The AAO General Instrument Simulator Code

What it is, how it works, and how to use it

Keith Shortridge, (Keith@KnaveAndVarlet.com.au), June 18, 2018.

Introduction

Over the years, AAO has made increased use of detailed instrument simulators to help in the development of control software for its instruments. These are software programs that provide as accurate as possible a simulation of the hardware for the instruments, at least in terms of how an instrument looks to its control software.

What is an instrument?

When we talk about ‘simulating an instrument’, what is at the heart of an instrument? It’s a collection of connected optical, mechanical and electronic mechanisms. There are interfaces that connect this instrument to one (or more) computers, and there may be other things that are controlled other than through computer interfaces.

The computer interfaces may be purpose-built interfaces, they may be RS232 interfaces, in days gone by they might have been CAMAC or IEEE interfaces, nowadays they are often CANBus interfaces. The computer(s) talk to these through a relatively thin code layer that allows the control software to communicate with whatever hardware interfaces are being used. (There doesn’t have to be, but the days of inserting direct I/O instructions into high-level control code are long gone.) In most cases, this layer is actual driver code with the familiar set of calls (open(), close(), read(), write(), ioctl(), etc.).

The non-computer controlled items are also important; they all have an effect on the operation of the instrument. They may be physical switches: a power switch, an emergency stop button. There may be parts of the instrument that are changed by hand: there may be different gratings that can swopped in and out. Even the physical environment may change - the temperature in a spectrograph room, the pressure in a hydraulic system.

It turns out to be convenient to think of the instrument in these terms, as 1) the collection of physical mechanisms, 2) the computer interfaces and 3) those external things not under computer control. Reflecting this in the structure of the simulator leads to a much neater software design.

How do we want simulation to work within a control task?

Simulating the hardware comes down to so arranging things that when the control code makes its calls to talk to the hardware, it gets the same responses when running in simulation as it would when running with the actual hardware. It is possible to have the higher levels of the control code run in a ‘simulation’ mode, but in practice, the most detailed and most useful simulation comes from introducing simulation at as low a point as possible in the code, preferably as close to the driver level as possible. The important point is that when running in simulation there is always a point at which there is a test for simulation, after which the code takes different paths when simulating and when not, and it is only code up to that point that is actually tested when run in simulation. That means there should be as thin a layer after that point as possible. This is discussed in detail in a number of AAO papers on simulation. See (??).

Simulating the hardware in as much detail as possible, and at as low a level as possible, makes it possible to test control code fairly thoroughly in the absence of the actual hardware.

This allows:

o Control code to be tested well before the hardware is available.
o Contention for the hardware, once it does become available, to be reduced - software can continue to be tested while the hardware is being used for other tests.
o Control code to be tested on laptops, instead of needing machines with specific hardware interfaces.
o Code to continue to be tested once the hardware is being used for observations and so normally unavailable for software tests.
o More detailed feedback from testing than the hardware itself can provide, as it is possible to probe the state of the simulator in some detail.

AAO first used detailed simulation in this way for the TCS project, then for HERMES, Taipan, and GHOST, and it is now being used for 2dF. TCS used purpose-built hardware interfaces, but since then most AAO projects have used CANBus interfaces, at least for some of the hardware.

CANBus is complicated, but its use does mean that the interfaces to quite different instruments work in very similar ways. The realisation that the overall structure of most AAO simulators was very similar, particularly in the way CANBus was used, has led to the development of the General Instrument Simulator code, with the aim of making it much simpler to develop simulators for future instruments, starting with the CANBus version of 2dF.

The General Instrument Simulator comprises a set of base C++ classes, which control an instrument that does nothing. It has a CANBus, but there are no nodes on the CANBus, and nothing attached to them. However, it provides a large part of the more intricate code needed to control a real instrument, and it is relatively simple to write classes specific to an actual new instrument, inheriting from these base classes, to create a simulator for that new instrument.

An absolutely trivial CANBus example.

This section walks through the process of building and testing a simulator for a ridiculously simple instrument. The instrument is just called Ex1, for 'example 1'. It is as minimal as an instrument can be. It has a CANBus interface. It has no servo amplifiers, and just one I/O module. And that I/O module has just one input bit, and one output bit.

The input bit, which a control task can set through the CANBus, is given the symbolic name 'EnableInstrument'. The instrument starts up disabled. Setting this bit to 1 will enable the instrument. Setting it back to zero will disable the instrument.

The output bit can be read to tell if the instrument is enabled or not. Boringly, it is called 'InstrumentEnabled'.

What makes this a trivial example is that all the CANBus aspects of the instrument are handled by the existing code modules, working from a configuration file that specifies the layout of the CANBus for the instrument. Given this, the standard CANBus simulation structure can handle what happens when a control task sets the EnableInstrument bit, at least in terms of handling the messages, determining what CANBus node is involved, what bit is being set, and can determine that a bit called 'EnableInstrument' is being set. What the standard structure doesn't know, is the meaning of that bit - how it is used by the instrument, and what the effects on the instrument-specific hardware will be. In this case, the standard code has no way of knowing that when this bit is set, the instrument is (conceptually at least) enabled, and this should be reflected in the value of the bit called 'InstrumentEnabled'.

Responsibility for embodying interactions like this between the instrument-specific parts of the hardware falls on what we call the 'instrument simulator' module. Ideally, this should be all that is needed for a new instrument - at least, one that uses standard I/O devices like CANBus for which a general simulation structure already exists.

Finally, there needs to be a main program to tie all this together.

Put like that, it sounds quite simple. And, for a simple instrument like this, it is.

The instrument simulator

The GenInstSimulator ACMM module provides the necessary basic structure for a simulator task. It provides a class called GenInstSimulator which implements the instrument simulator for the simplest instrument of all - one that does nothing at all. But it does provide a structure that a simulator for an actual instrument can inherit.

For example, when the standard CANBus layer determines that a control task has set the value of a bit or set of bits (an 'item') in an I/O module, it calls this simulator method:

```
void GenInstSimulator::InstSetDigitalItem (const string& Name,
                                           unsigned int DataValue)
```

and it passes the symbolic name of the item, which it gets from the configuration file. The basic GenInstSimulator code for this routine, of course, has no idea what this signifies, and does nothing. But the simulator code for the Ex1 instrument, overriding this routine, can know that if the EnableInstrument item is set, it has to enable the 'instrument', and that it has to set the InstrumentEnabled item to reflect this:

```
void InstSetDigitalItem (const string& Name, unsigned int DataValue) {
   if (Name == "EnableInstrument") {
      I_InstrumentEnabled = DataValue;
      if (I_InstrumentEnabled) LogMessage ("Instrument enabled");
      else LogMessage ("Instrument disabled");
      NoteDigitalInputChange("InstrumentEnabled",I_InstrumentEnabled);
   }
}
```

I_InstrumentEnabled here is an instance variable for the class that maintains the setting of the 'InstrumentEnabled' I/O item. Yes, there is also an I_EnableInstrument variable that maintains the setting of the 'EnableInstrument' item, but the base class can know to set that. The one thing the general structure doesn't know is the relationship between the digital output item 'EnableInstrument' and the digital input item 'InstrumentEnabled'.

Note one very important point here. I'm calling InstrumentEnabled an 'input' item, and yet this code is setting it in response to a change that's been made to EnableInstrument, which I'm calling an 'output' item. I bet that sounds the wrong way around.

Here's the point. The hardware documentation is all written from the point of view of the control task and the hardware interfaces it uses. So EnableInstrument is something that the control task sets. To the control task, it's an output item. InstrumentEnabled is something the control task reads - to the control task, it's an input item.

But what's 'output' to the control task is an 'input' to the hardware (you probably aren't used to thinking of interfaces from the point of view of the hardware!) and so it's an 'input' to the simulator. And vice-versa. See that call to NoteDigitalInputChange() is the code snippet above? This will make the new value of 'InstrumentEnabled' available to the control task, which treats it as an input. But it's something the simulator has just set, and it's a value output by the simulator. Just remember, this nomenclature is always from the point of view of the control task.

(As an aside, this gets worse in the XWing level, where the simulator code has Write() routines that are invoked to READ a new value, because they're responding to a write() call made by the control task. Sometimes, I wish I hadn't done that, but I think the alternatives are worse. You really didn't need to know that in these introductory stages. But one day you may find yourself unlucky enough to have to look at XWing code.)

InstSetDigitalItem() is one of the few methods of the base GenInstSimulator class that the simple Ex1 instrument simulator needs to override. The Ex1Simulator is so trivial it can be all put into a .h file (something I usually prefer not to do, but that's a matter of personal style.) And here it is, with most of the comments removed:

```
#include "GenInstSimulator.h"
class Ex1Simulator : public GenInstSimulator {
```

```
public:
   Ex1Simulator (void) {}
   ~Ex1Simulator () {}

private:

   //  Load a map giving the names of the supported CANBuses and
   //  their .ini files.
   void SetUpCANBusNames (std::map<std::string,std::string> &CANBusNameMap) {
      CANBusNameMap["CAN0"] = "Ex1.ini";
   }

   //  Load the digital input name map with the names and variable
   //  addresses for the input items.
   void SetUpDigitalInputNames (std::map<std::string,
                                        unsigned int*> &DigitalInputNameMap) {
      DigitalInputNameMap["InstrumentEnabled"] = &I_InstrumentEnabled;
   }
   //  Load the digital output name map with the names and variable
   //  addresses for the output items.
   void SetUpDigitalOutputNames (std::map<std::string,
                                       unsigned int*> &DigitalOutputNameMap) {
      DigitalOutputNameMap["EnableInstrument"] = &I_EnableInstrument;
   }
   //  Reset the state of the simulated instrument.
   bool ResetInstSim (void) {
      I_InstrumentEnabled = 0;
      I_EnableInstrument = 0;
      I_PoweredOn = true;
      return true;
   }
   //  Returns true if there is power to the CANBus.
   bool CANBusIsPowered(void) {
      return I_PoweredOn;
   }

   //!  Handle instrument-specific aspects of a change to a digital output item.
   void InstSetDigitalItem (const string& Name, unsigned int DataValue) {
      if (Name == "EnableInstrument") {
         I_InstrumentEnabled = DataValue;
         if (I_InstrumentEnabled) LogMessage ("Instrument enabled");
         else LogMessage ("Instrument disabled");
         NoteDigitalInputChange("InstrumentEnabled",I_InstrumentEnabled);
      }
   }
   //  Set true (1) if the instrument has been enabled.
   unsigned int I_InstrumentEnabled;
   //  Set true (1) to enable the instrument.
   unsigned int I_EnableInstrument;
   //  An internal flag indicating that the overall system power is on
   bool I_PoweredOn;
};
```

You can find the full set of expanded comments in the file Ex1Simulator.h.

Let's have a quick look at this code. I hope you noticed that there isn't very much of it.

SetUpCANBusNames() is called by the simulation structure on startup. It needs to know how many CANBuses are being simulated, and the names of the corresponding configuration files. This instrument only has one CANBus, 'CAN0', described by the file 'Ex1.ini'. Using a map is one way of passing on this sort of information.

There are three instance variables, I_InstrumentEnabled, I_EnableInstrument and I_PoweredOn. I like to use the I_ prefix to indicate an instance variable, and you'll find that convention

throughout this code. I'll come to I_PoweredOn in a moment. It's convenient for a simulator to have an instance variable for each I/O item, which is what I_InstrumentEnabled and I_EnableInstrument are. These need to be kept in sync with the CANBus hardware's view of the digital I/O items. The simulation structure can help a lot here, if it knows what instance variables correspond to which I/O items. During setup, the simulation structure calls SetUpDigitalInputNames() and SetUpDigitalOutputNames() to pass it this information. It also calls ResetInstSim(), and one thing this should do is set initial values for all these variables.

We've already looked at InstSetDigitalItem(), but you may have wondered why it doesn't set the value of I_EnableInstrument to match the value of Data. The answer is that the simulation structure has already done that before InstSetDigitalItem() is called. In fact, NoteDigitalInputChange() could in principle use the same information to set I_instrumentEnabled, but it doesn't. I prefer to set these explicitly in the main simulator code - I realise there's some inconsistency here.

Which only leaves CANBusIsPowered(). If the CANBus is actually powered down, none of the real CANBus interfaces will work, and that's true for the simulated interfaces too. The low level simulation code doesn't know if there is power available - there might be a physical power switch on the instrument, and that's something we have to simulate. Here, we use I_PoweredOn to reflect the power state of the whole of the instrument, including the CANBus, and we pass its value back to the low-level code when it uses CANBusIsPowered() to find out the power status. Forgetting to provide CANBusIsPowered() is a trap; the GenInstSimulator base class has an implementation that plays safe and returns false, and that means nothing works, for reasons that aren't immediately obvious.

The .ini file

Now let's look at the configuration file. This is the file Ex1.ini (with comments removed, but otherwise complete):

```
[Node_1]
Type=IOModule
Digital8In= 1
Digital8Out= 1
Analog16In= 0
Analog16Out= 0
[Node_1_DigitalIn]
InstrumentEnabled= 0 0 1
[Node_1_DigitalOut]
EnableInstrument= 0 0 1
```

The .ini format is inflexible, and can look a bit odd, but it has sections introduced by a line [SectionName], and each section then provides a set of name-value pairs. (The values are strings, but can be interpreted as formatted strings, so multiple values can be accommodated, as here, where there are three integers associated with InstrumentEnabled and EnableInstrument.)

The .ini file has to have one section, called [Node_n] where n is the CANBus node number. For each node, it has to have a Type value. Here there is just Node 1, and its type is IOModule. IOModules can have a number of different 8-bit digital input and output groups, and can have a number of 16-bit analog inputs and outputs. Here, this instrument just needs one 8-bit input and one 8-bit output group. (7 of the 8 bits of each group will be unused, but that's OK).

The [Node_n_DigitalIn] section has one line for each digital input item. Usually, each item is one bit, but it is possible to group bits, so that a 4-bit item could take any of 16 different values. The names can be anything you like - although it's best to stick to letters and numbers, They are case-sensitive.

This .ini file lays down the names to be used for the various digital I/O items, and their layout in the I/O module. So,

```
[Node_1_DigitalIn]
InstrumentEnabled= 0 0 1
```

says that there is only one digital input item, on this one node, and it is called InstrumentEnabled. This name is how both the simulator code and the instrument control task will refer to this item. (They both make use of the .ini file.) The three numbers give the number of the 8-bit group, the bit in the that group at which the items starts, and the number of bits used by the item. So InstrumentEnabled is in the first (zeroth) group of 8 bits, it starts with the first (zeroth) bit and it is one bit long.

The layout and the function of these bits is usually determined by the electronics group building the hardware. The names used in the .ini file should be as close as possible to those used in the hardware documentation. (The names do need to be unique within the instrument, however.)

The main program

And, finally, we need a main program to tie all this together. For a really simple simulator like this, we can use a minimal main program. Here it is, taken from Ex1SimulatorProg.cpp, again with most of the comments removed:

```
#include "Ex1Simulator.h"
#include "GenInstSimulatorThread.h"

#include <stdio.h>
#include <unistd.h>

extern int main (int argc, char* argv[])
{
   Ex1Simulator TheSimulator;
   TheSimulator.Initialise();

   GenInstSimulatorThread TheSimulatorThread;
   TheSimulatorThread.SetSimulator (&TheSimulator);

   if (!TheSimulatorThread.Activate()) {
      printf ("Error starting the simulator thread.\n");
      printf ("%s",TheSimulatorThread.GetErrorText().c_str());
   } else {
      printf ("Simulator ready\n");
      pthread_join(TheSimulatorThread.GetThreadId(),NULL);
   }
   return 0;
}
```

It doesn't do much. It creates an instance of the Ex1 simulator - the code we've looked at already - and initialises it (this will call the routines we saw earlier like SetUpCANBusNames(), SetUpDigitalInputNames(), SetUpDigitalOutputNames(), and ResetInstSim()).

It then creates an instance of a GenInstSimulatorThread, sets the Ex1Simulator as the simulator for it to use, and starts the thread. And then it waits for that thread to exit (which it will when the simulator handles a closedown request - we'll come to that) and then exits.

To be honest, this trivial example doesn't really need to have the simulator run in a separate thread. However, most practical simulator programs run some sort of more elaborate

environment in the main thread (you can run a DRAMA main thread, and make the simulator a DRAMA task, or you can run Qt in the main thread and give yourself a fancy GUI; we'll come to why you might want to do that) and then you need a separate thread for the simulator. So the simulation structure is coded to support that, and even if you don't need it, you might as well make use of the standard setup code provided by GenInstSimulatorThread. (If you look at the setup code in there, you'll realise you really don't want to do this yourself unless you really have to.)

Building and running the program for the first time

There is a standard Makefile provided with the Ex1 code, which will build the simulator program, Ex1SimulatorProg, and a couple of test programs. We'll come to the test programs in a moment, because they show what basic instrument control code can look like. But for the moment, just use the Makefile to build Ex1SimulatorProg:

```
make Ex1SimulatorProg
```

The Makefile is a very straightforward Makefile, but it sets up a lot of include directories and a lot of library directories, all ACMM sub-systems, which you need to have already built. It probably won't build first time on your machine, but once it does:

```
./Ex1SimulatorProg &

Simulator reset
Simulator ready
```

And that's all it does when it starts up. It says it's ready.

If that built OK, the test programs will probably build OK as well. There are two, Ex1Read and Ex1Set.

```
make Ex1Read Ex1Set
```

if those build OK, we can try them out. Ex1Read is an instrument control program that uses CANBus to read the values of both the InstrumentEnabled and EnableInstrument I/O bits. (Yes, an instrument control task can in fact read the value of an output bit as well as write a value to it. It can't write to an input bit.)

```
./Ex1Read
Doing init
InstrumentEnabled = 0
EnableInstrument = 0
```

Which is what you'd expect. Initially the instrument isn't enabled. Ex1Set takes a single integer command line argument. If it's zero it clears the EnableInstrument output bit, otherwise it sets it. If we set it, and if the simulator is doing the right thing, the simulator should enable the instrument and set the InstrumentEnabled bit to show this.

```
./Ex1Set 1
Doing init
Instrument enabled
InstrumentEnabled has been set
```

The 'Instrument enabled' message is from the LogMessage() call made from InstSetDigitalItem() in the Ex1Simulator code.

and now:

```
./Ex1Read
Doing init
InstrumentEnabled = 1
EnableInstrument = 1
```

Which all might seem rather uninspiring as a demo, but you have just run two CANBus-based instrument control programs and had them talk to a CANBus-based instrument simulator.

We'll discuss how to shut down a simulator program later, but for the moment we can use the Closedown utility built into the XWing sub-system. This is neater than just using 'kill' on the process, although that works as well. In this case, Closedown needs no command line arguments, it's enough to just run it. (Closedown makes use of an internal feature of the XWing I/O simulation system that underlies the whole AAO simulation code.)

Now we need to look at what's inside those instrument control programs.

Making an instrument control program work with a simulator.

This document isn't about explaining how to write an instrument control task, so it isn't going to go into a lot of detail about how a program like Ex1Set uses the CML library to access CANBus devices. But we do need to know how to switch that library so that it can interact with either the simulator or the real hardware.

One thing to point out first: the way this is packaged at the instrument control task level is slightly different when dealing with CANBus devices than with more specialised devices. The underlying principles are the same, and in all cases they involve inserting a test ('are we simulating?') at as low a point as possible in the code. With a specialised device, this usually involves putting a wrapper around the driver calls to that device, and this is described later. With CANBus, it involves telling the CML library to use such a wrapper layer, and there is now a packaged way to do this.

The ideal, of course, is that the code run by an instrument control task talking to a simulator should be exactly the same as that run when talking to real hardware. Realistically, 'exactly' the same is not feasible, but what differences there are should be as low down in the code as possible, and as simple as possible.

Here's the basic code for Ex1Set, with comments and most error checking code removed:

```
#include <cstdio>
#include <cstdlib>
#include "CML.h"
#include "SimCanInterface.h"
#include <unistd.h>
#include "CANBusConfigurator.h"

CML_NAMESPACE_USE();


const char *canDevice = "CAN0";     // Identifies the CAN device, if necessary
const char *server = "";            // Identifies the server, if necessary

int main(int argc, const char* argv[])
{
   bool Set = false;
   if (argc > 1) {
      if (atoi(argv[1])) Set = true;
   }
```

```
    cml.SetDebugLevel( LOG_EVERYTHING );
    SimCanInterface can( canDevice , server);

    CanOpen canOpen;
    const Error *err = canOpen.Open( can );
    CANBusConfigurator TheConfigurator;
    bool OK = TheConfigurator.Initialise("Ex1.ini");
    if (!OK) printf ("Error opening configuration file\n");

    IODataType Type;
    unsigned int NodeId,StartBit,Bits;
    OK = TheConfigurator.GetIOItemData("EnableInstrument",
                                       &NodeId,&Type,&StartBit,&Bits);
    if (!OK) printf ("Cannot get details for EnableInstrument\n");

    unsigned int Group = StartBit / 8;
    StartBit = StartBit % 8;

    IOModule module;
    printf( "Doing init\n" );
    err = module.Init( canOpen, NodeId );
    uint8 Data = 0;
    err = module.Dout8Read (Group, Data);
    uint8 Mask = 1 << StartBit;
    if (Set) Data = Data | Mask;
    else Data = Data & ~Mask;
    err = module.Dout8Write (Group,Data,true);
    printf ("InstrumentEnabled has been %s\n",Set ? "set" : "cleared");

    return 0;
}
```

The full code can be found in the Ex1Set.cpp.

If you aren't used to talking to CANBus devices using the CML library, the details of what is going on here will be a bit of a mystery, of course. CML provides a CanOpen class, instances of which are used to handle access to the CANBus network. It also supplies an IOModule class, instances of which can talk to individual I/O CANBus nodes. Here, canOpen is an instance of the CanOpen class, and module is an instance of the IOModule class. The code calls the Dout8Write ('Digital Output 8-bit write') routine of module to write to the I/O module. (Because this sets all 8 bits in the group, it first reads them, then sets just the one bit it cares about, then writes out all 8 bits.) What goes on under the hood of CML is highly multi-threaded and complicated, but that's why we use libraries like CML - to hide all the complexity.

Ex1Read looks almost the same, except that it reads both the bits we care about and prints out their values instead of trying to change them. Note that both programs use a CANBusConfigurator object to read the Ex1.ini file to find out the low-level details of the named I/O items they're interested in.

Note that at the start of the code, there is a call to the Open() routine of the CanOpen device, which passes it a reference to an instance of a CopleyCAN class - the object called 'can' in this code. CML is quite nicely structured so that the main CML code is independent of the actual CANBus interface used. Originally, AAO used Copley-built, then moved to Anagate interfaces, and then to CANBus-over-ethernet interfaces.

To make this work, the CanOpen class performs all its actual I/O via calls to an instance of a CanInterface class, which provides a set of routines for reading from and writing two an CANBus. Because all this is in C++, an instance of any class that inherits from CanInterface can be used instead, and this is how differences between the various possible interfaces are handled.

So the CopleyCAN class inherits from CanInterface and is able to work with the Copley-supplied CANBus hardware. There are equivalent classes that work with Anagate or ethernet-based hardware.

And - here's where simulation comes in - if you want to talk to a simulated CANBus, there is a class called SimCanInterface that can talk to a simulator task, and not to real hardware at all.

To run the above program using a Copley interface, all you have to do is replace the line:

```
SimCanInterface can( "CAN1" );
```

with a line that declares 'can' to be of type CopleyCAN.

and the whole program will work with the real hardware. (Actually, different devices do have to initialised in specific ways, and a CopleyCAN also needs a call to set its baud rate.) In a serious instrument control task, there would normally be a thin layer that hides this bit of trickery from the top level code. Also, it is possible to implement partial simulation by having multiple CANBuses, some simulated and some using real hardware. But that's getting complicated.

CANBus-based systems are a specific case of a type of device that can be handled in simulation. However, they are sufficiently important, particularly for AAO instruments, that there is a whole layer of simulation code, such as the SimCanInterface, that handles them. This is, of course, built upon a much more general layer of simulation code, and that brings us to the XWing sub-system.

Simulating the outside world

We can expand the Ex1 instrument code in many ways, but this is a good place to remember that an instrument exists in a physical environment, and that things may affect it that are not under computer control, but which affect the way it will respond to an instrument control program.

One good example is the question of power to the instrument CANBus. As mentioned earlier, if this is turned off, nothing works.

Try the following experiment. In the code for Ex1Simulator, change the line in ResetInstSim() that simulates starting up with the power turned on:

```
I_PoweredOn = true;
```

to:

```
I_PoweredOn = false;
```

rebuild the simulator and try to run it. The simulator will start up, but when you try to run one of the test programs, Ex1Set or Ex1Read, nothing will happen. Thee will be an error message to the effect that there was a timeout trying to talk to the CANBus. Which is what you'd expect; there is a quite complicated protocol (CANOpen) used to communicate over a CANBus, with messages and responses, and if the instrument has no power to its CANBus nodes they won't reply and the CML library will report a time-out.

(At this point, a good place to start looking for the cause of the problem is the CANOpen..log file written by the simulator. This logs all the traffic on the simulated CANBus, time-stamped to a tenth of a millisecond. In this case, you should find Node1 reporting that it is not responding to a CANBus message because it is powered off. At least, it does if you're using a version of the

SimCanOpen module later than version 1.63. Earlier versions didn't log this problem, leading to a lot of time being wasted trying to debug an obvious problem.)

The instrument control task has no way of switching on the CANBus power. In real-life, it needs someone to plug it in to the wall and work the on/off switch. And we need to simulate that.

This is why simulators generally need GUIs. We can write a GUI with a nice fancy widget that looks like a power switch that we can turn on and off. It's good to be have something like this to play with, because it actually encourages software developers to see what happens when they flick these switches - and that means they test how the instrument control software reacts to these externally-generated conditions. You can have a slider that controls the 'temperature' of the instrument, or the 'hydraulic pressure', and see what happens when these go out of limits. Does the instrument control software do something sensible, or had this case simply been forgotten about?

We're not going to write a GUI now. We're going to write a command line program that can twiddle with the power setting for the simulator. But whatever we use, standalone program, GUI, whatever, it has to have a way of interacting with the simulator, and we can use the same mechanism in both cases.

The GenInstSimulator class provides a 'hook' that can be used for this sort of communication. It has a routine called InstHandleEvent():

```
bool InstHandleEvent (const std::string& Name,int NArgs,const std::string Args[])
```

The default implementation does nothing, as you might expect, but this is a routine with fairly flexible interface that can be used for a large number of purposes. (It's the same interface as a command line program, and they seem to manage quite well.)

It should be pretty obvious how you'd write a version of InstHandleEvent() that could handle two events named "PowerOn" and "PowerOff". (Hint: they set I_PoweredOn to either true or false.) Or you could have an event called "Power" with arguments "On" or "Off" or 0 or 1 or whatever you fancy. (I've occasionally implemented commands equivalent to "TogglePower" but somehow they never seem to be really useful, except in the unlikely even that the actual hardware really did have a spring-loaded button that just toggled the power.)

But how do we arrange to call the simulator's InstHandleEvent() from outside the simulator?

The recommended way uses what's called a CommandStream device. This is a fictitious device called "Command" that doesn't have a real-world counterpart, but which is supported by the simulator for just this purpose. It is a text device that you can write to. The low-level simulator code can receive messages sent to this device, and it passes them by passing them on to the simulator's InstHandleEvent() routine.

There is a class called a CommandStreamWrapper which provides Open(), Write(), and Close() calls, which is all you really need. A program to send a "PowerOn" action to the simulator could look like this:

```
#include "CommandStreamSimDriver.h"
#include <string.h>
int main()
{
   CommandStreamWrapper TheWrapper;
   TheWrapper.Open("Command");
   char* Command = "PowerOn";
   TheWrapper.Write(Command,strlen(Command) + 1);
   TheWrapper.Close();
```

```
    return 0;
}
```

or you could make Command “Power On” and this would send the ‘Power’ command with ‘On’ as an argument.

There is a small program called SendCommandToString that provides a ready-made command line program that sends it command line arguments to the simulator as such a command. It’s essentially the same code as above, but with error checking and which allows arguments to have spaces if enclosed in quotes.

Armed with this, you can leave the Ex1Simulator powered off by default, and can then fix a failed attempt to run a test program, say Ex1Read, by giving the command:

```
SendCommandToStream PowerOn
```

This is a very flexible mechanism. It allows external conditions to be simulated easily using a command line interface, but exactly the same interface can be used for GUIs - a GUI can respond to GUI events by sending commands to the simulator via a CommandStreamWrapper. How the simulator sends information out to a GUI is another matter, to be discussed later.

An example of this sort of interaction has been added to the Ex2Simulator code, described below. The Ex2 example involves CANBus amplifiers, and taking power from an amplifier introduces some additional complications that need to be handled.

Updating the simulator ‘model’ on a regular basis

You can think of the instrument simulator as maintaining a ‘model’ of the instrument. It should have variables describing all the various components of the instrument and their current state. The I_EnableInstrument and I_InstrumentEnabled flags, together with the I_PowerOn flag, constitute the ‘model’ of the Ex1 instrument. Things happen - a power switch being operated - an enable bit being set by an instrument control task, and as a result the model gets updated.

Sometimes, the only thing that changes is the passage of time. There may be mechanisms that are not servo-amplifier controlled, but are simply set moving when an instrument control task sets a bit, and stop moving when they get to a certain point - perhaps when they hit an end stop. To keep the model of the instrument updated under these circumstances, we often want to run a general ‘update’ routine on a regular basis. This often turns out to be useful for many things, particularly updating a GUI.

The GenInstSimulator code calls a routine called InstUpdate() on a regular basis. It is called whenever something happens at the I/O level, and this is usually enough, particularly on CANBus systems where the underlying CANBus is constantly active. On other systems where low-level I/O is more intermittent, it is possible to set a maximum time that is allowed to elapse between calls to InstUpdate(). The default maximum time set by the GenInstSimulator code is a tenth of a second. (For those who have to know, it’s set by the thread startup code in GenInstSimulatorThread.cpp - look for a call to ServerIface.SetTimeoutCallback()). Updates may happen more frequently than that, and it is common for InstUpdate() implementations to check the elapsed time since the last actual update was run and return immediately if this is too short to worry about.

The trivial Ex1 instrument doesn’t need to override the null default implementation of this routine, but most actual simulators do.

CANBus IO Module summary

There really isn't much more to handling I/O bits through CANBus I/O modules.

In general, most of what you do with digital I/O bits in the instrument simulator code is:

o Respond to the CANBus telling you that an instrument control task has set the value of one of the named digital outputs. The GenInstSimulator code calls InstSetDigitalItem() when this happens, and the simulator should override this routine with one that knows how to respond to changes to the various digital outputs.

o Initialise and maintain the values of the various digital input items (items that are inputs to an instrument control task, which will read these as it needs). To set such a digital input item, call NoteDigitalInputChange().

It's worth summarising what you have to do to add support for a new digital I/O item.

o Add an entry for the item in the simulators configuration (.ini) file, showing which node it is associated with, its 8-bit group, start bit within that group, and number of bits. (At the moment, there is no support for multiple bit items that cross group boundaries, and no AAO hardware has had such things.) If a new I/O module has been added, a new Node_n section will be needed in the .ini file to describe it.

o Add an instance variable corresponding to the digital I/O item to the simulator's class definition in the .h file. Initialise this in the ResetInstSim() routine. (This is called as part of the Initialise() call that you can see made in the main simulator program, so you don't really need to initialise it explicitly in the constructor, but you can if you like. But it does need to be set to a default value in ResetInstSim() in case the simulator is ever reset.)

o Add this item and its associated instance variable to the map set up by either SetUpDigitalInputNames() or SetUpDigitalOutputNames(). This allows the GenInstSimulator code to handle a lot of the housekeeping required.

o If this is an output item (one written by the instrument control task), add code to InstSetDigitalItem() to handle what happens when the instrument control task changed its value.

o Add whatever instrument-specific code is needed to handle these items. This may be something that needs to go into the InstUpdate() implementation. If the instrument control task sets one of its output items, things may happen as a result in the instrument, and these have to be simulated. Such changes - or other things happening in the instrument, may result in changes to one of the items used as inputs to the instrument control task, in which case the associated instance variable needs to be updated, and NoteDigitalInputChange() must be called to ensure the CANBus layer simulates a read of the this item correctly.

That really does cover it. About the only other thing you can do is check the value of a named digital input, to see what value the instrument control task would get if it were to read it. Usually, you know this because you have that value in the associated instance variable, but there may be structural reasons for getting the value from the CANBus simulator instead. However, this isn't packaged nicely by the GenInstSimulator class. If you want to do it, you have to get access to the 'go-between' used to interface to the CANBus sub-system and then call its IOModuleCheckDigitalInput() routine. We've not yet covered the 'go-between', and it's mentioned here really just to point out that it is possible to do unusual things like this if you understand and are prepared to use some of the lower-level components of the simulator.

CANBus amplifiers and servo mechanisms

The other things that are usually controlled through CANBus are servo amplifiers. Indeed, one of the big attractions of CANBus is the availability of servo amplifiers built into CANBus nodes which can handle most of what's involved in moving instrument mechanisms. The CML library has a comprehensive set of amplifier control routines that can get amplifiers to move mechanisms from one point to another, following pre-determined trajectories if necessary, in combination with other amplifiers if necessary to provide a 'blended' move in a multi-dimensional coordinate space (moving in X,Y and Z at the same time, say).

Surprisingly, although CANBus amplifiers are complicated, most of what they do is sufficiently pre-determined that the simulated amplifiers built into the CANBus simulation module (SimCanOpen) can handle most of what's needed without the top-level instrument simulator code having to get involved at all. The .ini file entry for an amplifier contains a lot of details about how the amplifier should be operated, but most of this is for the benefit of the instrument control tasks that use the .ini file. All that the amp simulator code uses are the node Id, the amplifier name, and the hardware encoder limits.

An instrument control task sets an amplifier moving, having told it the coordinates and the parameters (velocity, acceleration, etc) for the move. With this and the limit information it gets from the instrument configuration file,the simulated amplifier has enough information to simulate the movement without help from the instrument simulator. What it does have to do is keep the instrument simulator informed of what is happening. The instrument simulator needs to know what the mechanism is doing - it may need to display the movement on a GUI, or it may be that other things depend on the movement - one mechanism moving past a certain point may trigger an interlock that can be read by the instrument control task (meaning a digital input may have to be set), or may prevent the movement of another mechanism.

Another possibility is that the instrument simulator may want to interfere with the movement of the mechanism. Usually this doesn't happen unless something has gone wrong, but if the instrument control task ignores an interlock and tries to move an interlocked mechanism then the instrument simulator has to intervene and stop the movement.

Most of this is handled either in InstUpdate() or by the InstAmplifierActive() method of the GenInstSimulator, another routine where the default implementation does nothing, and is intended to be overridden by a real instrument simulator. This is called repeatedly as the mechanism moves, and is passed a structure showing the details of the movement. Normally, InstAmplifierActive() returns true. It's unusual for InstAmplifierActive() to want to interfere with the amplifier movement, but if it does, it can change the contents of the amplifier details structure it is sent and should then return false.

For the purposes of updating a GUI about the progress of a movement, the simplest thing is to put that notification into InstAmplifierActive(), given that this is called on a regular basis when a servo-controlled mechanism is moving. How you pass that notification to the GUI is something we're going to come back to (it involves the 'interactor').

An alternative is to add code to InstUpdate that loops through all the amplifiers in the system, using calls to the 'go-between' (described later) to get the position of each mechanism and to pass this value to the GUI, in the same way.

A servo amplifier example

We can add a single amplifier to the Ex1 instrument. This will make it the Ex2 instrument. This has a single servo mechanism called 'XAxis'. We can imagine it moving something along an X axis. We don't need to make many changes to support this.

First, the .ini file.

There are a number of parameters used to characterise an amplifier in most .ini files, but most of these are there for the benefit of the instrument control tasks, and we've not included them here. Here is our minimal entry for the XAxis amplifier:

```
[Node_2]

Type=Amplifier
Name=XAxisServo
HardwarePosLimit= 150000
HardwareNegLimit= -2000
```

Now the simulator code:

We can use the Ex1 code as the basis for this. We need to change the name of the ,ini file used, so SetUpCANBusNames() now becomes:

```
void SetUpCANBusNames (std::map<std::string,std::string> &CANBusNameMap) {
   CANBusNameMap["CAN0"] = "Ex2.ini";
}
```

and we need to add an instance variable and to override a few more of the default GenInstSimulator methods.

We need to add a structure to the instance variables to maintain the amplifier state:

```
//  The X-axis servo amplifier.
AmpStateType I_XAxisServoState;
```

We need to override the SetUpAmplifierNames() routine, just as we do the equivalents for Digital input and output items. We pass it the name of the amplifier and the address of the structure instance variable for the amplifier.

```
void SetUpAmplifierNames (std::map<std::string,AmpStateType*> &AmplifierNameMap)
{
   AmplifierNameMap["XAxisServo"] = &I_XAxisServoState;
}
```

We need to override two enquiry routines, one that checks for an interlock that may prevent the amplifier from operating, and one that checks that the amplifier is powered on. We assume the same instrument power setting applies to the servo amplifiers as to the CANBus as a whole - ie it depends on I_PoweredOn. (Actually, the default implementation of AmpIsInterlocked() returns false, but we may as well introduce this routine here.)

```
bool AmpIsInterlocked (std::string &/*AmpName*/)
{
   return false;
}
bool AmpIsPowered (std::string &/*AmpName*/)
{
   return (I_PoweredOn);
}
```

And that should be all we need to run the amplifier. (And we don't really need AmpIsInterlocked().) However, we probably do want to supply a version of InstAmplifierActive(), so that the instrument-specific code is aware of what the amplifier is doing.

```
//  Instrument-specific amplifier movement handling.
bool InstAmplifierActive (const std::string& AmpName, AmpStateType* AmpState)
{
   //printf ("Amp: %s now at position %f (%s)\n",AmpName.c_str(),
   //     AmpState->CurrentPosition,AmpState->Moving ? "moving" : "stopped");
   return true;
}
```

The printf() call here is commented out, because in practice it produces too much output from the simulator. But it should give an idea of what is available to the routine. The full definition of the AmpStateType structure can be found in GenInstSimulator.h. Most of the comments there should be clear enough - LastParameterPosition is provided so that the last position reported via a GUI or other interface can be recorded, so reporting trivial changes can be minimised. Conversion between encoder position and physical units can be performed by the underlying simulator code if the .ini file includes values for encode scale and offset.

The Ex2Simulator.cpp example code is based on the Ex1Simulator.cpp code, but adds these routines to enable control of the amplifier, and also adds a version of InstHandleEvent() that responds to 'PowerOn' and 'PowerOff' commands, as described earlier.

We need a new version of the main program for the simulator, since Ex1SimulatorProg.cpp creates and runs an Ex1Simulator. Ex2SimulatorProg.cpp is exactly the same as Ex1SimulatorProg.cpp, except that it uses the new Ex2Simulator - just one line changed,

You can now startup Ex2SimulatorProg:

```
./Ext2SimulatorProg &
Simulator reset
Simulator ready
```

There is a standard test program provided as part of the ACMM SimCanOpen module that can probably be used to test the amplifier code in the simulator. If the SimCanOpen directory is at the same level as the directory containing the examples, you should be able to use:

```
../SimCanOpen/Tests/simio 2 CAN0
```

(Note that, as a general test program, simio needs to be told the node number and the CANBus in use for the amplifier.) This should home the amplifier, and take it through a series of test moves:

```
Doing init
XAxisServo: Distance to move 0.000000 counts
XAxisServo: Estimated move time 0.25 secs
Waiting for home to finish...
XAxisServo: Home. Actual and current positions now both 0
Setting up moves...
Moving to 16807 using s-curve profile
XAxisServo: Distance to move 16807.000000 counts
XAxisServo: Estimated move time 0.86 secs
Moving to 75249 using trap profile
XAxisServo: Distance to move 58442.000000 counts
XAxisServo: Estimated move time 2.16 secs
Moving to 50073 using s-curve profile
XAxisServo: Distance to move 25174.000000 counts
XAxisServo: Estimated move time 1.42 secs
Moving to 43658 using trap profile
```

```
XAxisServo: Distance to move 6417.000000 counts
XAxisServo: Estimated move time 0.72 secs
Moving to 8930 using s-curve profile
XAxisServo: Distance to move 34728.000000 counts
XAxisServo: Estimated move time 1.67 secs
Amp is software enabled and hardware enabled, mode 1e01
Disabling amp
Amp is software disabled and hardware enabled, mode 1e01
```

(I said 'should' a couple of times when describing simio. You may find that simio has not been built - there is a separate set of Makefiles for various systems in the Tests subdirectory of SimCanOpen. You may find that simio doesn't work quite as advertised here - it tends to be modified from time to time to test different aspects of amplifier performance, and it does make some assumptions about the encoder range for the mechanism - you can see that it moves to specific encoder count values.)

You can try removing the comments in InstAmplifierActive() to see what gets reported to the instrument-specific code, but it's too much to be shown here.

The Ex2Simulator.h code also implements the external PowerOn and PowerOff commands, but before we describe this, it's time to finally explain the role of the 'Go-Between'.

The Go-Between

A go-between is an intermediary. (Google for 'go-between' and you'll find a 1953 novel by L.P. Hartley, made into a 1971 film by Joseph Losey.)

The instrument simulator code works at a slightly abstract level when it comes to CANBus amplifiers and digital and analog I/O items. It uses (hopefully) meaningful names for them, rather than having to worry about which node they refer to, or the detailed bit allocations for those nodes. As much as possible, it knows nothing about the configuration of the I/O sub-system used by the instrument control tasks to communicate with the instrument. (Note that I think of the 'instrument' as the mechanical and optical parts of the instrument, and the higher-level electronics that controls them. I see the lower-level hardware interfaces - CANBus, serial lines, etc - as an interfacing detail that is separate from the 'instrument'.)

The general-purpose CANBus simulation layer, on the other hand, knows all about the details of the nodes and the bits. But it has no understanding of the meaning of these items. It has no way of knowing that bit 5 of group 3 on node 2 controls the instrument shutter, or that amplifier node 6 moves the Z-axis of a fibre positioner, and it wouldn't know what to do with that information if it did know.

The Go-Between is a block of code that sits between these two layers and acts as both intermediary and interpreter. There are two distinct parts to its API.

o It provides a standard set of calls that the CANBus layer expects it to provide. (This is defined by the CanSimGoBetween class, and the CANBus layer expects any Go-between to inherit from this.) For example, there is a routine IOModuleDigitalOutput() that the CANBus layer will can whenever a digital output item is set by an instrument control task. The parameters are BusAndNodeId, StartBit, Bits, and DataValue, which are things the instrument simulator would rather not be bothered with. There is also an AmplifierActive() call. The GoBetween will turn these into calls to the instrument simulator, and a simulator based on the GenInstSimulator (bear in mind that simulators do not have to be based on this, but it makes things much easier if they are) will turn these into the calls to InstSetDigitalItem() and InstAmplifierActive() that we've already seen. There is also a standard IOModuleDigitalInput() routine provided for the simulator

layer to use when it sets a bit for the instrument control task to read (this is used by NoteDigitalInputChange()).

o It provides a set of routines for the instrument simulator to use, which can be varied to meet the needs of the instrument simulator. This part of the go-between is entirely up to the writer of the instrument simulator, since they are, in principle, also the writer of the Go-Between.

For most instruments that use this design - HERMES was the first - there was an instrument-specific go-between written in conjunction with the instrument simulator. With the advent of the GenInstSimulator code, there is now a GenInstSimulatorGoBetween class that is intended to provide the facilities needed by the GenInstSimulator, and, possibly, any simulator that inherits from GenInstSimulator. If it turns out that the GenInstSimulatorGoBetween does not provide what's needed, then it should be straightforward to inherit from it and add the necessary functionality.

In practice, the line between the instrument simulator and the go-between can be a little blurred. The go-between code makes use of a 'Configurator' to access the low-level details of the hardware layout. (The code for the Ex1Set and Ex1Read test programs used a Configurator to get the bit allocations for the digital I/O items.) The Configurator is the way a program accesses the information in the .ini file. Most of what a go-between does involves working with a Configurator to understand the relationship between the hardware layout and the symbolic names used for I/O items and amplifiers, etc., and also to find out the overall structure of the hardware - how many CANBus nodes are ther? How man of them are amplifiers, and which ones?

Ideally, the instrument simulator would not need to access the Configurator, as the go-between would do that. However, in places, historical awkwardnesses or a desire not to complicate code have meant that instrument simulators make use of a Configurator directly. Perhaps it would be better if all code that used a Configurator were to be at the Go_Between level?

Additional complications come from the use of non-CANBus hardware, and the use of multiple CANBuses. In the case of HERMES, there was one CANBus, but also non-CANBus devices, such as serial line vacuum gauges. HERMES put the details of both CANBus and non-CANBus devices into the .ini file, and used a Go-Between that provided facilities for interaction with vacuum gauges as well as for CANBus devices.

TAIPAN and GHOST used multiple CANBuses, but no non-CANBus devices (I believe this is the case). Here, there was one .ini file for each CANBus, and a 'network configurator' was introduced, which coordinated the various individual CANBus Configurators. (The GenInstSimulator code uses a network configurator, even in cases where the network consists of only one CANBus. That works perfectly well.) An unresolved question is the best way to handle cases where there are multiple CANBuses as well as non-CANBus devices.

(Note that, just as there is a CanSimGoBetween class that defines the Go-Between interface expected by low-level CANBus simulation code, there is also a VacSimGoBetween class that does the same job for serial line vacuum gauges. The GenInstSimGoBetween in fact inherits from both.and it may be that this can be extended neatly to additional purpose-built devices for specific instruments.)

Having explained the role of the Go-Between, we can now look at what happens when power is taken away from a servo mechanism.

Enabling and disabling servo amplifiers

Putting a couple of things together, the Ex2Simulator.h code also implements the external commands 'PowerOn' and 'PowerOff'. The code is fairly straightforward:

```
bool InstHandleEvent (
   const std::string& Name,int /*NArgs*/,const std::string /*Args*/[])
{
   bool ReturnOK = true;
   GenInstSimulatorGoBetween* GoBetween = GetGoBetween();
   if (Name == "PowerOn") {
      if (!I_PoweredOn) {
         I_PoweredOn = true;
         LogMessage("Instrument power turned on");
         if (GoBetween) GoBetween->EnableAmplifierMotions();
      }
   } else if (Name == "PowerOff") {
      if (I_PoweredOn) {
         I_PoweredOn = false;
         LogMessage("Instrument power turned off");
         if (GoBetween) GoBetween->DisableAmplifierMotions();
      }
   } else {
      LogError("Unexpected event %s",Name.c_str());
      ReturnOK = false;
   }
   return ReturnOK;
}
```

but note that we have had to make use of a direct call to the Go-Between. The problem is that the low-level amplifier simulator code needs to have the amplifiers explicitly disabled when the power is removed. It isn't constantly checking using AmpIsPowered(), and in any case needs to be told immediately if something as dramatic as removal of power from an amplifier happens. The Go_Between provides two useful calls: DisableAmplifierMotions() and EnableAmplifierMotions(), but these have to be made directly to the GoBetween.

This also needs the simulator to have the additional include line:

```
#include "GenInstSimulatorGoBetween.h"
```

This is easy enough. The GenInstSimulator has a routine GetGoBetween() which returns a pointer to the Go_Between in use (best always to test this for NULL, by the way), and the rest follows easily enough.

Turning power off will now stop the simove test from working.

(It has to be admitted that in testing this example, things don't work quite as nicely as they should. Turning the power off certainly has the expected effect, but the system doesn't recover nicely when power is turned back on. Some things get tested more than others. On AAO instruments, amplifiers are constantly enabled and disabled (this happens in HERMES during observations, because the amplifiers use optical encoders), which is why the Go_Between provides these routines, but power is not often removed dramatically from the system.)

Interactors

Something we've not mentioned until now is how the simulator thread communicates with the outside world. An important point here is that the GenInstSimulator code makes no assumptions about what is outside the thread it runs in. It knows nothing about its outside environment. It might be running as a simple command line program, in the way we've run our example programs so far.

Most AAO simulators run as DRAMA tasks. DRAMA is the AAO main instrumentation software environment, and it is possible to build a simulator with a DRAMA main thread and the separate instrument simulator thread. In this case, the DRAMA task can run its own GUI directly, using Tcl/Tk features, or it can set DRAMAM parameters that can be monitored by a separate DRAMA simulator GUI task.

Alternatively, for 2dF, there have been experiments where the simulator is built as a Qt program, with the main thread running a Qt GUI, using OpenGL calls to display a 3D model that reflects the current state of the instrument.

This means that the instrument simulator code can't even assume that a printf() call will be seen by the user. (They aren't when the Qt program is built using XCode as an OS X application, since these run without a terminal on which to display the printf() output, so this isn't just a theoretical consideration.)

This means that all simulator interaction with the outside world has to go through an abstract interface that can be implemented in different ways depending on the environment in which it runs. The GenInstSimulator code defines a fairly basic class called a GenInstSimInteractor. This provides calls like LogEvent(), LogError(), LogMessage() etc. These take a single string argument. (The LogMessage() calls we've used in the example code are to the GenInstSimulator's own LogMessage() routine, which presents a printf() style interface. This routine formats the final string to be output and uses the interactor's LogMessage() to output that string.)

Routines like LogMessage() are used so often that it's worth having a higher level version implemented in GenInstSimulator. Other calls are less common, and the usual way to make them is to use GetInteractor() to get a pointer to the Interactor being used (test it for NULL!) and then use it to call the interactor routine directly.

A useful set of routines provided by the interactor are all called SetNamedValue(), and allow named entities to be set to double, long, bool, or string values. How the environment makes use of these is up to it.

The GenInstSimInteractor interface also provides an Exit() call that can be used to shut down the simulator task. (This is the proper alternative to using the XWing Closedown utility.)

Note that thread to thread communication is potentially tricky. You could have a GUI thread, or a DRAMA thread, where the code provided a LogMessage() call. You might think of writing an interactor whose implementation of LogMessage() would simply call this LogMessage() routine provided by the code in the other thread. This might be rash. That LogMessage() routine would execute the code that forms part of the other thread, but it would be in the context of the simulator thread. This is probably not a good idea.

To address this, the GenInstSimulator module provides a class called GenGuiInteractor, which inherits from GenInstSimInteractor. This implements all its routines by sending messages in a defined format through a pipe. The idea is that the GUI thread will monitor this pipe, and will read the messages and act on them in its own thread context. To simplify this, the GenInstSimulator module also provides a GenGuiReader class that can be used to read these messages from the pipe, classify them, and provide the parameters supplied in the original call made by the simulator. After that, it's up to the GUI thread.

This is still all under development, but the intention is that specialised interactors can inherit from either GenInstSimInteractor or GenGuiInteractor and provide additional facilities as specific instrument simulators need them. To date, the ability to set specific named entities to values has proven a very simple and flexible interface that can accomodate a great many needs.

The Interactor interface also provides a GetInteractorClass() call that allows an instrument simulator to find out what specific type of interactor it is using. This sort of real-time type idenfification mechanism can be abused, but it can also be very useful. (A C++ dynamic_cast can provide the same information, but its use can lead to linking issues on some systems, and this seems simpler.)

Non-CANBus instruments

The sections so far have covered quite a lot of what is needed to write a simulator for a CANBus-based system. It's easy to package up support for CANBus interfaces into something like the GenInstSimulator class, because they are using a well-defined, general-purpose interface.

Working with non-standard interfaces means that none of this general-purpose CANBus code will serve, and more specific code will be needed. This is going to need code that works at a lower-level, and that is going to require a more detailed understanding of the way simulator tasks work in the AAO environment. Even if you are only working with CANBus interfaces, you may find this useful. Knowing how a system works internally makes it much easier to understand (sometimes even to guess) the best way to use it.

This is going to take us into the XWing sub-system, with its simulated device interfaces and its simulated drivers. What we'll find is that to handle a new type of device, you need to write a 'sim driver' for it, which will emulate read(), write() and possibly ioctl() and other calls to the actual device driver. (Writing real drivers is a notoriously specialised job. Don't worry, writing a sim driver is much, much simpler. For one thing, you don't actually have to worry about I/O operations.) You then need your 'sim driver' for the device to interface to the instrument simulator code, preferably in the same abstract way that a CANBus 'go-between' does. You'll probably find that you want a layer not unlike that provided by a Go-Between, but you do have a lot of flexibility - you control both sides of the interface to this layer, so can implement it just how you want.

First, though, we have to dive into XWing and the detailed simulator structure.

What is XWing and what does it do?

XWing is the sub-system that handles I/O redirection from real hardware to a simulator, and from the instrument control task end it looks a lot like a device driver. There is a quite detailed XWing manual, and you should read that for full details. This section tries to provide just enough detail to give you an idea of what's going on.

Forget CANBus for a moment, and consider some purpose-built hardware interface. This is the sort of thing used for the TCS project, where specialised interfaces handled I/O to the telescope motors and other bits of hardware such as the control panel for the telescope. These were implemented properly, with the instrument control code interfacing to them through device drivers (written at AAO). For HERMES, the vacuum gauges used were interfaced through RS232 serial lines using the standard drivers for such devices built into any operating system.

In all these cases, the key is that there is a device driver involved. You open it with a call to open(), you write to it with a call to write(), read from it with a call to read(), and so on. If you want to introduce simulation at as low a level as possible – as discussed right at the start of this document, in the introduction, you would simply provide alternative device drivers that interacted in some way with a simulator task instead of the real hardware, and the only change needed would be in the strings passed to the open() calls - you'd call open("/dev/sim_tty0") instead of open("dev/tty0").

That approach was seriously considered, but ultimately rejected. Elegant though it is, it requires writing drivers, and that gets complicated. Moreover, it's complicated in different ways for different operating systems, and we didn't want to get into that.

Instead, it's much simpler, and more portable, to introduce a thin wrapper for the drivers, and this is where the XWing subsystem comes in. Ideally, the wrapper will have the same set of driver-like calls (open(), close(), read(), write() etc.). In each case, the code for these calls simply tests to see if the device is being simulated. If not, it calls the equivalent real driver routine. If the device is being simulated, it calls the equivalent routine for an object of type XWingDeviceIface. Each simulated device has to have its own XWingDeviceIface object, but the standard XWingDeviceIface code handles all the standard device driver calls. There is no need to sub-class or inherit from XWingDeviceIface – you supply an identifying name for the device type in the constructor, and the standard code takes care of all the rest.

The XWing manual has a complete implementation of such a thin wrapper for a 'device' that looks like a file. The following is the whole of the code; it only supports the open(), close() and read() driver calls, in the interests of compactness, but write() would be equally straightforward.

```
using namespace std;
static const int FileDeviceType = 1;
class FileDeviceWrapper {
public:
   FileDeviceWrapper (void) : I_SimIface (FileDeviceType,"File device",0) {
      I_Simulating = false;
      I_Fd = -1;
   }
   ~FileDeviceWrapper () {}
   int Read (int Fd, void* Buffer, unsigned int MaxBytes) {
      if (I_Simulating) return I_SimIface.Read (I_Fd, Buffer, MaxBytes);
      else return read (I_Fd, Buffer, MaxBytes);
   }
   int Open (const std::string& DevName) {
      if (I_Simulating) I_Fd = I_SimIface.Open (DevName);
      else I_Fd = open (DevName.c_str(),O_RDWR,0777);
      return I_Fd;
   }
   void Close (void) {
      if (I_Simulating) I_SimIface.Close(I_Fd);
      else close (I_Fd);
   }
   void SetSimulation (bool Simulating) {
      I_Simulating = Simulating;
   }
   std::string GetErrorText (void) {
      if (I_Simulating) return I_SimIface.GetErrorText();
      else return XWingUtil::GetErrnoText();
   }
private:
   int I_Fd;
   bool I_Simulating;
   XWingDeviceIface I_SimIface;
};
```

(Calls to real drivers use open(), close(), etc with lower case names; a convention is that emulated versions of these calls, such as those for the wrappers or for the XWingDeviceIFace methods, start with upper case: Open(), Close() etc. Note that there is also a SetSimulation() call that allows simulation to be turned on, and a GetErrorText() call that returns a description of any error – when not simulating, XWing provides GetErrnoText() which returns the standard description of any errno error.)

What XWing actually does is turn each ‘driver’ call made to the XWinfDeviceIface code into a standard XWing message sent through a socket monitored by an XWing communications layer in the simulator task. The message includes details of the type of call that is being made (open, close, read, write, etc) the simulated device in question, and details of the call. An XWing communications layer in the instrument control task interacts through a series of messages with the XWing layer in the simulator task to complete the call and return any requested information and status to the caller. As you can imagine, this can get complicated when, for example, multiple threads in an instrument control task are performing overlapping blocking read() calls, but XWing handles all that.

At the other, end, in the simulator task, the XWing communications layer handles the messages it gets from the instrument control task. Note that there may be more than one instrument control task involved. One of the aims of this simulation scheme was to handle the case where multiple tasks are simultaneously working with the hardware. It may be that there is one task moving mechanisms, and another running an engineering interface that monitors the positions of these mechanisms. It’s important that the simulator provide a coherent picture of the instrument, so when one task moves a mechanism, another can see it move.

The code needed in the simulator task is a little more complicated that that in the instrument control task, which was deliberately designed to allow simulation to be introduced as simply as possible. However, XWing provides a lot of the necessary structure needed.

In the simulator task, for each simulated device, an instance of a class that inherits from XWingSimDriver needs to register with the XWing communications layer. This XWingSimDriver is what eventually receives the calls made to that device from the instrument control task, and needs to provide the standard set of driver-like calls (open(), read(), etc.). For example, this is what happens when a call is made in simulation to the Open() routine of the File Device Wrapper shown above:

1) The wrapper spots that simulation is active, so calls the Open() routine of the XWingSimIface being used for the device. This calls the XWing communications layer in the instrument control task with the details.
2) An XWing message containing all the relevant information - ‘this is an open call, for this device, with the following parameters’ - is written to the socket being used for communication with the simulator.
3) The XWing layer in the simulator reads this message, works out which device it is for, and that it is an open() call, and it calls the Open() routine for the XWingSimDriver registered for that device.
4) The XWingSimDriver in question handles the call, and returns the result to the XWing communications layer, which builds up an XWing reply message that it sends back to the XWing layer in the instrument control task. In this case, the only relevant information is the status of the operation.
5) The call made by the XWingSimIface to the XWing communications layer now returns with the relevant status, and the call to the wrapper can now complete.

The protocol required to handle blocking I/O calls is more complicated, but this should give the basic idea. Essentially, an Open() call to the wrapper in the instrument control task eventually ends up as a call to the Open() routine for the relevant XWingSimDriver in the simulator task, having been routed there via the XWingDeviceIface in the instrument control task.

(The naming of the various XWing components s a potential source of confusion. The XWingSimDriver lives in the simulator task. Since it’s called a ‘Sim driver’, you might think it’s what you’d talk to in the instrument control task instead of the real driver, but no - what you talk to in the instrument control task is the XWingDeviceIface. Eventually, your call does end up with

the ‘sim driver’, but this lives in the simulator task, and you’re only talking to it indirectly through the XWing communications layer.)

The description of stage 4) above, simply says that the XWingSimDriver ‘handles’ the call. How it does so is key to the structure of the simulator task.

Writing the code for the class that inherits from XWingSimDriver to handle the I/O requests in the simulator task is more complicated. The basic housekeeping is fairly straightforward. XWingSimDriver is a pure virtual class that defines the interface, but not an implementation. XWing provides a class that inherits from XWingSimDriver, called an XWingBasicSimDriver, and this provides most of the necessary structure for interacting with the XWing communications layer.

The example provided in the XWing documentation shows a possible implementation of the SimDriver for the “FileDevice” whose wrapper was shown above.

```
static const int FileDeviceType = 1;

class ReadSimDriver : public XWingBasicSimDriver {
public:
   ReadSimDriver(const string& DevName) : XWingBasicSimDriver(DevName)
   {
      I_InternalFd = -1;
   }
   int Open (const string& Path)
   {
      int Fd = -1;
      I_InternalFd = open (Path.c_str(),O_RDWR,0777);
      if (I_InternalFd < 0) {
         SetErrorText("Unable to open " + Path + ": " +
                                             XWingUtil::GetErrnoText());
      } else {
         Fd = OpenHousekeeping (Path);
      }
      return Fd;
   }
   int Read (XWingSimFILE& /*File*/, void* Buffer,
                                    unsigned int MaxBytes, bool* /*Blocked*/)
   {
      int Bytes = read(I_InternalFd,reinterpret_cast<char*>(Buffer),MaxBytes);
      if (Bytes < 0) SetErrorText (XWingUtil::GetErrnoText());
      return Bytes;
   }
   int Close (XWingSimFILE& File)
   {
      if (I_InternalFd >= 0) close(I_InternalFd);
      return CloseHousekeeping (File);
   }
private:
   int I_InternalFd;
};
```

A good place to look to see this in action is the code in the ACMM SimSerialDriver module, where the SimSerialDriver class inherits from XWingBasisSimDriver to provide something that looks very much like a driver for a serial line device such as an RS232 device. The code is quite short, and most of it is connected with supporting multiple devices, so the same driver can simultaneously handle “/dev/tty0”, “dev/tty1”, etc.

The problem is that the XWing housekeeping (although occasionally intricate, which is why XWingBasicSimDriver exists) is not the difficult part. Imagine a digital I/O device with a number

of I/O bits, where a Write() call sets bits that control the hardware and a Read() call reads other status bits that reflect the state of the hardware. When your 'sim driver' gets a Read() call, it gets the relevant bit values from the simulated hardware, and the structure provided by XWingBasicSimDriver makes it easy to send this data back to complete the original read() call made in the instrument control task.

The question is: how does the sim driver know what those bit values provided by the simulated hardware should be?

And the answer takes us into the question of the overall structure of a simulator task.

The structure of simulator tasks

The simulator task is usually written with a main thread that sets up the environment for the task - this could be a standalone-program, a DRAMA task, or a Qt-based program. The code is designed to be flexible and to accomodate all these - and possibly other - options. The main thread will start up a separate thread to run the actual instrument simulator, and for the moment, this description will concentrate on the structure of that thread, which is the same no matter what environment is used for the overall task.

If you didn't skip the previous section on XWing, you'll know that at the bottom of any simulator task is an XWing layer that handles the redirection of low-level I/O calls so they they end up being dealt with in the separate simulator task. The low-level XWing code deals with all simulated I/O operations initiated by instrument control tasks, and directs them to 'sim drivers' in the level above that handle the simulation of I/O calls for their specific devices.

The XWing layer, of course, knows nothing about the details of the hardware being simulated. This is specific to the instrument in question. There is a limit to what general-purpose code can do, and after that instrument-specific code has to take over. In the case of, for example, the purpose-built interfaces used for the AAT control system, the general purpose code can go as far as delivering a read request for a specific I/O bit to a 'sim driver' for the device being read, but the way that sim driver responds to that read requires that it know - or be able to find out - what these bits represent, and that depends on the detailed design of the instrument. These sim drivers have to work directly with code that understands the instrument in detail.

You can't just build that detailed knowledge of the instrument into the sim drivers themselves. A Write() call that sets a control bit on one interface might start some mechanism moving. A Read() call to a completely different interface might request the position of that mechanism. The Read() and Write() calls will probably be handled by different sim drivers. Clearly, there has to be a single layer of code, above the sim drivers, that represents the instrument as a whole.

In any case, it makes sense to structure the simulator task so that there is a clear demarcation between the details of how the I/O requests come through (this is the job of XWing and the sim drivers) and what they mean in physical terms for the instrument.

The details of the interaction between the sim drivers and the instrument simulation layer - really, the question here is - what is the API for the instrument simulation layer? - will vary from instrument to instrument. In general, it works best if this API is couched in terms that mean something as physical as possible. So, rather than a call that says 'bit 5 of such and such an I/O interface has been set', you have a call that says '"EmergencyStop" bit has been set.' It helps a lot to give each individual bit a meaningful name. This is best done through a configuration file - a text file that contains the information 'EmergencyStop is bit 5 of interface 3'. Importantly, this should be read by both the instrument control task and the simulator, and both should work as much as possible with named I/O items like 'EmergencyStop' - this makes the code clearer,

keeps both instrument control task and simulator in agreement, and makes it easy to re-allocate bits to different purposes when needed. Most AAO code uses a code layer called a ‘Configurator’ to handle this sort of configuration details, as described in the CANBus examples earlier.

With this structure, the sim drivers will make direct calls to the instrument simulator API to handle the I/O requests they receive. You could have a general API call that was BitWasSet(BitNumber,Interface), in which case the instrument simulator code would call the Configurator to discover that, say, BitWasSet(5,3) set the bit called EmergencyStop. Or you could let the sim driver do that configurator call and then call an instrument simulator routine BitWasSet(ItemName), in this case, BitWasSet(“EmergencyStop”). It’s up to the implementor - I’d prefer to have the instrument simulator API use named items as much as possible, myself.

But an important point here is that, when specialised interfaces like this are being simulated, the sim drivers end up working directly with the instrument simulator.

CANBus is an exception to this. CANBus uses an established, general-purpose, protocol called CANOpen. In a CANBus system, as shown in the CANBus examples earlier, the instrument control task code does not make direct I/O calls to the CANBus interfaces. Instead, it makes calls to a library such as CML, which has an interface that talks about devices such as servo amplifiers and I/O nodes. To control all these, each CANBus node, including the CANBus interface used by the instrument control task, is constantly sending 8-byte CANBus protocol messages around the CANBus, and reading those sent by other nodes. The actual low-level I/O, which really does little more than read and write these 8-byte messages, is handled in simulation by XWing, of course.

On the instrument control task side, the SimCanInterface object passed to CML to use for low-level I/O in simulation does indeed make use of an XWingDeviceIface object. In the simulator task, there is a CANBus-speciifc ‘sim driver’, inheriting from XWingSimDevice, that simulates the reading and writing of the 8-byte CANBus messages by the simulated CANBus nodes.

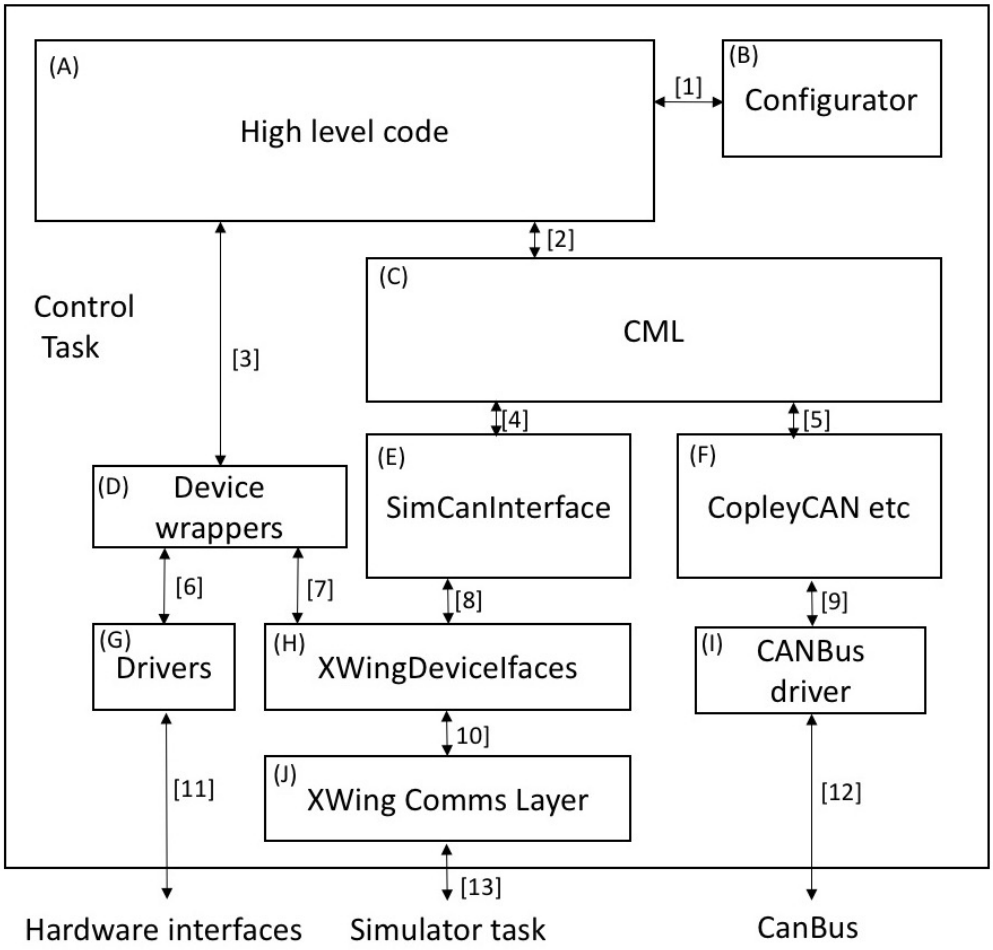
On top of the CANBus-specific ‘sim driver’ is another general-purpose layer that handles the bulk of the simulation of the CANBus devices. This layer consists of a set of objects that simulate either CANBus amplifiers or CANBus I/O modules. (They are all instances either of IOModuleSim or AmpSim, both of which inherit from a general NodeSim class.)

When the simulator starts up, it can be passed details of the CANBus configuration - how many nodes there are, which are servo amplifiers and which are general purpose I/O nodes, how many I/O bits each I/O node supports, etc. It sets up the set of node simulators. The CANBus ‘sim driver’ will then handle the low level I/O simulation, passing on the CANBus messages it receives to all the nodes, and allowing them to send their responses back to the simulated CANBus. The node simulators understand the CANOpen protocol used by the CANBus messages and can handle a lot of what each message requires. An amplifier simulator can handle a surprising amount of what is required – in most cases it can simulate an entire move all by itself, merely needing to report what is happening to the higher level instrument simulator layer.

Obviously, there are limits to what the simulated nodes can do. An I/O node can work out that bit 5 of its second digital input bank has been set, but it doesn’t know what to do with that. The best it can do is use the configurator to find out that this is the bit called, say, EmergencyStop and then call the simulator layer routine that says a bit was set, using that name. And at that point, we’re pretty much where we were with the previous, specialised interface, example, except that the CANBus emulation layer with its simulated nodes sits between the sim driver and the instrument simulator itself.

Task structure diagrams

The diagrams show the structure of an instrument control task and the corresponding simulator task. The description given here mostly repeats the description given above, but the overall view provided by the diagram should help make things clearer.



The instrument control task diagram shows the following components, labelled using letters:

A) The high level instrument control code. This needs to be specific to the task and the instrument, and does whatever is required to control some or all of the functions of the instrument. For CANBus hardware, this will use the CML library. Purpose-built devices need purpose-built control code, and this control code would normally talk to device drivers directly. To support simulation, it goes through device driver wrappers.

B) The Configurator. This provides details of the hardware layout of the instrument, which it usually reads from a configuration file. This same code, and configuration file, is used by the simulator task, ensuring both have the same view of the instrument and use a common naming convention for its items. This code is usually based on that in the ACMM module AAOCANBusIniConfigurator.

C) The Copley Motion Library (CML). This is a proprietary library that provides ways of controlling CANBus modules, including servo amplifiers and general purpose I/O modules.

D) The device wrappers, one for each device driver, provide a way of directing what would be standard driver calls (open(), read(0, etc) either to the actual hardware drivers or to the simulator task. These need to be provided for each device in use, but the code for each is usually trivial.

E) The SimCanInterface code conforms to the requirements for an interface between CML and the hardware interface to a CANBus, but instead direct communications to the simulator task.

F) CopleyCAN etc, represents the set of standard interfaces between CML and the various available hardware CANBus interfaces. CopletCAN interfaces to a Copley-supplied interface, alternative modules interface to other hardware interfaces or to a CANBus-over-internet interface.

G) Drivers represents the set of operating system drivers provided for interfacing to real hardware. These may be standard drivers, such as that for RS232 interfaces, or may be written in-house for purpose-built interfaces.

H) The XWingDeviceIfaces - different instances of the XWingDeviceIface class, one for each device type being simulated - look like device drivers for their devices (they provide the standard set of driver calls, open(), read() etc,) but work by redirecting the calls through the XWing communications layer to the simulator task, where they will be handled by a corresponding set of XWingSimDriver objects.

I) The CANBus driver is the standard operating system driver for the CANBus interface in use, usually provided by the vendor of the interface in question.

J) The XWing communications layer handles communications over sockets between the XWingDeviceIfaces in the instrument control task and the corresponding XWingSimDrivers in the simulator task. Although this is conceptually a separate layer, and is shown as such here, the boundary between this and the XWingDeviceIface code is actually somewhat blurred.

The interfaces between the various code modules in an instrument control task are labelled by numbers:

1) The high level instrument control code uses the Configurator API to enquire about the configuration of the hardware in use. Generally, all individual I/O bits, servo amplifiers, analog inputs and outputs, etc., are given symbolic names, and, given such a name, the configurator will provide the hardware details required to access it. For a named CANBus amplifier, for example, this will be mainly the CANBus node number for the amplifier, but the configurator may also provide additional details such as scaling and limits for the device. Ideally, as little information as possible about any hardware should be hard-coded into the instrument control task.

2) To control devices on a CANBus, the high level code makes use of the CML library. The CML API is described in detail in the Copley-supplied documentation. (Being Doxygen-generated, the documentation is very good in the details of the individual routines, but not quite so good at providing an overall view of how best to use the library.)

3) For non-CANBus devices, the high level code would normally work at the device driver level. (In practice, the high level code would normally be itself structured so that it has lower levels that make device drive calls and higher levels that work at a more abstract level. To support simulation, however, it makes calls via the device wrappers instead. This communication is all in terms of calls to the standard driver calls (or their wrapper equivalents), such as open(), read() etc.

4) When CML is initialised, it is passed a reference to an object it can use to write and read CANBus messages to and from the hardware, using an interface defined by the CML CanInterface class. If simulation is not being performed, these will be the standard objects like CopleyCAN etc. If it is being performed, it will be passed a reference to a SimCanInterface, which conforms to the CanInterface requirements (it inherits from CanInterface), but which directs calls to the simulator task via the XWing system.

5) Just as for 4), communication follows that defined by the CanInterface class, but in this case it will be ultimately directed to the actual CANBus hardware.

6) If not running in simulation, the device wrappers direct their calls to the actual hardware drivers, using the usual driver set of calls.

7) If running in simulation, the device wrappers direct theri calls to the XWingDeviceIface for the device in question. The interface looks like that for an ordinary driver, except that the routine names have leading capitals, eg Open() instead of open(). Note that in some cases a device wrapper can be more imaginative in the way it uses the XWingDeviceIface objects, if this makes things simpler. Some devices, for ezample, find it easier to use two XWingDeviceIface objects for one driver, one for reading and one for writing. The XWing manual discusses this in more detail.

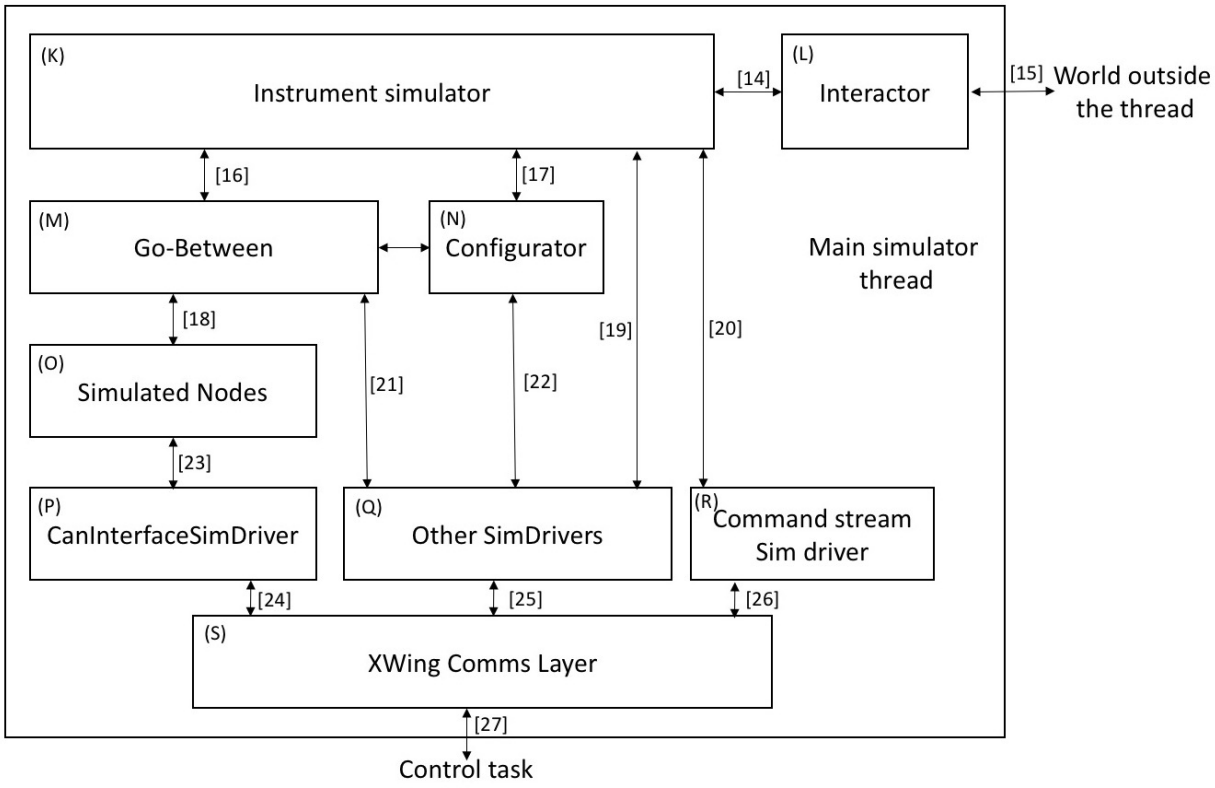
8) The SimCanInterface sends CANBus messages to the XWing comms layer through the set of XWingDeviceIfaces. (It is one case where two such interfaces are used where one might expect only one to be needed, but this is completely transparent to the rest of the system.)

9) The standard CopleyCAN interfaces make use of the operating system drivers for the hardware device in question, usually that supplied by the hardware vendor.

10) The XWingDeviceIfaces implement a lot of the conceptual XWing comms layer functionality themselves. It’s probably cheating to show this as a defined interface.

11) Real drivers talk to real hardware in the way drivers do. Fortunately, we don’t need to know anything about the details of how they do it.

12) See 11). Doubly so for CANBus.



The simulator task has one thread that runs the main simulation code, and it has the following main components, indicated by letters:

K) The instrument simulator itself. This is the block of code that maintains a simulation of the state of the instrument, responds to changes as the result of commands from an instrument control task or externally, and can respond to enquiries as to the current state of the instrument. The interface to this should be as abstract as possible; this level should not need to know about the bit layout of interfaces. Its API should be at the level of named items, such as ‘EmergencyStop’, ‘ShutterClosed”, “GratingServoAmp”.

L) The Interactor abstracts interaction between the instrument simulator and the outside world. Usually, this is connected with information going from the simulator to the outside world, rather than the other way around - because of issues to do with thread context, the CommandStream mechanism should be used for that - see R). The interactor would normally be used to update whatever mechanism is used to communicate with a GUI (DRAMA parameters, for example), output messages, etc. It has an abstract interface that makes no assumptions about the outside world. The implementation of the interactor will depend on the nature of the outside world.

M) The Go-between is what allows the simulated CANBus nodes to work in hardware-based terms such as ‘bit 5, digital output group 2, on CANBus node 3’ and the instrument simulator to work in more abstract concepts like ‘PowerOn - the bit that controls power to the instrument’. It works with the configurator to translate between these two ways of looking at the instrument hardware. It also provides a standard API the CANBus nodes can use, which it converts to calls to the instrument-specific API of the instrument simulator.

N) The Configurator is the module that knows the details of the instrument hardware configuration. It reads this from a configuration file, usually using the .ini format. A fairly standard format has emerged for configuration files for CANBus nodes, and this includes information about servo amplifier parameters. It is also possible for a configurator to provide similar services for more specific hardware - this would depend on the instrument. A complication comes from

the use of multiple CANBuses, and has led to the use of a ‘Network configurator’ which controls a set of individual CANBus configurators, one for each CANBus. The ‘Configurator’ block in the diagram represents the overall set of configurators and the network configurator, if one is used.

O) The simulated CANBus nodes. This is a set of Amplifier simulators and I/O module simulators provided by the ACMM module SimCanOpen. The CanInterfaceSimDriver (P) directs messages for individual nodes to these. At startup, these are created on the basis of the configuration details supplied by the Configurator (N), and their code is instrument-independent. Instrument-specific aspects are handled by the Go-Between (M).

P) The CanInterfaceSimDriver is a standard module provided by the ACMM module SimCanOpen. It emulates the overall CANBus network, routine messages between the individual simulated nodes (O) and the XWing comms layer (S). Note that the CANInterface sim driver is just an XWing sim driver like any other at this level of the diagram; it’s shown separately because CANBus devices are common to so many instruments and a standard way of handling them, using the simulated nodes and the go-between, has emerged and it makes sense to show that explicitly in the diagram.

Q) This covers any sim drivers for devices other than CANBus devices and the ‘phoney’ command stream sim driver (R). This includes drivers for serial interfaces like those used for vacuum gauges, and drivers for other purpose-built interfaces. They can work in any way that makes sense to the simulator designer - they can use a configurator directly, or via a go-between, or can even talk directly to the instrument simulator.

R) The command stream Sim driver emulates a purely conceptual device - an XWing Command Stream device - that exists purely to allow messages to be sent to the simulator task from some external entity. A typical example of such an entity would be a simulator GUI that needs to send messages to the instrument simulator. Sending these messages through the underlying infrastructure rather than calling instrument simulator routines directly means these are handled properly, in the context of the simulator thread, which can be important.

S) The XWingComms layer talks to the corresponding XWing comms layers in the instrument control task(s), and routes messages between these and the various sim drivers (P,Q,R).

The interfaces between the various code modules in the simulator are labelled by numbers:

14) The interactor provides an API for the simulator to use that makes no assumptions about the nature of the outside world. Typical calls are LogError(), LogMessage() etc. Note that the ‘outside world’ just means whatever is external to the simulator thread - it may be a DRAMA task, a Qt task, a standalone application, or something else entirely. No existing interactors make calls back to the simulator code. (This could be allowed, but there might be issues of thread context involved.)

15) The interactor is aware of the nature of the outside world, and communicates with it appropriately. LogMessage() may be handled by a simple printf(), or it might be handled through a call to a GUI routine or a DRAMA call, depending on just what’s out there. Usually, this interface is only used to send information to the outside world. The best way to get information into the simulator from outside is to use an XWing command stream device.

16) The interface between the instrument simulator and the go-between will depend entirely on the needs of the simulator. Both ends can be tailored as required. Nonetheless, the requirements of most CANBus based instruments are very similar at this level, and the interface for an existing instrument can probably be adapted easily for a new one. This interface is definitely two-way; the simulator tells the go-between when things have changed, so this can be reported via the CANBus; the CANBus nodes tell the simulator when changes have occurred in the hardware

interfaces, usually as a result of an instrument control task writing to the interfaces. Note that a simulated amplifier will handle most of the details of a movement itself, reporting progress to the simulator through the go-between.

17) If the configurator is just used for CANBus interfaces - and this is usually the case - the existing CANBus configurator code and the network configurator code from the ACMM module AAOCANBusIniConfigurator can be used and this interface. This existing interface consists entirely of enquiries that can be sent to the configurator and the resulting replies.

18) There is a standard API used between the simulated nodes and the go-between, allowing the nodes to tell the go-between that the instrument control task has just done something, such as set an output bit or started an amplifier moving, and allowing the simulator to tell the nodes, via the go-between, that other things have happened as a result - an interlock may have set, which needs to be reflected in the value of an input bit. (Remember, an 'input' bit is something the instrument control task reads, meaning it is something set by the simulator.)

19) Other instrument-specific sim drivers may need instrument-specific ways of calling instrument simulator routines, and vice versa.

20) The command stream sim driver only does one thing. It receives messages from the outside world, and passes these to the instrument simulator. It provides a SetWriteCallback() routine that allows the instrument simulator to specify the routine it should call when it receives such a message.

21) Other Sim drivers can in principle use a go-between for their own purposes. In this case whatever calls they need can be added to the go-between, and the sim drivers.

22) If an other sim driver wants to use a configurator, a specialised configurator with the necessary support will be needed, and the details of the interaction will be instrument-specific. (There does not have to be one configurator just because there is one block here. There can be a configurator specific to one interface, if that allows code to abstracted in a useful way.)

23) The interface between the simulated nodes and the CanInterfaceSimDriver is an internal matter for the SimCanOpen code. (In practice, most CANBus messages are sent to all nodes - which is what happens in the real hardware - and the nodes decide if they are interested in them.)

24) The interface between the XWing sim drivers and the communications layer is in terms of what resemble standard driver calls - Open(), Read(), etc. This is the same for all the sim drivers.

25) See 24.

26) See 24.

27) The XWingComms layer uses sockets to transmit XWing messages between the simulator task and the instrument control task(s). The protocol is defined by the XWing system.

Support for specialised interfaces

So how should one structure a simulator that handles a specialised interface? There are no hard and fast rules, of course, since almost everything in a simulator is up to the implementor. The XWing specification defines what has to happen at the very lowest layers. The XWing comms layer should really be left as it is, unless it turns out that some new feature is essential. (The XWing documentation acknowledges that there are limitations with the existing implementation,

but also shows how most of these can be worked around. Read that carefully before considering any changes to XWing itself.)

The XWing specification also defines the interface between the XWing comms layer and the Sim driver for whatever new devices are being simulated. However, everything from there up can be implemented in whatever way suits the simulator author. If there is only one device involved – it would be a simple instrument, but they do exist – then all the simulation might possibly be handled in the sim driver itself. However, once multiple devices are involved, there really needs to be a common piece of code that simulates the overall instrument, and ideally the API for that code should be at a more abstract level than one that talks about specific bits on specific interfaces.

Bear in mind that this is not a new problem, and the way exisiting simulators have solved it has tended to involve the use of Configurators that read configuration files, and Go_Betweens that act as intermediaries between the sim drivers and the instrument simulators.

A useful example may be the way the vacuum gauges were handled by HERMES. Most of the general-purpose code here can be found in the ACMM module SimSerialLine. This is really another example of a general purpose interface, except that this case it is a serial line interface instead of a CANBus interface. On the control task side, the control tasks read and write to RS232 interfaces (devices call /dev/tty0 etc.). On the simulator side, any serial device can be handled by a general purpose serial line sim driver: a SerialSimDriver, which inherits, as it must, from XWingSimDriver.

A SerialSimDriver merely keeps track of a number of objects that handle specific serial devices, all of which have to register with it an object that implements an interface defined by the SerialDeviceSim class. Both of these are defined in SerialSimDriver.h. When an I/O request comes in for a serial line device, the sim driver routes it to whichever SerialDeviceSim object has registered to handle that device.

The code in VacuumGaugeSim.h and VacuumGaugeSim.cpp implement the SerialDeviceSim interface (VacuumGaugeSim inherits from SerialDeviceSim), and provide simulation of the Pfeiffer TPG261/262 gauges used by HERMES. They make use of a VacSimGoBetween class, defined in VacSimGoBetween.h. The original intention was that this should make use of a configurator, but in practice the interface that emerged was too simple to require one.

The only thing a vacuum gauge does is provide a vacuum reading. A VacuumGaugeSim can handle all details of interaction with the control task, the way the device uses the serial line etc, but the one thing it doesn't know is that actual vacuum reading. In other words, the whole interaction between the VacuumGaugeSim and the instrument simulator – the interaction handled by the Go-Between – consists entirely of the instrument simulator responding to the question: 'what is the vacuum gauge reading for this specific device?'

So the VacSimGoBetween class only requires a constructor, a destructor, and this one interface routine. Here is the actual code in VacSimGoBetween.h, with comments removed:

```
class VacSimGoBetween {
public:
   VacSimGoBetween (void) {}
   virtual ~VacSimGoBetween() {}
   virtual bool GetVacuumReading (
        const std::string& /*Device*/, int* Status, double* Reading) {
      *Status = 0; *Reading = 0.0; return true;
   }
   virtual void Update (void) {}
};
```

(The Update() routine was presumably added because it was thought it might be useful to allow a vacuum gauge to force an update of the simulator model, but in practice it isn't used.)

It's hard to suggest designs for situiations that don't yet exist, but this may have helped to show how some devices can be handled.

Existing simulators

No document of readable length can possibly cover all aspects of how to write an instrument simulator, especially not one for a new instrument. There is, however, a large body of additional help to be got from looking at the code for existing simuators. This document has tried to give enough of a description of the underlying concepts to make it clear what the code in existing simulators is doing. However, you need to understand that AAO simulators have evolved over the years, since these concepts were first introduced for the reworking of the AAT Telescope Control System (TCS). The Generic Instrument Simulation code described in this document has, so far, only been used for the 2dF refurbishment project, and at the time of writing (June 2018), this has only really exercised a small subset of what a simulator can be asked to do,

TCS

The TCS was the first AAO project where this level of simulation was used, and it was very successful. The hardware interfaces to the TCS provided analog I/O and digital I/O interfaces, and a separate somewhat unusual synchronous serial interface to the telescope axis encoders. The Linux drivers for these devices were written by Lew Waller, at AAO. There were driver wrapper routines used to switch between calls to the real drivers and XWing simulated interfaces, just as suggested in this document).

No configuration files were used. Bit allocations were embedded in .h files, like TcsDigitalInput.h in the ACMM module TcsHardware, but these were used by both the instrument control code routines and by the simulator code, so both were kept in sync that way. This led to some awkward pieces of code, particularly those that returned the symbolic name used for a given I/O item. This code was often generated auromatically by programs that parsed the .h files – this led directly to the idea of a configuration file, although too late for TCS.

No go-betweens were used – the XWingSimDevice implementations in the lower levels of the simulator task were sufficiently specialised that they knew enough to be able to call the correct instrument simulator routine directly when something changed.

The API presented by the simulator was abstract in the sense that it knew nothing about bit allocations in I/O devices. It provided routines like SetOutputHAMotorVoltage(), called directly from the analog input sim driver when it read a new value that it knew represented the HA motor voltage.

TCS introduced the concept of 'partial' simulation, where some parts of the system are simulated and some are not. This was handled at the device wrapper level, driven by the setting of 'TCS variables' read from a startup file, which could be used to control various aspects of TCS operation, including which parts of the system were simulated.

The TCS simulator needed a GUI to represent the AAT hardware control panel, The simulator task ran as a DRAMA task, with the simulator in one thread, and a fairly simple DRAMA task running in the main thread, mainly to interface with the GUI. The GUI – a DRAMA program using Tcl/Tk - could send commands to the DRAMA task, for example when a button was pressed on the panel - which would then call routines in the simulator thread directly. These were called in

the context of the DRAMA thread, which meant some care was needed. The instrument simulator code could set DRAMA parameters that would be picked up by the GUI, but this had to be done indirectly using the DRAMA signal mechanism, which would trigger a monitor action built into the DRAMA task for this purpose.

WFMOS

There was never a full simulator developed for WFMOS, but some simulation work was done for WFMOS. This used more or less the same structure as for TCS, although there were some changes made to XWing, which had been a TCS-specific system, but now diverged into TcsXWing and WfXWing, with WfXWing gaining WFMOS-specific classes such as WfXWingSpineIOIface, clearly something that was really not general-purpose code.

The most interesting thing about simulation for WFMOS was that it simulated not only the movement of the spines in the fibre positioner, but also the operation of the firewire-based metrology camera, the two being linked to provide simulated camera images that represented accurately the simulated positions of the fibres. Structurally, WFMOS introduced little new into the simulation, other than the realisation that XWing really was a general-purpose piece of software and needed to be reworked as such.

The WFMOS simulation code probably doesn't have much to offer – it was specialised, and represented something of a false track for sub-systems like XWing.

HERMES

HERMES marked a big step forward for simulation for AAO instruments. By now it was clear that this sort of detailed instrument simulation was really useful, and it followed from that that some effort should be put into making general interfaces that were instrument-independent. At the same time, it was the first AAO instrument to use CANBus, and a big effort went into writing a CANBus simulator that could handle the whole of the CANOpen protocol. (Well, actually, that subset of CANOpen protocol used by the CML library for the purposes of HERMES, but that covered most amplifier and IOmodule operations.) This became the ACMM SimCanOpen module.

XWing finally became an instrument-independent ACMM module, just 'XWing'. An XWing manual was written, and in the process of writing it, it became obvious how unnecessarily complex some of the instrument control code had become. Previously, each device had to provide its own specialised simulated device interface, inheriting from XWingDeviceIface. By modifying the Open() call for the device, it proved possible to make the base XWingDeviceIface code capable of handling any device. Similarly, documenting the complicated process of writing the Sim Driver for each device supported by the simulator task proved more difficult than restructuring the code so it was easier to use and describe. HERMES sim drivers were based around the resulting XWingBasicSimDriver, which now handled most of the standard housekeeping needed.

HERMES introduced the use of a Configurator and a .ini file to describe the configuration of the instrument. There was – and remains – some lack of clarity about how simulation should be handled at the instrument control task level. This applies mostly to partial simulation, which presents some problems with CANBus devices, since a CANBus system operates as an entity and it is difficult to simulate some parts and not others. HERMES used two CML CanOpen systems, running together, one simulated and one not, and used entries in the configuration file to control which of these was used by the wrapper routines that accessed the hardware. This was effective, but made the simulation mechanics visible at a higher level in the instrument control code that one would have liked.

With CANBus handled by so much general-purpose code, the concept of a Go-Between was introduced, mainly to fit between the CANBus node simulators and the instrument simulation code. The HERMES Go-between handled both the CANBus devices and the serial line vacuum gauges, which were the only non-CANBus devices used by HERMES.

The HERMES simulator task ran with a main DRAMA thread. This controlled a simulator GUI, used to show the internal state of the simulated instrument. The instrument simulator was able to signal the DRAMA thread that it had parameters to be updated, and the DRAMA thread would then make a call to a parameter update callback routine in the instrument simulator. This allowed the instrument simulator to have code that manipulated DRAMA parameters directly, but since it was only called from the DRAMA thread, this was run in the correct DRAMA context. This worked well in the fixed case where the simulator knows it is working with a DRAMA task, but is hard to generalise to the case where some other form of GUI is used (e.g. one using Qt).

The HERMES code represents a comprehensive repository of examples of how to use the AAO simulation structure for a CANBus instrument. HERMES was comprehensively simulated, with a proper simulator GUI. However, the code was specifically designed to work with a DRAMA main program.

TAIPAN

Some simulation code was written for TAIPAN. This is complicated by a number of things, one being that in this context it isn't really clear what 'TAIPAN the instrument' really is. The fact is that there was no proper simulation of the complete TAIPAN instrument, if by that we mean the Starbug system, the spectrograph, the system that moves the field plate up and down, etc. There were in fact three quite separate TAIPAN simulators:

o A simulator for the starbug controller and the metrology camera. (TaipanSimulator, in ACMM)
o A simulator for the CANBus-based part of the instrument. (TaipanCanSimulator, in ACMM)
o A simulator for the spectrograph and calibration lamp. (TaipanSpectroSimulator, in ACMM)

The starbug simulator, TaipanSimulator, doesn't follow the structure described in here at all. In fact, it isn't even an XWing-based system. This is because the StarbugController is a separate piece of hardware that itself presents an interface to an instrument control task, and itself sends commands through a socket to the actual Starbugs. It was not feasible to use the usual structure here. Instead, this code simulates the Starbugs, reading commands from the socket. The upper levels of this task are similar to those described in this document, but the lower levels are quite different.

The other two simulators, TaipanCanSimulator and TaipanSpectroSimulator do simulate CANBus-based instruments. The Starbug controller is a one-off design, but the rest of TAIPAN is CANBus-based. There is some overlap between the two simulators, and they were written by different people. However, both are essentially copy-and-paste-and-modify versions of the HERMES code, and probably isn't much to learn from them. TAIPAN uses a Go-Between based on that for HERMES, and it also uses a Configurator and a .ini file. Actually, TAIPAN uses multiple Configurators and multiple .ini files.

The most interesting new aspect of the TAIPAN CANBus simulation code is that TAIPAN has more than one CANBus, and so this was the first time multiple CANBuses needed to be supported. This led to the introduction of the 'network configurator', now included with the original configurator code in the ACMM module AAOCANBusIniConfigurator. The network configurator provides an interface that resembles that of the individual configurators but which works with more than one CANBus, each with its own .ini configuration file. (This does leave open the

question of how to deal with non-CANBus items – should they be in a separate configuration file, or can they just be included in any one of the CANBus ,ini files?)

In the low-level CANBus simulation code (in the ACMM SimCanOpen module), the SimCanInterface class – as used in the code described in this document – was introduced to replace the SimCopleyCAN and SimAnagateCAN CANBus device interfaces used until now. This has the advantage of being more general an interface, and able to handle multiple CANBuses. It has the disadvantage that it always operates in simulation, and no longer has the SetSimulation() call provided by the earlier classes - since it isn't tied to any one CANBus interface - and this means the higher-level code is marginally messier.

GHOST

A fairly straightforward simulator was written for GHOST. This emulated the basic servo amplifier mechanisms used by the instrument, but had no GUI or more elaborate simulation of the instrument as a whole. The code in GhostSimulator was another copy-and-paste-and-modify version of the HERMES code, and probably has little to teach anyone, although just by being much simpler than the HERMES code it may be more approachable. GHOST uses a Go-Between based on that for HERMES, and it also uses a network configurator and ,ini files based on those used for TAIPAN.

The most interesting aspect of the GHOST simulation was that this was another multiple CANBus system, but this time the two CANBuses in use worked with almost completely separate parts of the instrument. This meant that there were good arguments for using two separate simulator tasks, one for each of the two parts of the instrument. This had some technical implications, as until now XWing device interfaces (in the low levels of the control task) had always just had one fixed simulator task to talk to. This led to the introduction of the concept of the named XWing server. The constructor for a SimCanInterface was extended to take an optional server name (you can see this being used in the Ex1Read example code in this document), and this could be used to direct communications to one of multiple possible simulation tasks, each supporting to a differently-named XWing server.

2dF

For the 2dF refurbishment project, which involves replacing the old interface electronics with CANBus interfaces, a new simulator was written. (The ACMM module 2dFCanSimulator.) This was originally going to be yet another copy-and-paste-and-modify version of the HERMES code, but it was realised that most of the housekeeping provided by this code was really instrument-independent, and the code could be refactored into an instrument-independent layer (which became the ACMM GenInstSimulator module) and a thin instrument-dependent module (in this case 2dFCanSimulator). This structure is what is described in this document, and the 2dF simulator code provides the best examples of how this was intended to be used. Unfortunately, at the time of writing, June 2018, the 2dF project – and the simulator – is still incomplete, so this at best provides an incomplete set of examples.

To see how something not yet supported in GenInstSimulator might be implemented, the best this to look at the equivalent HERMES code. In most cases, all that is required is to add the HERMES code for WhateverItIs() to GenInstSimulator, with the HERMES-specific parts abstracted into a call to a private method called InstWhateverItIs(). This is essentially how the GenInstSimulator code was produced.

As an experiment, the GUI used for 2dF was not DRAMA-based, but used a main thread running Qt to provide a GUI that included a 3D representation of the 2dF positioner written using OpenGL. The instrument-specific code for this is in 2dFCanSimulator, but, as for the simulator

code, most of the Qt-based housekeeping is provided by routines included in GenInstSimulator. GenInstSimulator does not mandate any specific form of environment – DRAMA, Qt, command line, etc. However, it provides general-purpose interfaces that can be used for any of these. GenGuiInteractor and the CommandStream support, described earlier, provide a way for the instrument simulator code to communicate with the outside environment in an environment-agnostic way. It also provides some general-purpose code (GenGuiGLWidget, and GenGuiArtist, for example) that are Qt-specific. In time, DRAMA equivalents could be added.