# Software control of the Advanced Technology Solar Telescope enclosure PLC hardware using COTS software

Alastair J. Borrowman*[a], Lander de Bilbao[b], Javier Ariño[b], Gaizka Murga[b], Bret Goodrich[c], John R. Hubbard[c], Alan Greer[a], Chris Mayer[a], Philip Taylor[a].
[a]Observatory Sciences Ltd, William James House, Cowley Road, Cambridge, CB4 0WX, UK;
[b]IDOM, Avda. Zarandoa 23, 48015 Bilbao, Spain;
[c]National Solar Observatory, 950 N. Cherry Ave. Tucson, AZ, USA 85726-6732.

## ABSTRACT

As PLCs evolve from simple logic controllers into more capable Programmable Automation Controllers (PACs), observatories are increasingly using such devices to control complex mechanisms [1, 2]. This paper describes use of COTS software to control such hardware using the Advanced Technology Solar Telescope (ATST) Common Services Framework (CSF). We present the Enclosure Control System (ECS) under development in Spain and the UK.

The paper details selection of the commercial PLC communication library PLCIO. Implemented in C and delivered with source code, the library separates the programmer from communication details through a simple API. Capable of communicating with many types of PLCs (including Allen-Bradley and Siemens) the API remains the same irrespective of PLC in use.

The ECS is implemented in Java using the observatory's framework that provides common services for software components. We present a design following a connection-based approach where all components access the PLC through a single connection class. The link between Java and PLCIO C library is provided by a thin Java Native Interface (JNI) layer. Also presented is a software simulator of the PLC based upon the PLCIO Virtual PLC. This creates a simulator operating below the library's API and thus requires no change to ECS software. It also provides enhanced software testing capabilities prior to hardware becoming available.

Results are presented in the form of communication timing test data, showing that the use of CSF, JNI and PLCIO provide a control system capable of controlling enclosure tracking mechanisms, that would be equally valid for telescope mount control.

**Keywords:** Allen-Bradley, ATST, communication, Linux, Java, JNI, PLC, PLCIO.

## 1. INTRODUCTION

The ATST is currently under development by the National Solar Observatory (NSO) of the USA; it will be the world's first 4m class solar telescope and is to be located on the summit of Haleakalā, Maui. The telescope's enclosure will be 25m in diameter and contain a number of controllable mechanisms including azimuth carousel, altitude shutter and aperture cover. The prime responsibility of the ECS is to receive tracking coefficients from the TCS (Telescope Control System) for azimuth and altitude positioning, convert these into hardware demands, send them to the Enclosure Mechanical Control System (EMCS) and read back status to ensure no enclosure hardware interferes with observations. The ECS has no responsibilities for enclosure thermal control as this is done by the Facility Thermal Control System.

The ECS has a number of constraints placed upon it:

- As a software system of the ATST it must be developed and operated using the observatory's CSF software infrastructure running on Linux.
- The EMCS will run on an Allen-Bradley ControlLogix PLC and connect to the ECS PC using a dedicated Ethernet link.

*ajb@observatorysciences.co.uk; phone +44 (0)1223 508257; fax +44 (0)1223 508258; www.observatorysciences.co.uk

## 2. THE SEARCH FOR AN ECS TO PLC COMMUNICATION SOLUTION

Prior to beginning the design of the ECS the project required a solution for communicating between ECS Linux PC and EMCS Allen-Bradley PLC.

### 2.1 Estimation of data to be transferred

In order to understand communications requirements an estimation of data to be transferred was made. Based upon experience of previous ECS development and using the TCS to ECS Interface Control Document (ICD) as a source for information of data the ECS will either pass to or receive from the EMCS the following was identified:

- Data to be sent from ECS to EMCS are of two types:
    - Trajectory stream to position enclosure's tracking axes; consisting of three items of type real containing demand position, demand velocity and timestamp, to be sent at 20Hz. Estimate of data length was 24 bytes for both trajectories (in a ControlLogix PLC the type real is stored in 4 bytes[3]).
    - One-off demands to set hardware position e.g. open the aperture cover, to be transferred as a logical bit in a data word as and when required. Estimate of data length was 2 bytes.
- Data to be read by the ECS from the EMCS are also of two types:
    - Tracking axes position data, consisting of four items of type real containing current position, current position error, current velocity and timestamp, to be read at 20Hz. Estimate of data length was 32 bytes.
    - Hardware status including motor torques, drive power, etc., contained in a mixture of reals, integers and logical bits contained in data words, to be read at 1Hz. Estimate of data length was 235 bytes.

### 2.2 PLC Ethernet connection

The ControlLogix PLC (also known as a PAC) is not packaged with built-in Ethernet. For remote connections, the user plugs a module into the PLC's backplane supplying the capabilities required. Communication across the backplane uses the Common Industrial Protocol (CIP). The native network application layer protocol supported by Allen-Bradley communications modules running over TCP/IP Ethernet is *EtherNet/IP*, where IP stands for *Industrial Protocol*. EtherNet/IP is the name given to CIP as implemented over standard Ethernet[4] (see Figure 1). The protocols CompoNet, ControlNet and DeviceNet also use CIP but not over Ethernet.

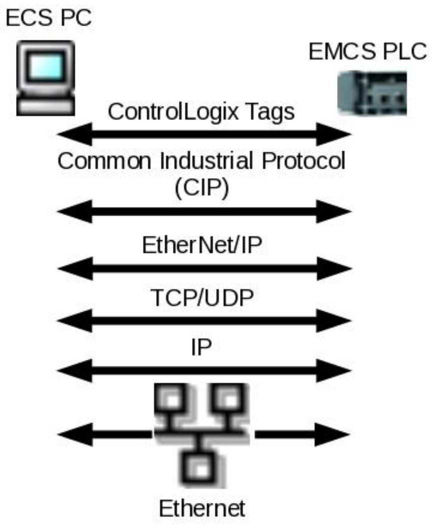

Figure 1. ECS to EMCS EtherNet/IP communications stack.

EtherNet/IP and CIP are registered trademarks of the Open DeviceNet Vendor Association (ODVA). Access to EtherNet/IP specifications and Vendor IDs are controlled by them and only granted to their members, minimum membership cost is $1000. In this paper *Ethernet* is used to refer to the network link layer commonly used to deliver TCP/IP, and *EtherNet/IP* is used to refer to the network link layer protocol used by Allen-Bradley communication modules to communicate over Ethernet networks.

For the ECS, in the above communications stack the TCP/UDP layers and below are supplied by Linux and the observatory's network infrastructure, the EtherNet/IP layers and above must be supplied by the ECS. As an alternative to this there are PLC backplane modules available from Allen-Bradley partner manufacturers that provide the capability to transfer PLC data as ASCII text across TCP/IP. At the time of investigation (early 2011), available modules were the 56SAM (Special Applications Module) and eATM xCoupler both manufactured by Online Development[5].

The 56SAM contains a CPU running Linux and is delivered with libraries enabling direct access to ControlLogix data across the PLC backplane. Unfortunately the version of Linux used is outdated as it runs kernel version 2.4 (superseded by 2.6 in 2003) and the communication libraries are only supported for this. Therefore development and ongoing maintenance for this module through the observatory's lifetime will be difficult, making it unsuitable for the ECS.

The eATM xCoupler provides a TCP/IP connection to the PLC. It is configured and managed using a program called WorkBench that runs on a PC (Linux supported). The WorkBench utility is used to define how ControlLogix data from the PLC are made available over the network. However, the module does not support bi-directional TCP data that is a requirement of the ECS, but rather is designed to be used to gather data to be placed in reports (e.g. production line productivity).

The possibility of using OPC UA (open connectivity via open standards – Unified Architecture) as an alternative to EtherNet/IP was also investigated. At the time of investigation Allen-Bradley only supported OPC UA connections to ControlLogix through its Windows only network connection software RSLinx. Therefore to use an OPC UA connection from Linux would depend upon finding an OPC UA server capable of talking EtherNet/IP. A product from Inductive Automation[6] called Ignition was found, however this is web-server based and does not publish an API accessible through a software library. Other OPC UA servers may be available from other manufacturers that do run on Linux and publish an API; however, as no Allen-Bradley ControlLogix compatible OPC UA module is currently available the OPC UA calls will always be translated into EtherNet/IP before access is made to PLC data. This adds another layer to the communications that adds complexity and increases software maintenance overhead, therefore if a suitable EtherNet/IP library could be found this was recognized as the preferred solution, as shown in Figure 2.

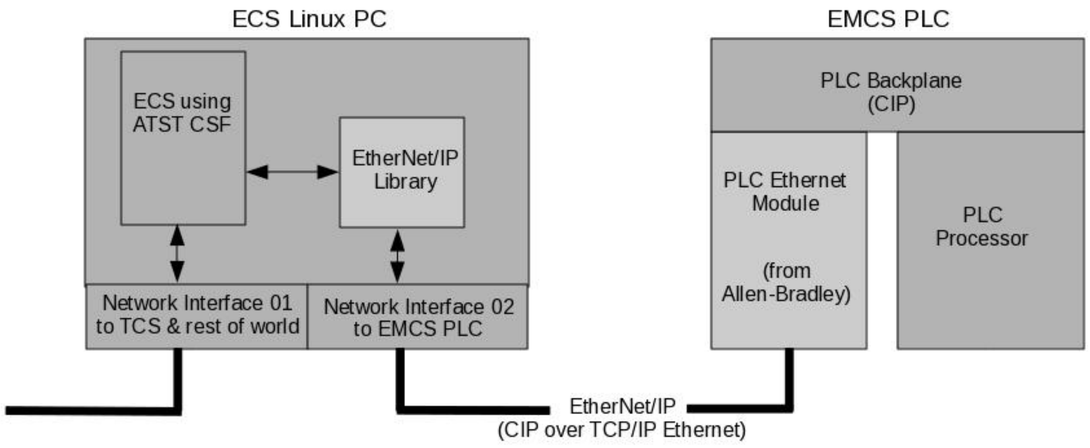
ECS Linux PC
ECS using ATST CSF
EtherNet/IP Library
Network Interface 01 to TCS & rest of world
Network Interface 02 to EMCS PLC
EMCS PLC
PLC Backplane (CIP)
PLC Ethernet Module (from Allen-Bradley)
PLC Processor
EtherNet/IP (CIP over TCP/IP Ethernet)

Figure 2. ECS to EMCS Ethernet connection using EtherNet/IP.

### 2.3 EtherNet/IP software library

To find a suitable library to provide EtherNet/IP connection functionality to the ECS various options were considered, these included:

- Using an open source library (e.g. EPICS EtherIP) as a base for developing a library suitable for ECS requirements. This solution was considered unsuitable due to requiring an in-depth knowledge of CIP and EtherNet/IP, and increase to ECS development time.
- Using a COTS CIP/EtherNet/IP network stack development kit to develop our own library. Once again this solution would require a good knowledge of CIP and EtherNet/IP, moreover the cost of the kits ranged from $6000 to $17000.
- Using a COTS CIP/EtherNet/IP library. Of the two discovered, only PLCIO from Commercial Timesharing Inc. (CTI)[7] seemed a suitable solution and priced at $1795 meant limited investment was needed to enable evaluation to be carried out; CTI do not provide evaluation copies of PLCIO as it is only supplied in source code format.

### 2.4 The PLCIO EtherNet/IP Library

PLCIO is written in C and was initially developed for UNIX; it is delivered with all source code following signing of a non-disclosure agreement. It provides a simple file I/O API (open, close, read, write) to communicate with various PLCs, including models from Allen-Bradley and Siemens. The API remains the same irrespective of PLC being used as PLCIO hides the underlying PLC communication protocol below the level of the API. The API function used to open a connection to the PLC contains a parameter used to specify the protocol in use, when using a ControlLogix PLC this is given the value *cip*.

The benefits of selecting PLCIO for use with the ECS were identified as:

- Removes the necessity of in-depth knowledge of EtherNet/IP and CIP from the ECS developer and maintainer.
- PLCIO uses a well-defined relationship between ControlLogix data representation in PLC and the equivalent representation in C programming language data structures. It includes a utility to read all PLC public data and output C data structure format that may be used in API function calls to read and write the data.
- PLCIO is contained in a C library that can therefore be linked with C++ and called from Java via JNI, and so can be easily used by applications developed in both CSF supported languages.
- PLCIO can communicate with many different brands of PLC and therefore accommodate possible future changes to the choice of EMCS PLC.
- Performance data available from the PLCIO website and from discussions with a third-party user of the library, suggested that our estimated communication data size and rates could easily be accommodated using PLCIO.
- PLCIO enables the ECS to EMCS ICD to be defined in terms of named ControlLogix tags that the ECS shall read and write to control the enclosure hardware. In ControlLogix PLCs the tag is the primary data structure used to access I/O addresses, bits, variables, timers, etc., they can contain scalars, arrays and be user defined to contain any combination of these data much like a C structure.

A downside to PLCIO is that it is not thread safe. The software using it must be designed to ensure only one of the library's API functions is called at any time.

## 3. USING THE PLCIO LIBRARY FROM WITHIN AN ATST CSF APPLICATION

### 3.1 Overview of ECS design using CSF

CSF uses the component/container model to provide distributed control to coordinate all observatory systems through command-action-response messaging[8]. The framework is implemented in Java and C++ and runs on Linux. The CSF provides services to the ECS to enable communication with other ATST software systems and all housekeeping, including health and alarm notification, logging and a persistent information store (Property Service). The ECS has been designed to make most use of the facilities provided by the CSF and only modify their behavior when necessary through normal OO techniques. Therefore the ECS has a single Management Controller sub-classed from CSF and five components sub-classed from suitable CSF component/controller classes, which each have responsibility for a specific aspect of the enclosure. The ECS component class diagram is shown in Figure 3; all super classes named on the diagram but not shown are from CSF.

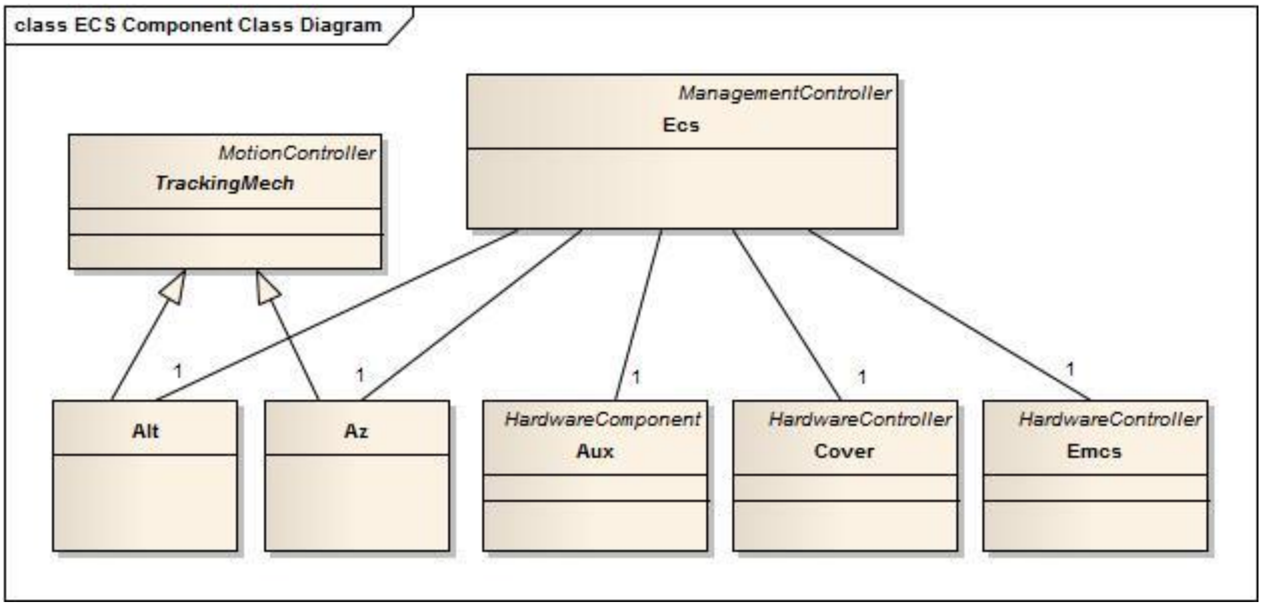

Figure 3. ECS component class diagram.

## 3.2 The CSF hardware connection paradigm and ECS hardware connection classes

Each ECS component that connects to hardware uses a Connection object that implements the CSF Connection interface. Using this interface the CSF calls the class's hardware connect and disconnect methods at the appropriate time based upon changes to the component's lifecycle, i.e. at *startup* and *shutdown* respectively. The connection class of the ECS is ABPlcioConnection – the connection class for Allen-Bradley PLCs using the PLCIO library. In addition to the methods of the CSF Connection interface ABPlcioConnection also implements an interface providing abilities specific to Allen-Bradley PLCs, e.g. the reading and writing of ControlLogix tag data. The Connection uses objects of the Channel class to read and write data to the hardware, which is done through the singleton Master class object that is responsible for all access to the PLC. In the ECS these classes are ABPlcioChannel and ABPlcioMaster. The design using Connection, Channel and Master classes and associated interfaces follows the CSF connection based approach used by other ATST developers to communicate with other hardware, e.g. the Delta Tau motion controller.

Even though CSF is implemented for both Java and C++ the preferred and most supported language is Java. Moreover controllers implemented in different languages cannot be run in the same container. To enable the ECS to run in one container and have full access to all CSF capabilities the decision was taken to implement all of the ECS in Java. Therefore JNI is used to access the PLCIO C library. As the ECS requires use of only four functions of the library the JNI can be kept to a minimum. Following the connection paradigm described above results in all calls to the JNI methods being contained in the singleton ABPlcioMaster class. This also ensures that Java synchronization can be used to prevent multiple simultaneous calls to the non-thread-safe PLCIO library. PLCIO can maintain multiple open communications channels to a PLC but only one read or write command, using one channel can take place at any time.

Figure 4 shows a diagrammatic representation of ECS controllers' use of Connection classes, JNI and PLCIO to communicate with the EMCS PLC.

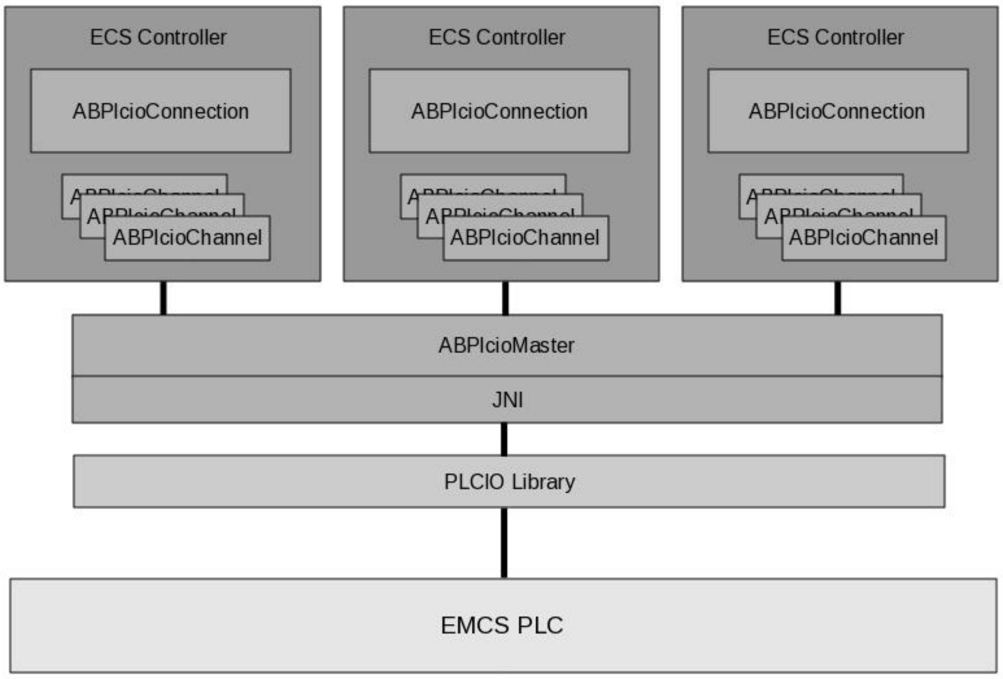

Figure 4. ECS controller use of Connection classes, JNI and PLCIO to communicate with EMCS.

### 3.3 Initial communication testing and results

On first delivery of the PLCIO library a simple C program was implemented to enable communications tests to be carried out with PLC hardware. The PLC was programmed to contain a number of ControlLogix tags the byte length of which could be modified for different tests. The C program was run from the command line and accepted a number of arguments to select read or write, tag identifier, frequency of tag transfer and number of transfer iterations. The following tests were carried out using a 32bit dual-core laptop PC with 4GB of memory:

- Effect of packet size was tested by increasing the size of a ControlLogix tag being written and read from tens to hundreds of bytes. The time for the write or read operation to complete remained stable between 1 and 2ms.
- Communication frequency achievable was tested by writing and reading a tag of byte length equating to that estimated for trajectory stream position data. A frequency range from 1 to 100Hz was tested with write and read operation times remaining stable between 1 and 2ms; the ECS frequency for trajectory stream data is 20Hz as this is the frequency at which the stream is updated by the TCS.
- To measure the effect of PLC processing load on the time taken to service write and read calls from PLCIO, a test was run with the PLC load at three different levels of stress; light, medium and heavy. The *medium* stress equated to the PLC processing load expected when running the EMCS. Under light and medium load the response times for write and read calls remained between 1 and 2ms. Under heavy load response times increased to between 7 and 8ms, which still remains within acceptable limits considering the fastest ECS processing will have a 50ms period between calls.

- The Linux Foundation's *netem* network emulation program[9] was used to test the effect of various network problems. The results are contained in the table below.

Table 1. Network emulation test results.

| Effect | Level | Mean read/write time (ms) | Max read write time (ms) |
|---|---|---|---|
| Packet loss | 1 % | ~ 1.5 | 280 |
| Packet loss | 10 % | 32 | 1600 |
| Duplication | 1 % | ~ 2 | 8 |
| Duplication | 10 % | ~ 2 | 8 |
| Duplication | 50 % | ~ 2 | 9 |
| Reordering | 25 % (delay 10 ms) | 12 | 22 |
| Reordering | 25 % (delay 20 ms) | 22 | 41 |
| Reordering | 5 % (delay 25 ms) | 28 | 39 |
| Corrupt | 1 % | 4.8 | 680 |
| Corrupt | 10 % | 24 | 1600 |

The above tests involved opening multiple connections to the PLC by running multiple copies of the test program simultaneously; no degradation was seen in the response to read and write requests irrespective of number of simultaneously open connections.

During initial testing we saw that approximately once in every one thousand iterations a read or write operation would be significantly delayed. Further investigation showed that connecting the PC to the PLC's Ethernet module directly, rather than through a switch, removed these delays. As yet no more investigations into network equipment effects have been carried out.

### 3.4 Soak test results

To measure the library's performance over a longer period a one hour soak test was performed. During the test the data transferred were write and read *fast* data at 20Hz of length 24 bytes and *slow* data at 1Hz of length 176 bytes for write and 76 bytes for read.

This test was run using two testing methods; #1 ran simultaneously four copies of the C program used for initial testing and #2 used a prototype CSF ECS Java application implemented following the design outlined in this paper, including connection classes and JNI access to PLCIO API functions. The prototype used four threads to read and write the four data types.

Results of running the tests on a 64bit quad-core desktop PC with 8GB of memory are shown in the following tables.

Table 2. Test method #1 results using C program.

| **#1 Results** | **Max read/write time (ms)** | **Min read/write time (ms)** | **Mean read/write time (ms)** | **Modal read/write time (ms)** | **Percentage total of read/writes accomplished in:** | |
|---|---|---|---|---|---|---|
| | | | | | **2ms** | **5ms** |
| **Write Fast** | 32 | 0 | 2.446 | 2 | 53.31% | 99.89% |
| **Read Fast** | 8 | 0 | 2.427 | 2 | 53.71% | 99.94% |
| **Write Slow** | 7 | 1 | 2.599 | 3 | 47.47% | 99.53% |
| **Read Slow** | 7 | 1 | 2.477 | 3 | 51.75% | 99.53% |

Table 3. Test method #2 results using CSF, Java and JNI.

| **#2 Results** | **Max read/write time (ms)** | **Min read/write time (ms)** | **Mean read/write time (ms)** | **Modal read/write time (ms)** | **Percentage total of read/writes accomplished in:** | |
|---|---|---|---|---|---|---|
| | | | | | **2ms** | **5ms** |
| **Write Fast** | 11 | 0 | 2.119 | 1 | 65.33% | 99.13% |
| **Read Fast** | 11 | 0 | 2.181 | 1 | 61.28% | 98.96% |
| **Write Slow** | 12 | 1 | 2.822 | 3 | 42.97% | 96.69% |
| **Read Slow** | 11 | 1 | 2.514 | 3 | 50.92% | 98.41% |

The following additional tests were also run using the ECS prototype:

- To understand where timing overheads occur split timings were recorded to detail time taken to acquire lock for Java method giving access to PLCIO, and time taken to call PLCIO from JNI, i.e. similar to time recorded by C program but with additional overhead of making call through JNI. Results from this test showed no undue delays while waiting for synchronization lock or calling PLCIO through JNI.
- Data integrity was checked by modifying the data on each iteration of test prior to writing to the PLC then reading back the data and checking values were equal. During testing no errors between data written and read occurred.
- To ensure passing all PLC communications through a single synchronized Java method is a scalable solution for the ECS, a test was carried out simulating the data transfer equivalent to running 5 tracking axes, i.e. running 5 times the number of threads as previous tests, reading and writing the same data as used for the soak test described above. This test involved transferring much more data than is required by the ECS, which is likely to have only 2 (or at most 3) tracking axes. The results showed an increase in the time taken to accomplish 98 to 99% of read/writes from 5 to 20ms; this still offers acceptable performance for the ECS.

Further details of all test software, testing environment and results can be obtained from this paper's primary author.

## 4. PLC SIMULATION USING PLCIO

The PLCIO library is delivered with what the manufacturer terms a *Virtual PLC*. This is an additional communications module that is used instead of the module used to connect to a real PLC, e.g. instead of using the CIP module as used for ControlLogix PLCs. As is the case when using any of the PLCIO's communication modules all access to the module is made through the library's API, the only change is in the argument value passed to the function used to open a connection to the PLC, e.g. instead of specifying CIP using the keyword *cip*, the keyword *virtual* is used.

As delivered the virtual PLC contains two blocks of data that can be written to or read from. When a connection is initially opened to the virtual PLC it obtains and opens two file descriptors, which it then uses to write and read binary data on demand. Due to the module design and structured interface used between the library's API front-end and PLC communications modules, the virtual PLC source code can be used as a template to create other virtual PLCs. This has been done to create a Virtual EMCS (VEMCS).

The VEMCS contains the ControlLogix data tags as defined in the ECS to EMCS ICD. For each tag the VEMCS uses a binary data file to store its current value, in this way the hardware state of the simulator is persistent between simulator runs. To use the VEMCS an ECS controller defines in its CSF Property Service data the connection class ABPlcioConnectionSim (as opposed to ABPlcioConnection); the Sim class extends ABPlcioConnection but overrides only its connection method. In this method the superclass obtains the IP address of the EMCS from an input parameter, prefixes *cip* and calls the PLCIO open PLC connection function, the simulator class' version is simply hardcoded to call the open function with the argument *vemcs*. This is the only difference in any ECS code between using the real EMCS and the simulator VEMCS.

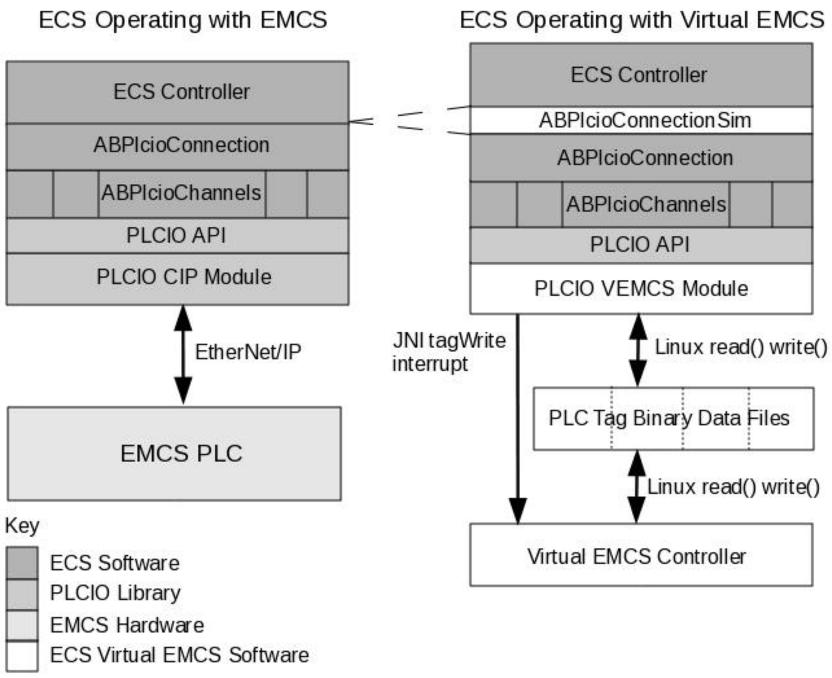

Figure 5. Comparison between ECS software structure using EMCS and VEMCS.

To enable the VEMCS to use features provided by CSF, e.g. servo simulation classes, the PLCIO VEMCS communication module uses JNI to communicate with a VEMCS CSF controller. All simulation of enclosure hardware behavior occurs in this controller. Moreover the controller provides a CSF enabled GUI giving users the ability to interact with the simulator to test the ECS, e.g. to simulate occurrence of an interlock, motor fault or timeout, etc. Moreover, CSF scripting facilities can interact directly with the VEMCS controller and therefore automated test scripts can be used that manipulate the simulators behavior to test the reactions of the ECS.

## 5. CONCLUSIONS

The ATST ECS design as presented in this paper and using PLCIO is currently in the first stage of implementation. The alpha release will be delivered in September 2012. The PC to PLC communication results detailed here together with the use of the VEMCS simulator give the project confidence that the chosen technologies will enable delivery of a well understood, robust and maintainable system, which will integrate well into the observatory's infrastructure.

## REFERENCES

[1] Kiekebush, M. J. et al, "Evolution of the VLT instrument control system toward industry standards", Proc. SPIE 7740, 77400T-1 (2010)
[2] Pessemeir, W., "Towards a new Mercator Observatory Control System", Proc. SPIE 7740, 77403B-1 (2010)
[3] Stenerson, J., [Programming ControlLogix Programmable Automation Controllers], Delmar Cengage Learning, Independence, KY 41051 (2009)
[4] ODVA, "A Guide for EtherNet/IP Developers", PUB00213R0, 2008, <http://www.odva.org> (26 May 2012)
[5] Online Development, <http://www.oldi.com> (26 May 2012)
[6] Inductive Automation, <http://www.inductiveautomation.com> (26 May 2012)
[7] Commercial Timesharing Inc. PLCIO, <http://www.ctiplcio.com> (26 May 2012)
[8] Hubbard, J. et al, "The ATST Base: Command-Action-Response in Action", Proc. SPIE 7740, 77402R-1 (2010)
[9] The Linux Foundation, "netem" <http://www.linuxfoundation.org/collaborate/workgroups/networking/netem> (26 May 2012)