Background

Below is described the architecture and associated problems for a system used to run a photo-acoustic spectrometer (PAS). The PAS consists of a five cells. Ambient air is drawn through each cell (the rate of which is controlled by five individual mass flow controllers) and particles in the flow are irradiated by a laser which is operated at a specific wavelength and a unique frequency. Particles absorbing light at that wavelength reemit the absorbed energy as heat given off at the frequency of the laser. This results in audible pressure waves which can then be detected using a microphone. The signal is amplified by operating the laser at the natural resonant frequency of the cell. This frequency is determined by periodically generating a chirp over a wide bandwidth using a speaker located in the cell.

The system whole system is maintained at a single temperature using a thermoelectric cooler which circulates cool water throughout the base plate. Each cell's temperature can be controlled individually using thermoelectric coolers mounted on the base of the cells. All housekeeping variables (temperatures, flow rates, humidities, etc) are logged along with the detected laser (captured using a photodiode at end of the cell opposing the laser) and the microphone signal.

This system has been in continuous development since the spring of 2009 and operated on an aircraft platform in the spring/summer of 2010. Although the system was initially developed to operate independent of any other instrument, the instrument operation became inextricably interweaved with another instrument (called the CRDS) thus requiring some form of aggregating communication under a single interface. Thus, both systems were set up in a command architecture with both instruments running a server on a the Windows XP side of the controller and communicating via TCP/IP (using the STM library) with a single host on a Windows 7 machine.

Due to unplanned changes that resulted in this setup (as well as my own limited experience with RT), we were unable to take advantage of the RT aspect of the controllers. This resulted in non-ideal operation as it required users to switch between the host and the server systems using a remote desktop tool for such operations as proper shutdown. This, coupled with the desire to combine the two instruments into a single entity with a single chassis, motivated the desire to update the architecture, the ultimate goal being an autonomous instrument that can be operated remotely on an aircraft. All changes (and their subsequent problems) apply to not only the software associated with this instrument described above but also with that related instrument, the CRDS.

Current Configuration

Currently, I am redeveloping the software using Labview such that it can operate autonomously on an RT machine. The RT machine is located in an 8-slot, standard PXI chassis (1042Q) and consists of the following components:

- An 8110 controller
- Two 6143 S series cards for phase coherent acquisition of 10 signals
- A 6229 M series card for acquisition of approximately 25 low frequency, housekeeping signals

- A 7841R R series multifunction RIO for generation of 5 analog out signals at five distinct frequencies
- A 8430/2 module for acquisition of RS-232 data from two devices
- A 6733 AO series card for generation of 5 AO signals

The system is designed to run autonomously (i.e. it will run on power up with no user interaction). The main VI utilizes a queued state machine to control the current state of the machine while a separate loop polls shared variables (used for communication with a host possibly operating on a Windows machine) for changes in outputs.

Architecture

All of this is fairly standard, but now we get to the part where my architecture may deviate from more standard architectures. First, all data and actions pertinent to operation of the machine are rolled into an overarching class (called *PAS* in Figure 1). This class is further composed of (or possibly arrays of) other classes:

- The *Serial IO* class which methods and attributes associated with serial communication. At configuration time, the different objects in the array are typed according to what is in the config file:
    - The *TEC* class is used to communicate with a thermoelectric cooler
    - The *AlicatCommBank* class is composed of an array of Alicat objects and is used to communicate with a bank of addressed controllers on a particular port. The Alicat objects are further typed according to what the device is. All of the Alicat devices in this application are mass flow controllers so they are children of the *Controller* class (MC) which is in turn a child of the *Alicat* class.
- An array of PAS cell objects which contain attributes specific to the operational cells (up to 5)
- A *PASData* class which encapsulates all of the data relevant to the PAS as well as methods to properly extract the data. This data is flattened to string in the main VI and passed to any listening devices using a shared variable.

This last class is further used by all *Serial IO* objects to encapsulate the data associated with these objects using the appropriate child of the superclass *Serial IO*. Figure 1 lays out the class diagram describing the architecture of the system as originally intended.

In the main VI, the *PAS* object that is instantiated is placed in a data value reference (DVR). This is routed to both the state loop and the loop listening for communication from a host. The DVR exposes the PAS object to the interface loop for limited interaction (i.e. changes to control value). Use of the DVR allows information to be properly updated in both loops while preventing communication clashes. Because actions performed in the interface loop occur over a very limited span, risk of overruns in the state loop are minimal.

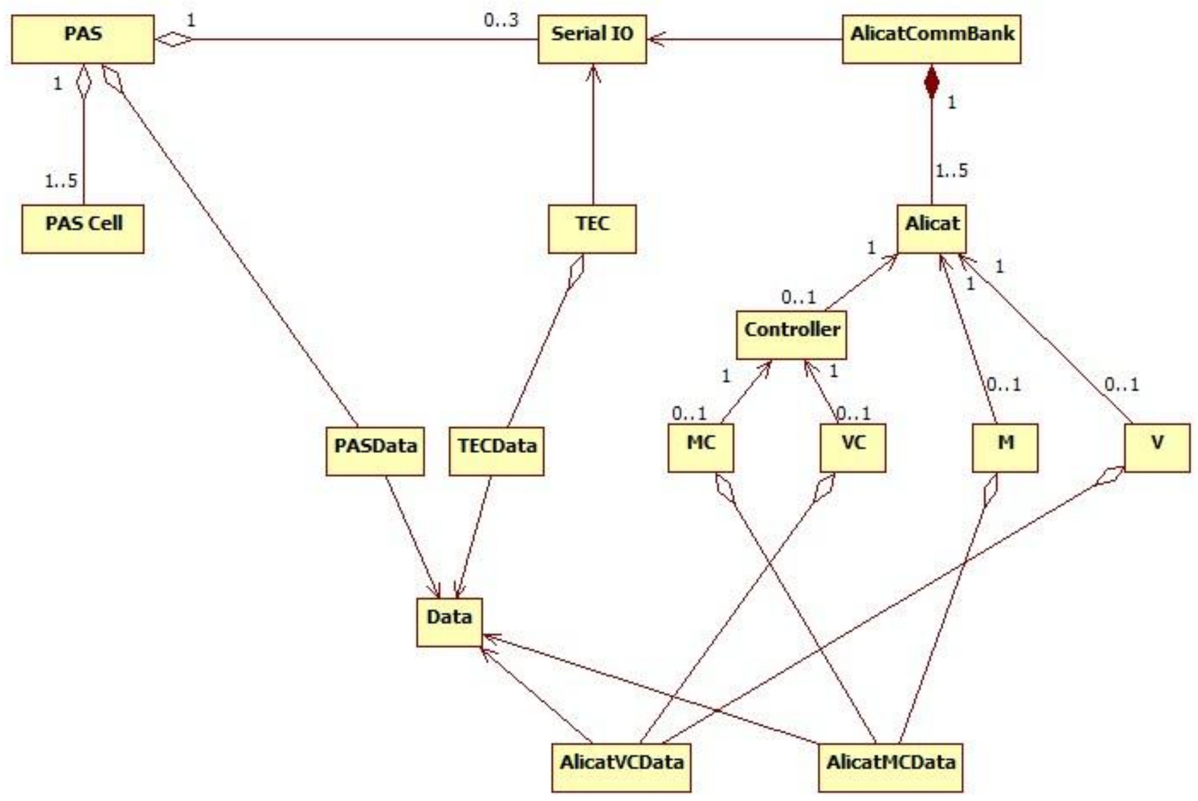

Figure 1. Class diagram for initial architecture. While the top level VI could be run in the development environment on the target, executables built on this architecture caused fatal exceptions forcing the system to boot into safe mode.

Problem

The architecture described above was originally developed and debugged using the Labview development environment and running on the target. Once the system was running reasonably well, I built an executable for startup on the target. Although the results were not always consistent, the system never ran as expected. The most common result was that a system exception would be thrown at startup which ultimately resulted in reboot into safe mode. The addition of a wait at the beginning of execution had no noticeable impact, not even in a delay of the reboot.

Further debugging indicated that the problem was related to the instantiation of the *Serial IO* objects. And, by drilling down, I found that the problem lay with the *AlicatCommBank*. In the PAS, the type of Alicat device is a mass flow controller (*MC* which is a child of *Controller* which in turn is a child of *Alicat* and is used by *AlicatCommBank*). When an executable was built for just the *Serial IO* objects, the system did not crash. However, data was never properly returned by the mass flow controllers indicating that the software was having problems correctly typing the object.

Solution

I was able to get a functional serial IO executable by providing a specific class for the PAS Alicat controllers that reduced a level of abstraction (i.e. it inherited only from the *Alicat* class). When I tried to implement this solution in the larger program itself, I found the same results – an exception was thrown that resulted in reboot. It was only by removing the S*erial IO* objects from the *PAS* object that the executable was successfully built and run. If the diagram in Figure 1 were redrawn, the aggregation line between the *Serial IO* class and the PAS class would be removed and the inheritance under the *Alicat* class would stop at *Controller*.

Conclusion

The system is operational and is something I can work with, but it is not my ideal. At best, the *Serial IO* class is intended to be generic and reusable. But, because of the issues with the children classes, I have had to develop some classes that are specific to their instantiation. In addition, I am duplicating some actions in the main program (such as configuration and data acquisition) due to the decoupling of the serial IO from the main class that would naturally fit into the main class. Regardless of whether this is considered good programming practice, the results point to the fact that things are not correct on the target when the executable is run.