STR 96-027

A Hardware Control Class Hierarchy

T. J. Radcliffe
Department of Physics,
Queen's University at Kingston,
K7L 3N6, Canada

1 Introduction:

This document describes a C++ class hierarchy for doing realtime control. The classes described herein don't actually do any interaction with hardware, but provide some features that any hardware control system requires. Software objects are sometimes likened to integrated circuits, with the individual lines of code being similar to discrete components. The **Hardware** class hierarchy is more like a NIM-bin: a standardized crate that various modules can be slotted into with ease. The individual **Hardware**-derived classes are like NIM modules that can be slotted into the standard structure.

An overview of the design is given in Section 2, which includes an example of a main loop to implement the classes. How to write a new **Hardware**-derived class is shown in Section 3, including a discussion of multiple inheritance. Other functionality of the **Hardware** classes is discussed in Section 4; many of these helper functions are very useful when writing **Hardware**-derived classes, and so this section is recommended reading.

Note on capitalization etc.: there is a widespread practice in object-oriented programming circles for the first letter of class names to be upper case, and the first letter of object names to be lower case. This is a practice that is slowly being built into the existing **Hardware**-derived code, and I heartily encourage everyone to follow it. It may be mindless conformance to arbitrary, socially defined conventions, but then again, so is speaking English. In this document, class names will be mostly **bold face** and function names will be mostly *italic*. Code examples and variable names will be `typewriter`.

Note on standards: C++ is not very standardized just yet, and all of this code was meant to run on IBM PC compatibles when compiled with the Borland C++ compiler. Parts of it have also been complied with the GNU g++ complier version 2.6.8 under LINUX, and I found that a few minor changes had to be made to do this. When I refer to "ordinary C++" I mean whatever the people at Borland think of as ordinary. If you try to port this code to another platform or even another compiler on the same platform, you may find a number of errors due to small differences in standards.

2 Functionality:

The functionality these classes support is:

- polling of all derived classes
- command distribution to the right object
- basic command parsing.

These functions are supported by a hierarchy of three classes:

- **Hardware** class to provide a common interface for polling and commands, and to provide automated name handling
- **Command** class to provide simple parsing, to the extent of calling member functions based on command string input
- **Controller** class to allow different derived classes to have different **Command** lists but still be handled via the common **Hardware** interface.

The idea is to allow the user to send a command to any **Hardware**-derived object via a single call to the static member function *Hardware::doCommand()*, and to poll all **Hardware**-derived objects with a single call to the static member function *Hardware::doPoll()*. For example, a simple main loop is shown in Table 1.

With the main loop shown in Table 1 every object that is a **Hardware**-derived class has a chance to claim the command in inString as its own. The syntax for all **Hardware** commands is:

```
objectName commandName arg1 arg2 arg3 ...
```

when *Hardware::doCommand(inString)* is called all the **Hardware**-derived objects are looped over and their names are compared to the first token in inString. If none match then an error message is returned (error messages are discussed in detail below in Section 5). If an object is found that matches

```
/* Hardware class initialization stuff... */
Motor m1(``motor1''); // some Hardware-derived classes
Motor m2(``motor2'');
Penguin p1(``Pingu'');

char inString[512]; // input string from keyboard

do
{
  if (getStringFromKeyboard(inString)) // TRUE if input is ready
  {
    Hardware::doCommand(inString); // parse inString and run command
  }

  Hardware::doPoll(); // poll all Hardware objects

} while (0 == 0);
```

Table 1: Simple main loop to implement **Hardware** functionality

the object named by `inString` then the rest of `inString` is passed to it via the *command()* member function of the **Controller** class. **Controller** is a template class that is derived from **Hardware**. There is a different instance of the **Controller** template for each **Hardware**-derived class, and each **Controller** instance maintains a table of **Command** class objects appropriate to that class. When inString is passed to *command()* this table is searched for the appropriate command name. If it is found, then the associated member function is run. An error string is returned if the command name is not found. Beyond this point the member function has to provide its own parsing, which is a sensible division of labour as only it can know what it wants to do with the rest of the command arguments.

At the moment there are also some helper messages that are displayed when error conditions occur. If a command string contains an object that can't be found a list of known objects is given. If a command can't be found a list of known commands is given. **NOTE that name matching is case-insensitive.**

This scheme allows each object to do its own parsing, rather than having a separate parser that deals with commands for all objects. The advantage of the former method is that every object supplies its own functionality, and the loftier bits of the hierarchy (i.e. the **Hardware**, **Controller** and **Command** classes) can be utterly ignorant of the nature of derived classes and their member functions. To pull this off it is necessary that each derived CLASS have its own table of **Command**-derived objects. The natural way to do this in C++ is to derive **Hardware** classes from a base class that has a static array of **Command** objects. The difficulty with this approach is that inheritance from a non-template class would result in members of ALL derived classes sharing the same static array of **Commands**, which is clearly not appropriate. The solution is to insert a template base class between the **Hardware** class and its derived classes. All **Hardware**-derived objects inherit **Hardware** solely through an instance of this template base class, which is called **Controller**.

The **Controller** template class takes two formal parameters: one is a class type specifier, and the other is an integer expression that gives the number of commands (i.e. the size of the **Command** array) for that instance. The class type specifier is a dummy that is used only to ensure that each instance is unique: it is not actually needed by any of the **Controller** member functions. Each **Hardware**-derived class inherits a static array of **Command** class objects via the **Controller** template class, and because each instance of a template class contains unique static data, each **Hardware**-derived class has its own set of **Command** objects. These objects must be initialized by the subclass code in a manner described below.

The **Controller** template class contains a function that overloads the pure virtual function *command()* in the **Hardware** class. The purpose of this function is to search the command array for a **Command**-derived object that has a name that matches the command name passed in via the *Hardware::doCommand()* input string. If such a **Command**-derived object is found, its member function *subCommand()* is run, with the `this` pointer and the command argument string parsed from *Hardware::doCommand()* passed as arguments.

The *subCommand()* member function of **Command** is generally a one-liner that just uses the `this` pointer argument to call a member function of the class that contains the **Command**-derived class. This member function generally takes the argument string directly as an argument, and does its own conversions for numerical data contained in the string.

3 Creating a New Hardware-derived Class

If you're read the previous section carefully, you are probably pretty confused by this point. Fortunately, one of the great things about object-oriented programming is that you don't have to understand the details of the base classes to use them, any more than you have to understand thermodynamics to use a car or principles of just government to be Prime Minister.

From a programmer's point of view, use of the **Hardware** hierarchy follows a very simple algorithm. First create a header file for the new class with a declaration of the form shown in Table 2.

The derived class inherits all of its **Hardware** functionality via an instance of the **Controller** class. The **Controller** template takes two parameters (the things in <> after the word "Controller" at the top of the declaration.) The first of these is the type of the derived class (in this case **Derived**)and the second is the number of commands the class takes. The command number is used internally by the base classes to search the derived class's command array for the name of a command.

By convention, the derived class has three sections to its declaration. The first section is a private part that contains all the ordinary internal data that describes class objects. The second section is a public part that contains the ordinary member function prototypes, as well as the prototype for the *poll()* member function that will be overloaded and a constructor that takes a character string containing the object name as an argument. The third section is a private part that contains the declarations for the Command-derived classes that bind a command name to an ordinary member function.

```
#define DERIVED_COMMAND_NUMBER 3 // number of Commands in list

class Derived: public Controller<Derived, DERIVED_COMMAND_NUMBER>
{
 private:
   int i;  // ordinary internal class data

 public:   // public class member functions
   Derived(char* objectName); // constructor (this is REQUIRED)
   void poll(void); // overloaded polling function

   char* getI(void); // ordinary public class member functions
   void  setI(int);
   void  die(void);

 private: // Command-derived private members
   class GetI: public Command // get internal value
   {public:
       GetI(void):Command(''getI''){;} // pass name to Command
       char* subCommand(Hardware *hw, char* inString)
         {return ((Derived*) hw)->getI();}  // call member function
   };

   class SetI: public Command // set internal value
   {public:
       SetI(void):Command(''setI''){;}  // pass name to Command
       char* subCommand(Hardware *hw, char* inString) // call function
         {((Derived*) hw)->setI(atoi(inString)); return ''HW_ERR_OK''}
   };

   class Die: public Command // kill off process
   {public:
       Die(void):Command(''die''){;} // pass name Command
       char* subCommand(Hardware *hw, char* inString) // call function
         {((Derived*) hw)->die(); return ''HW_ERR_OK'';}
   };
 }; // end of Hardware-derived class declaration
```

Table 2: Header file for Hardware-derived class

Not much needs to be said about the two initial sections, as they are nothing more than ordinary C++. The third section contains a list of class declarations for **Command**-derived classes. These declarations have two important parts: the constructor and the *subCommand()* member function. The sole purpose of the constructor is to pass the NAME of the command to the **Command** base class. When this name appears in a string passed to the command parser for the derived class, the *subCommand()* function will be run. The *subCommand()* function takes a pointer to a **Hardware** object and a character string as arguments. Passing the first argument as a pointer to **Hardware** avoids problems with C++ type checking. Inside the function this pointer is then cast to the derived type, and one of the derived type's member functions is run. For member functions that do not return a value, the string ``HW_ERR_OK'' is returned to indicate that the command was run.

The basic code file for a **Hardware**-derived class is equally simple, as shown in Table 3. The derived class code consists of four sections:

- initialization of static data in **Controller**
- naming constructor that handles `commandArray` element initialization
- *poll()* member function that handles hardware access
- ordinary member functions that give access to internal data

Each of these sections warrants a few comments. Static data of a C++ class must be initialized. The following explanation has been lifted verbatim from the G++ info pages:

```
Declare *and* Define Static Members
-----------------------------------

   When a class has static data members, it is not enough to
   *declare* the static member; you must also *define* it.
   For example:

     class Foo
     {
       ...
       void method();
       static int bar;
```

```
#include ``derived.h'' // Derived class declaration
#include <stdlib.h> // atoi() prototype
#include <dos.h> // inport() prototype

/* initialize static data in Controller class: */

int Controller<Derived,DERIVED_COMMAND_NUMBER>::objectNumber = 0;
Command* Controller<Derived,DERIVED_COMMAND_NUMBER>::commandArray[DERIVED_COMMAND_NUMBER];

/* Naming constructor passes object name to base classes */
Derived::Derived(char* objectName):
  Controller<Derived,DERIVED_COMMAND_NUMBER>(``Derived'',objectName)
{
  i = 10; // initialize ordinary data

  if (objectNumber == 1)  // first object of this type
  {
    int index = 0; // commandArray index

    commandArray[index++] = new GetI;
    commandArray[index++] = new SetI;
    commandArray[index++] = new Die;

    if (index != DERIVED_COMMAND_NUMBER)
    {
      fprintf(stderr,``Bad command number in Derived constructor\n'');
      exit(-1);
    }

    for(index = 0; index < DERIVED_COMMAND_NUMBER; index++)
    {
      if (!commandArray[index])
      {
        fprintf(stderr,``Allocation failure in Derived constructor\n'');
exit(-1);
      }
    }
}

/* The polling function does the actual hardware access */
void poll(void)
{
   i = inport(0x300); // 0x300 is a typical board base address
}

/* The following are just the ordinary member functions of the class. */
char* getI(void) {static string[15]; sprintf(string,``%d'',i); return string;}

void setI(char* string) {i = atoi(string);}

void die(void) {fprintf(stderr,``Shutting down\n'');  exit(-1);}
```

Table 3: Code for a simple Hardware-derived class

```
  };

This declaration only establishes that the class 'Foo' has
an 'int' named 'Foo::bar', and a member function named
'Foo::method'.  But you still need to define *both*
'method' and 'bar' elsewhere.  According to the draft ANSI
standard, you must supply an initializer in one (and
only one) source file, such as:

  int Foo::bar = 0;
```

So the first two code lines in any **Hardware**-derived source file should contain the definitions of the static members of the corresponding **Controller** instance, as shown.

The naming constructor is is used to pass the name of the object being created up the hierarchy to the base classes. This is necessary so the name of this object will be known when *Hardware::doCommand()* tries to parse input strings and send commands to the appropriate object. This parsing is done by simply matching the first token in the string with the name of the object. The constructor also has the task of initializing the elements of `commandArray` for the class the first time it is called. `objectNumber` is a static member of the **Controller** class that counts how many objects of that type have been created. The base class constructors are called prior to the derived class constructor, so `objectNumber`, which is initialized to zero, will have a value of one when the derived class constructor gets to it. The constructor then assigns each element of `commandArray` to a different **Command**-derived member function of the derived class. `commandArray` is a static member of **Controller**, and once *Hardware::doCommand()* finds the right object for a command it passes the rest of the command string (stripped of the object's name) to the *command()* function of that object. The *command()* function loops over all of the commands in `commandArray` and calls the function whose name matches the first token in the string. The matching is case-insensitive.

One of the uglier aspects of this implementation is that the deallocation of the `commandArray` elements is handled by the **Controller** destructor, even though they are allocated at a lower level. I haven't thought of a nice way of allocating them higher up, and so long as care is taken to do the allocation properly and to CAREFULLY CHECK FOR ERRORS after allocation is nominally complete this should not be a problem.

The sole purpose of the *poll()* member function is to access hardware and update any variables that need to reflect the current state of the hardware. Note that *poll()* does not take any arguments or return any values. All it does is deal with hardware and handle any other updating of the internal state of the class.

The ordinary member functions are the complement of *poll()*. They should not access hardware at all, but only return values that have been grabbed from hardware by *poll()*. This ensures that if another object uses an ordinary member function to access the state of an object that the same values will be returned on sequential calls that are not interrupted by another cycle through the polling loop. If another object requires continually updated values it must call *poll()* for the object it is interrogating by hand.

A note on inheritance: the **Hardware** hierarchy has not been designed to support multiple inheritance of **Hardware**-derived classes. If you need to create **Hardware**-derived classes that contain other **Hardware**-derived classes this must be done by inclusion, not inheritance.

You now know pretty much everything you need to create your own **Hardware**-derived class. The examples above contain all of the required functionality, and simply copying them out and compiling them should produce a working (although not very interesting) class.

4 Other Hardware Class Functionality

This section describes various other member functions of the **Hardware** class. A complete list of the **Hardware** class member functions is given at the end of the section.

Any **Hardware**-derived object has a state described by the private flags `commandable` and `pollable`. There are a few cases, particularly for debugging purposes, when you may want to turn off polling for an object. The polling state is changed by the *pollOn()* and *pollOff()* member functions. Likewise, it is very often the case that when an object is part of a higher-level object you may want to prevent it from accepting commands while the higher-level object is doing something. For instance, it would be very bad to reset a counter while it is being used by a higher-level object to determine the position of something that is being moved by a motor: the motor-control algorithm of the higher-level object would have a hard time coping with this. So the command-acceptance state can be changed by using the *commandOn()* and *commandOff()* **Hardware** member functions. All objects are default commandable and pollable.

For example, suppose you have a **Valve** class that includes a **Motor** and an **PositionEncoder** object. The member function *shutValve()* might look like:

```
void Valve::shutValve(void)
{
  valveMotor->stop(); // stops any motion currently happening

  valveMotor->commandOff(); // makes sub-objects un-commandable
  valveEncoder->commandOff();

  moveValue(-amountOpen);  // closes valve by the amount it is open
}
```

The example assumes that amountOpen is a private member of the **Valve** class that tracks the amount the valve is open. Note that the *poll()* function, which will be responsible for stopping the motor when the valve is shut, must contain the lines:

```
void Valve::poll(void)
{
  ...

  if (amountOpen == desiredOpening) // valve has reached desired position
  {
    valveMotor->stop(); // stop motor

    valveMotor->commandOn(); // make sub-objects commandable again
    valveEncoder->commandOn();
  }

}
```

The **Hardware** class also provides the functions *getClassName()* and *getObjectName()* to find the class and object name of any **Hardware** object. Polling of a single object by name is provided by the static member function *void doPoll(char* objectName)* this is mostly useful for debugging.

A complete list of **Hardware** class functionality is as follows:

static void doCommand(char* commandString) match an object name to the first token in commandString and pass the rest of the string to that object's static *command()* function for further processing.

static void doPoll(void) call the *poll()* functions of all known **Hardware**-derived objects. The functions are called in the same order the objects were created.

static void doPoll(char* objectName) poll a single object by name

pollOn()/pollOff() change pollable state of an object

commandOn()/commandOff() change commandable state of an object

char* getClassName() return name of object's class as set by constructor

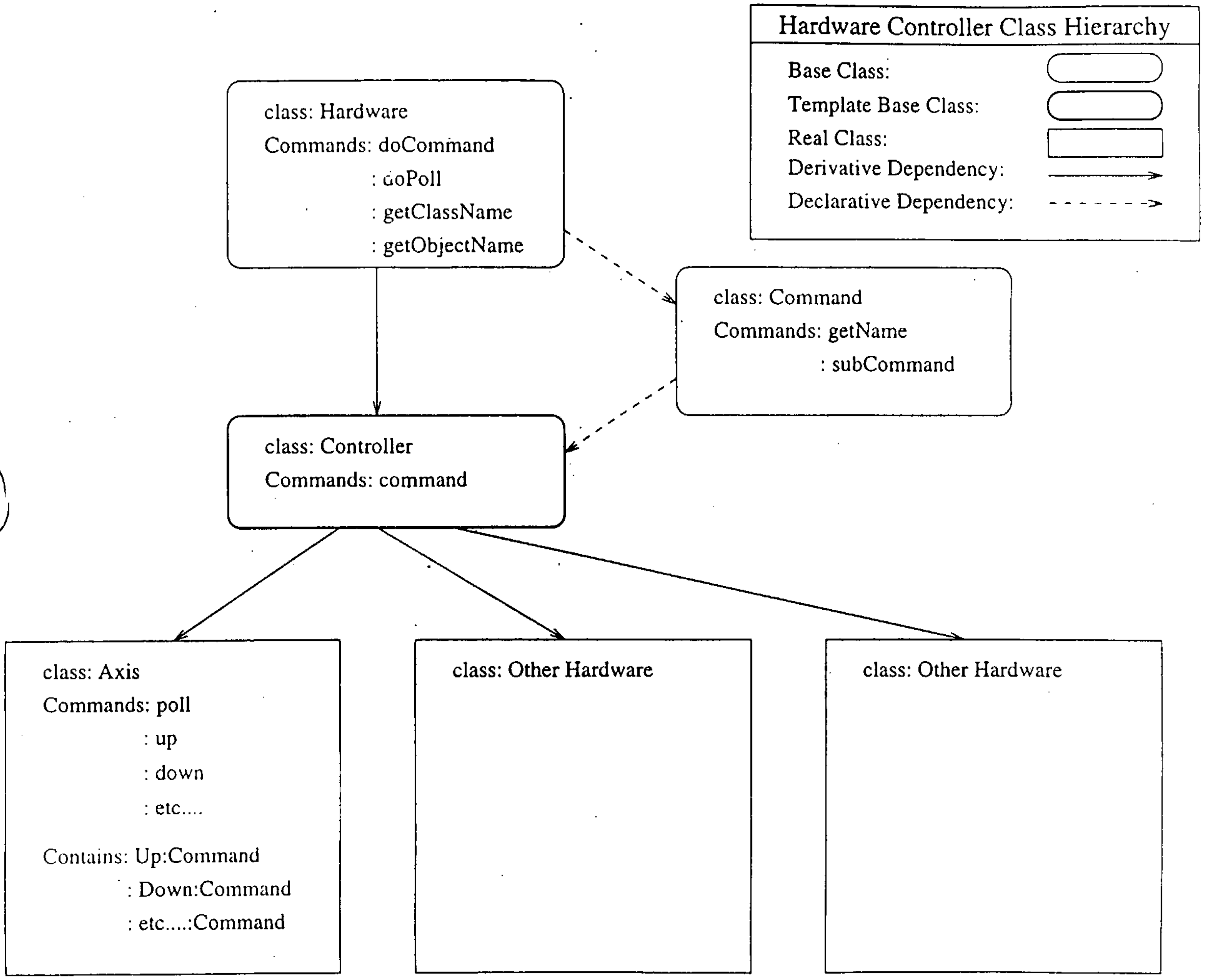
char* getObjectName() return name of object

5 Error Messages

The *doCommand()* function returns three standard strings:

- `HW_ERR_NO_OBJECT` could not find an object to match the first token in the command string
- `HW_ERR_NO_COMMAND` could not find a command to match the first token in the command string after the object name
- `HW_ERR_OK` so far as the **Hardware** class can tell the command ran ok

Individual **Hardware**-derived class member functions may return their own strings in place of these messages.


Hardware Controller Class Hierarchy
Base Class:
Template Base Class:
Real Class:
Derivative Dependency:
Declarative Dependency:
class: Hardware
Commands: doCommand
: doPoll
: getClassName
: getObjectName
class: Command
Commands: getName
: subCommand
class: Controller
Commands: command
class: Axis
Commands: poll
: up
: down
: etc....
Contains: Up:Command
: Down:Command
: etc....:Command
class: Other Hardware
class: Other Hardware