTR [edx], eax
```

```
18  reg.maskset( 0xf0, 0x30 )
    mov     eax, DWORD PTR [esi+32]
    mov     ecx, DWORD PTR [esi+28]
    and     eax, -193  ; ffffff3fH
    or      eax, 48    ; 00000030H
    mov     DWORD PTR [esi+32], eax
    mov     DWORD PTR [ecx], eax
```

Conclusion

Using fairly main-stream C++ constructs we provide a hardware register abstraction that enables us to write classes that represent hardware with a clear and regular design. The register abstraction allows for different methods to access the registers and with this it also provides the means to test register read and write access. And thanks to compiler optimisation, for memory-mapped registers access it has a performance akin to pointer-based access. So if we can speak of any degradation of performance, it is offset by enhanced testability.

Acknowledgements

Thanks to editor Ric Parkin for the gentle guidance of a first-time Overload author. Also thanks to Gert Jan van Baarle and Joost Frenken for their support in writing this article and their rapid review that made it possible to publish it one issue earlier than first envisioned.

Source code

The article's source code is available as a tar.gz file from the following web page:

http://www.eld.physics.LeidenUniv.nl/~moene/accu/overload/95/register/

Notes and References

[Boost] Boost free peer-reviewed portable C++ source libraries, http://www.boost.org/.
[Dennett09] James Dennet, accu-general mailing list, December 2009, http://lists.accu.org/mailman/private/accu-general/2009-December/018308.html.
[DMA] Direct Memory Access, access system memory independent of the CPU.
[DT] Duck Typing (Wikipedia), http://en.wikipedia.org/wiki/Duck_typing.
[Feathers04] Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004.
[Flexible] I almost wrote the words *flexible* and *reuse* here, however: "The word *flexible* is like *reuse*: it should alert you that something nebulous is probably up. Classes and functions are not designed to be flexible, they are designed for a purpose: flexibility is not a purpose, nor is it either a quality or a quantity; it is a bucket term, a catch all, snake oil." See [Henney02].
[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley Professional, 1st edition, 1999.
[Fowler09] Martin Fowler, TechnicalDebtQuadrant, October 2009, http://martinfowler.com/bliki/TechnicalDebtQuadrant.html.
[Frenken05] Joost Frenken et al., Pushing the limits of SPM, Materials Today, May 2005, http://www.physics.leidenuniv.nl/sections/cm/ip/group/PDF/Materials%20Today/%282005%2920.PDF; For a more in-depth article, see [Rost09].
[Goodliffe05] Pete Goodliffe. A Technique for Register Access in C++, ACCU Overload 68, August 2005, http://accu.org/index.php/journals/281.
[GTest] Google C++ Testing Framework, http://code.google.com/p/googletest/.
[Henney02] Kevlin Henney, minimalism, the imperial clothing crisis, http://www.two-sdg.demon.co.uk/curbralan/papers/minimalism/TheImperialClothingCrisis.html.
[Henney06] Kevlin Henney, Context Encapsulation, Three Stories, a Language, and Some Sequences, January 2006, http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pdf.

[Henney08] Kevlin Henney, The PfA papers: Deglobalisation, Overload, February 2008, http://accu.org/index.php/journals/1470.
[Henney09] Kevlin Henney, GUT Instinct, Sticky Minds, May 2009, http://www.stickyminds.com/pop_print.asp?ObjectId=14973&ObjectType=ART.
[HR] Hardware register (Wikipedia), http://en.wikipedia.org/wiki/Hardware_register.
[IP] Interface Physics, Universitet Leiden, Netherlands, http://www.physics.LeidenUniv.nl/sections/cm/ip/.
[LPM] Leiden Probe Microscopy, http://www.leidenprobemicroscopy.com/.
[Meszaros07] Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison–Wesley Professional, 2007. See Test Spy, http://xunitpatterns.com/Test%20Spy.html.
[Roberts] Tim Roberts, If every hardware engineer just understood that...write-only registers make debugging almost impossible, http://www.microsoft.com/whdc/resources/MVP/xtremeMVP_hw.mspx#ETB.
[Rost09] Marcel Rost et al., Video-rate Scanning Probe Control Challenges: Setting the Stage for a Microscopy Revolution, Asian Journal of Control, March 2009, http://www.physics.leidenuniv.nl/sections/cm/ip/group/PDF/Asian%20J.%20of%20Control/11%282009%29110.pdf.
[SPM] scanning probe microscopy (Wikipedia), http://en.wikipedia.org/wiki/Scanning_probe_microscopy; See also [SPMBBC].
[SPMBBC] scanning probe microscopy (BBC), http://www.bbc.co.uk/dna/h2g2/A717563.
[TC] Tunnel current is the quantum effect that a small current can flow between conductors that have no physical contact if they are a few nm apart, (Wikipedia) http://en.wikipedia.org/wiki/Scanning_tunneling_spectroscopy.
[TDD] Test Driven Development (Wikipedia), http://en.wikipedia.org/wiki/Test-driven_development.
[TeX] inspired on the \relax command of Donald Knuth's TeX typesetting system (Wikipedia), http://en.wikipedia.org/wiki/TeX.