A C++ Active Library Device Driver Development

Russell Yanofsky – rey4@columbia.edu
Department of Computer Science, Columbia University

***Abstract* – This report presents a C++ library that can assist in the development of device drivers. The library provides mechanisms to replace low-level bit manipulation code used to control a piece of hardware through its registers with calls to higher level routines that the library generates from a hardware description. The paper describes the use and implementation of the library and compares it to other tools which provide similar functionality.**

I. Introduction

Device drivers act essentially as glue code that pass instructions and data between a hardware and a software interface. Drivers tend to have a lot of code in common. On a given operating system, device drivers of the same type will be given similar structures in order to conform to their host's API. On a more fundamental level though, all hardware device drivers use similar bit-manipulation techniques, primarily for sending and receiving information from hardware registers. Unfortunately, this code that nearly all device drivers have in common, cannot be expressed in a generic way in C, the language in which most device drivers are written. A generic approach is desirable because it can make code more uniform. Many drivers define their own sets of C macros for accessing their registers in a high level way. This can make writing code easier because it eliminates redundancies, but it can make reading and extending code more difficult because there is an additional driver-specific layer of abstraction to understand. Another problem with writing driver code in C is it is harder to provide the compile time and run time checking that is possible when other languages when used.

These drawbacks have spurred the creation of tools that work outside of C to make driver development easier. For dealing with the problem of providing generic access to registers, various Domain Specific Lanugages (DSLs) have been proposed and implemented. DSLs allow driver writers to write high level code in a specialized language that can be used to automatically generate C code that users can invoke in their drivers to handle low level tasks. Currently though, none of the DSL created for this purpose are widely used. DSLs suffer from the inherent drawback that they require developers to learn a new syntax or GUI. They also normally do not offer general purpose imperative language features, so they can be difficult to extend. And because they must be executed, they introduce another dependency in the build process.

This paper will describe an alternate approach, in which domain-specific code generation is implemented in a C++ library. C++ is a general purpose programming language which has some meta-programming facilities which allow it to mimic features of many DSLs. The first part of this paper describes the features offered by existing DSLs, and the second describes the use and implementation of the library.

II. Related Work

Consel et al [1] designed an implemented a language called Devil (DEVice Interface Language) which generates low level driver code for use in Linux and SunOS. With the Devil language, a programmer specifies how to access hardware registers for a device, and identifies bit ranges within those registers which can be treated as distinct variables. The Devil compiler then generates a series of C macros which read and write to the variables, and the driver code can call these macros to receive status information and send commands to hardware.

Since reads and writes to hardware registers have side-effects, Devil allows the programmer to specify conditions under which variables should be cached. And, to allow for cases when many variables need to be read of written at the same time, Devil allows variables to be grouped into structures, and generates functions to read and write entire structures at once.

To deal with complicated hardware schemes for accessing registers, the driver writer may specify actions that must occur before and after register reads and writes. The more complicated access schemes can make use of private variables in actions, virtual registers, which map to other registers, and parameterized registers that take integral parameters for use in pre and post actions.

One of the major goals of Devil, aside from making driver code easier to read and write, is to provide safety. Each variable that the programmer declares can be constrained by a series of enumerated constants or by a bitmask. When a value is read or written which does not satisfy the constraints an error message can be printed at runtime. (Due to C's weak type system, Devil cannot always ensure that C code accessing variables is correct)

Devil is partially based on the Graphics Adapter Language (GAL) by Thibault et al [2] which was created a year earlier by the same research group. GAL provides similar means of accessing hardware registers as well as a number of other features specific to video drivers.

Manolitzas [3] describes a DSL for implementing Linux network drivers. His language is essentially to be a superset of Devil. But instead of being an interface language which generates C helper functions, it is an imperative language which can be used to generate an entire driver. The language requires the driver writer to specify at least 5 functions: init, open, transmit, receive, and stop as well as an interrupt handler. The functions and the handler are written in a Pascal-like imperative language that has special features for reading and writing to hardware variables, managing memory, and performing synchronizations.

DSLs need not be text based. Compuware Driverworks provides a set of GUIs that can generate driver stub code for Windows platforms. Unlike, the DSLs described above though, the purpose of this product is to simplify the portions of driver code that interact with the operating system, rather than the parts that interact with the hardware.

In some sense, solving a problem with C++ is the exact opposite of solving a problem with a DSL. DSLs are small, limited purpose languages while C++ is a huge general purpose one. But while these differences may mean a lot to the people implementing domain specific solutions (since developing a C++ library is a different process than writing a compiler), the end goal is nearly the same. The ultimate goal is to allow the end user to write a small amount of expressive code to replace large amounts of repetitive or complicated code.

```
// device
device BusMouse (base : bit[8]
  port @ {0..3})
{
  // index register
  register index_reg = write base @ 2,
    : bit[8];

  // index variable
  variable index = index_reg[6..5] : int(2);

  // registers for low and high bits
  register y_low  = read base @ 0, pre
    {index = 2} : bit[8];
  register y_high = read base @ 0, pre
    {index = 3} : bit[8];

  // dy variable
  variable dy = y_high[3..0] # y_low[3..0],
    volatile : signed int(8);
}
```

```
// index register
typedef Register<8, PortIO<2> > IndexReg;

// index variable
typedef Variable<AtRegister<IndexReg, 6, 5> >
  Index;

// registers for low and high bits
typedef Register<8, IndexIO<
  PortIO<0>, Index, 2> > Y_Low

// registers for low and high bits
typedef Register<8, IndexIO<
  PortIO<0>, Index, 3> > Y_High

typedef Variable<list<AtRegister<Y_Low, 3, 0>,
  AtRegister<Y_High, 3, 0> > > DY;

struct BusMouse: public Device<list<IndexReg,
Y_High, Y_Low>, BusMouse >
{ ... };
```

Figure 1. The left column shows the Devil hardware description for a Logitech mouse driver. The right column shows the same description as a series of C++ type definitions. Example based on [1]

The use of C++ Libraries to implement domain specific solutions is nothing new. Blitz++[*] and POOMA[†], two pioneering C++ numeric libraries that perform domain specific optimizations on high level numeric code are now more than 5 years old. Other code-generating C++ libraries like parser generators and finite state machine emulators are being actively developed. In comparison, this library, which generates instructions to access registers from variable accesses, is simple, and even straightforward. Past efforts at building DSL-like libraries in C++ have led to the discovery of generally useful techniques for code generation. Czarnecki and Eisenecker [4] describe many of these techniques in detail, and specifically how they can be used to emulate features in real DSLs. These will described later in this paper in the context of this library's implementation.

III. Library Design

The C++ library described in this paper attempts to provide developers with simpler ways of accessing the hardware in the same way that Devil does.

Registers and hardware variables which are described using language constructs in Devil are described with type declarations in C++. Figure 1 shows some of these the C++ type declarations and the equivalent Devil statements which partially describe the hardware interface to a Logitech mouse.

Each variable and register that is used to interact with the hardware corresponds to C++ type which is specified by the user by parameterizing generic `Variable` and `Register` template classes. Users never need to instantiate these types directly, they can just use them as parameters to the `Device` template class and the methods it provides. The `Device` template class is meant to be inherited from. It provides four template methods to the derived class: `get<X>()`, `set<X>(int)`, `read<X>()`, and `write<X>()`. The `get` and `set` methods are used to access cached values of registers. The `write` and `read` methods respectively copy the cached value into the hardware register, and copy the current value of the hardware register into the cache. All four methods take a template member X. The `get` and `set` methods require X to be a variable type. The `read` and `write` methods let X be a variable, a register, or a list of variables or registers.

A class that inherits from the `Device` template class provides its parent with a list of registers and its own type as parameters. `Device` takes the list of registers and

[*] http://oonumerics.org/blitz/
[†] http://www.acl.lanl.gov/pooma/

instantiates each one as a member (registers need to be instantiated because as objects they hold the cached register values). `Device` takes the type of its subclass so its read and write methods can access information stored in the subclass, for example a base address or an operating system handle. Passing the type of a subclass as a parameter to the parent, is part of a generally useful technique called the *curiously recursive template pattern*, which will be described in the next section.

This library is designed to be very extensible. The place where extensibility is most needed is in allowing the user to provide methods for reading and writing registers. The user can do this by writing a simple I/O class with two static methods called read and write. The methods take a reference to a Driver class and a reference to a Register class as parameters, and can use the interfaces provided by those objects and operating system APIs to perform whatever actions are needed. The library comes with a few of these classes built in. PortIO and IndexIO and are demonstrated in Figure 1. Also provided are MemoryIO, DebugIO, and NoIO. Some of these classes allow chaining. For example, IndexIO sets a device variable to a constant value before reading or writing a register through some another I/O type passed in as its first parameter. DebugIO is another chaining type that prints the values of registers as they are read and written. Each register takes an I/O class as a parameter.

Other types of extensibility can be achieved by writing drop in replacements for Variable and Register types. Types passed in to the device interface as variables and registers need not be instances of `Variable` or `Register` template classes. They only need to provide the same methods.

IV. C++ Techniques

This section describes three generally applicable C++ techniques used in the implementation of the library: Typelists and the curiously recursive template pattern, and static checking.

Many of the parameters for the Variable and Register Templates are lists of arbitrary length, instead of individual values. These lists are called as typelists. The most comprehensive and easy to understand description of typelists was written by Alexandrescu in [5], but typelists are also mentioned in [1]. Typelists are made up of a chain of `Node` types where each `Node` type contains an arbitrary type and the type of the a successor Node. The last node has a successor type of `Null`, where Null is just a special placeholder class. Figure 2 shows the specific definition for the `Null` type and the `Node` template class. Figure 3 shows how to declare a list of three types (int, signed int, unsigned int) using Node and Null classes. This notation is verbose and cumbersome, because it requires that template parameters be deeply nested. It is possible to provide a shorthand syntax using a template class that accepts a variable number of parameters. This shorthand is also demonstrated in Figure 3.

```
struct Null;

template<typename T, typename NEXT>
struct Node
{
  typedef T type;
  typedef NEXT next;
};
```

Figure 2. Node template class and Null types are used to make typelists.

Direct typelist declaration:

```
typedef Node<int, Node<signed int,
  Node<unsigned int, Null> > >
  MyList;
```

Shorthand typelist declaration

```
typedef List<int, signed int,
  unsigned int> MyList;
```

Figure 3. How to declare a list of three types, using the `Null` and `Node` classes directly, and by using `List` shorthand.

The real power of typelists comes from the fact that they can be manipulated and used to generate classes and values using compile-time algorithms. Figure 4 shows a simple algorithm class called `Length` that determines how many elements are in a typelist passed to it as a parameter. The length template class is specialized for the Null type to give a length of 0. It is specialized for any Node type to give a length of 1 plus the length of the Node's successor list. So when it is passed `MyList` it will give a length of (1 + (1 + (1+ 0))) = 3.

```
template<typename LIST>
struct Length;

template<>
struct Length<Null>
{
  enum { value = 0 };
};

template<class T, class U>
struct Length< TypeList<T, U> >
{
  enum { value = 1 + Length<U> };
};

cout << Length<MyList>::value;
```

Figure 4. Definition and use of the Length algorithm.

Length is one of the simplest typelist algorithms. Other commonly used algorithms return classes which inherit from every type on the list or returned sorted or filtered versions of lists. There are entire libraries filled with algorithms for manipulating typelists, including Boost::MPL[‡] (MetaProgramming Library) and Loki[§]. MPL was used heavily in the implementation of this library. Along with providing algorithms for typelist manipulation, it provides cross-platform versions of metaprogramming constructs like if statements and lambda expressions, that eliminate the need for rote template specialization. MPL was written by Aleksey Gurtovoyi and described in a long paper by Abrahams et al [6].

The `Device` template class which gets inherited by user's device classes can be seen as an abstract base class, because it can depend on information stored in descendants for performing register I/O.

Abstract base classes are normally able to call methods on the classes which inherit from them. By default this works in C++ and some other object oriented languages through virtual function calls. Virtual function calls are problematic in this case, however, because C++ does not support virtual calls on templated functions. Additionally, virtual calls only allow abstract base classes to access their descendants' functions, and not their internally defined constants and type definitions. A solution to both of these problems comes in the "curiously recursive template" pattern, also known as Barton and Nackman trick. This trick works by turning the abstract base class into a template class which takes the type of the descendant class as a single parameter. This way, when the base class needs to call a function on its descendant, it can cast its `this` pointer to the descendant type, and proceed to invoke the correct method. Figure 5 shows an example of this technique taken from Veldhuizen [7].

```
template<class T_leaftype>
class Matrix {
public:
  T_leaftype& asLeaf()
  { return
static_cast<T_leaftype&>(*this); }

  double operator()(int i, int j)
  { return asLeaf()(i,j); }
};

class SymmetricMatrix :
  public Matrix<SymmetricMatrix> {
  ...
};
```

Figure 5. The curiously recursive template technique.

Another generally useful technique is static checking. Static checks are performed in macros that look just like assertions, except that if the condition is not true there will be an error at

[‡] http://www.boost.org/libs/mpl/doc/
[§] http://www.moderncppdesign.com/

compile time instead of run time. Static checks don't require advanced multiprogramming functionality, and they can be implemented even in C, although they are less useful there. In C++ most static checks work by attempting to instantiate a template class that is specialized for <true> and undefined for <false>. Various approaches to static checking are described and compared by Alexandrescu [5].

V. Implementation

The metaprogamming features described above and many others are used to implement a number of features in this library. The ability to synthesize classes from typelists is used to use instantiate the registers passed to the `Device` template class as members of the resulting type.

Lists are also used to associate variables with bits from multiple registers or with a list of non-continuous bit segments of a single register. In these cases, the get and put methods traverse the list at runtime, getting and setting bits at each location in the list.

When a list of variables is passed to the read and write methods, typelist algorithms are used to map it into a list of registers, and then to eliminate duplicate registers from that list, before traversing it at runtime to call methods on each register. Static checking is used all over the library to ensure consistency in the device specification. For example, if an invalid bit range is specified in parameters to a variable, the compiler will show an error on the line which checks the bit range.

VI. Conclusions

So far, the library has not been used to implement a complete driver. However, the fact that the interface is so similar to Devil should suggest that the same readability and programmability improvements that were measured using that language would be likely to apply to this library as well. Performance is another area that needs to be investigated. In theory, assuming all of the library code is inlined, a driver produced with this library should be equivalent to a driver that has all the bitwise operations done in place. In practice, though it is not a good idea to take C++ compiler optimizations for granted.

The library's interface could also be extended to emulate some more of Devil's redundant language features such as "algebraic" enumerations, which are functionally equivalent to device variables, except that they allow the user to associate relevant register bit ranges with constants instead of device variables. A list of all language features in the Devil language specifications and their equivalents in this library is included in the source.

---

[1] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, Gilles Muller. Devil: An IDL for Hardware Programming. OSDI 2000, pages 17-30, San Diego, October 2000.

[2] Scott Thibault, Renaud Marlet, Charles Consel. A Specific Domain Language for Network Cards. 2001. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/reports/apostolos.pdf

[3] Apostolos Manzolitas. A Specific Domain Language for Network Cards. 2001. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/reports/apostolos.pdf

[4] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[5] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patters Applied*. Addison-Wesley, 2001.
Techniques for Scientific C++

[6] David Abrahams and Aleksey Gurtovoyi. The Boost C++ Metaprogramming Library. *Unpublished*. http://www.boost.org/libs/mpl/doc/paper/mpl_paper.html

[7] Todd Veldhuizen. Techniques for Scientific C++. Indiana University Computer Science Technical Report #542, August 2000.
http://osl.iu.edu/~tveldhui/papers/techniques