ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA119

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

# ASYNCHRONOUS DRIVER EVALUATION AND DEVELOPMENT FOR DIGITAL SYSTEMS AT THE ARGONNE TANDEM LINEAR ACCELERATING SYSTEM*

C. E. Peters†, J. Reyna, D. Stanton, Argonne National Laboratory, Lemont, USA


### *Abstract*

The ATLAS (Argonne Tandem Linear Accelerating System) accelerator at Argonne National Laboratory, near Chicago, IL., has recently been upgraded via the addition of a pulsed mode Electron Beam Ion Source (EBIS). Pulsed operation requires finer levels of control of various digital systems like fast switching high-voltage power supplies and remotely controlled function generators. Additionally, pico-level and femto-level ammeters need per-device zero correction and calibration to accurately read beam intensities. As the facility moves away from fast register-based analog signals, new and slower digital protocols adversely affect the perceived execution time of the control system. This work presents options, research, and results of implementing an asynchronous layer between high level user interfaces and the low level communication drivers in order to increase the perceived responsiveness of the system. Solutions are evaluated ranging from in-house codes, which implement system-wide mutual exclusion and prioritization, to drivers available from the EPICS control system. Key performance criteria include ease of implementation, cross platform availability, and overall robustness.


## INTRODUCTION

The ATLAS accelerator is located at the United States Department of Energy's Argonne National Laboratory in the suburbs of Chicago, Illinois. It is a National User Facility capable of delivering ions from hydrogen to uranium [1] for low energy nuclear physics research in order to perform analysis of the fundamental properties of the nucleus. The majority of the current control system has been based on a CAMAC Serial Highway [2] (SH) architecture since the 1980s. Access to this hardware bus from software relies on PCI based personality cards which in turn connect to the serial highway. While this system is clearly outdated from a technology progress perspective, it continues to provide distributed serial networking with low latency and high reliability. This improves the perceived responsiveness of the control system and allows simple single-threaded access via the use of the operating system's register-based PCI subsystem interface.

Moving away from CAMAC and fast register access has commonly been accomplished by interfacing to (non-CAMAC) serial devices in the form of USB/RS-232/RS-485 specifications. However these devices use slower baud rates and typically control more complicated devices. This results in longer latency delays for each command. It should be noted this application is for a 'slow' control system and all values are only updated at about ~1-2Hz.

### *ATLAS Control System Software Description*

The ATLAS Control System (ACS) group only consists of 2 – 5 full-time members, depending if the definition includes students and temporary assignments. Therefore, a third-party vendor Vista Control Systems, Inc. [3] is used to provide software libraries to supplement the creation of database structures, operator interfaces, logging tools, etc. The EPICS control system is acknowledged to be the largest and most comprehensive offering in the space, however the amount of overhead to implement and maintain a large and diverse open-source package can be prohibitive for small groups. Even borrowing individual modules like the EPICS Asyn driver [4] can be resource prohibitive unless the group has already committed to the full EPICS ecosystem.

## BENCHMARKS

In order to implement a modern solution to register base CAMAC which do not cause the main operator interface (OPI) to lock, we need to understand the current level of latency in the existing software/hardware loop.

### *CAMAC/PCI/OPI Latency*

- Single core 400 MHz Alphaserver 1200 CPU running OpenVMS 8.2 with idle CPU usage and 1GB memory.
- Kinetic Systems 2115 PCI Serial Highway Driver running in byte-wise mode at 2.5MHz clock speed
- Single 16-bit CAMAC NAF Operations

Table 1: CAMAC Execution Latencies

| Operation | # of Calls | Time | Latency |
|---|---|---|---|
| 16-bit Read | 1,000,000 | 47 sec | 47 µSec |
| 16-bit Write | 1,000,000 | 47 sec | 47 µSec |
| Fast Process* | 100,000 | 38 sec | 380 µSec |
| OPI Slider | 5000 | 262 sec | 5,240 µSec |

* A Data acquisition process running at its fastest software loop

### *Non-CAMAC Serial Latency*

It is noted here that raw CAMAC latency is quite low (see Table 1). This will be difficult to match. However as more and more software overhead is added, the latency for each loop of software <u>adds</u> to the hardware latency, and the actual required latency of any replacement system becomes more reasonable. The fastest process running on the SH is only about 0.5msec of latency, and the human interfaces

* This work was supported by the U.S. DOE, Office of Nuclear Physics, under Contract DE-AC02-06CH11357. The research used resources of ANL's ATLAS Facility, a DOE Office of Science User Facility.
† ChrisPeters@anl.gov

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA119

(sliders) execute 10 times that at ~5 milliseconds. Table 2 represents latency testing on alternatives to CAMAC serial highway communication architectures.

These times can now be compared to modern serial:

- Single core Intel Pentium 4 at 2.8GHz running Scientific Linux 6.7 with idle CPU usage and 1GB memory.
- Weiner CC-USB CAMAC Crate Controller
- Linux TCP/IP Packets with 16-bit payload.

Table 2: Non-Camac Serial Execution Latencies

| Operation | # of Calls | Time | Latency |
|---|---|---|---|
| 16-bits@57600 | (theoretical) | N/A | 277 µSec |
| USB CAMAC | 1,000,000 | 265 sec | 265 µSec |
| TCP@100Mbit | 1,000,000 | 293 sec | 293 µSec |
| Keithly6514 | (single read) | N/A | 30,000 µSec |

The overall conclusion of this testing is that the existing CAMAC latency is in the ~40µSec range and modern serial replacements are about an order of magnitude more. This drives motivation to develop our own set of software layers which provide multi-threaded support such that the operators do not notice a significant increase in lag.

## ASYNCHRONOUS SOLUTION

We can now assemble a list of requirement for our new software layer which will enable highly responsive applications for this specific set of ACS software libraries.

### *Requirements*

The software solution should be simple and be based primarily on Linux, as this is the common operating system of the ACS. It should use native operating system primitives to accomplish locking and memory sharing. In addition, this layer should be aware of priorities of executing threads determined by assigning a write higher priority than read. In general the C language is used unless there are specific object-oriented runtime advantages.

### *Architecture*

It is important to note that VSystem is based on a remote procedure call (RPC) signalling system which spawns multiple processes accessing a single channel. Therefore there are 2 processes running handler code as shown in Figure 1.

### *Shared Memory Port Locking Algorithm*

There are two sets of procedures depending on if the call to the serial handler is a high priority or a low priority. At this time, there are only 2 implemented priorities. In the example below, the main OPI thread spawns a worker thread and immediately returns, allowing the user to continue interacting with other devices or functions.

In summary, a high priority thread needs to reserve only the mutex, but a low priority thread has to reserve both the mutex and the semaphore thereby causing a higher probability the higher thread will execute first.

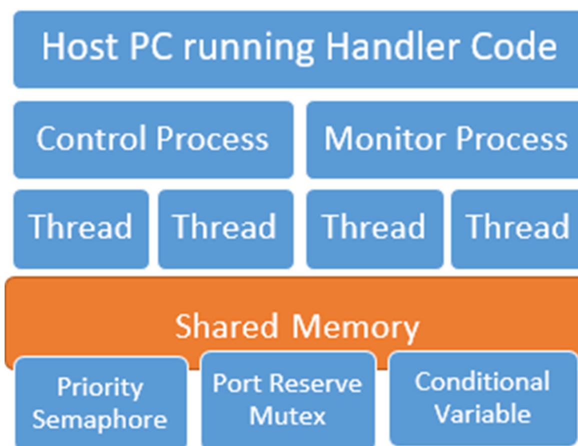


Figure 1: Hierarchy of key software components.

### *Higher Priority Thread*

1. Call "sem_wait" on a semaphore in shared memory to signal other threads a high priority thread is waiting.
2. Once semaphore is locked, it signals that other lower priority threads have paused and we are clear to run.
3. Attempt to lock shared memory's port mutex to acquire rights to the shared serial port.
4. Now that we have acquired the port, "sem_post()" the high priority semaphore to signal no longer waiting.
5. Do long running serial transaction….
6. When done, call pthread_cond_broadcast() on the semaphore to wake up sleeping low priority threads.
7. Finally, unlock the port mutex to release the port.

### *Low Priority Thread*

1. Block on attempting to lock shared memory's mutex to acquire rights to the port (note: no semaphore lock).
2. Call sem_getvalue() to determine if there is currently a higher priority thread waiting, if not do transaction.
3. If the value returned by sem_getvalue() is zero, then enter a conditional wait loop. The conditional wait also releases the mutex allowing other processes run.
4. Get woken up by a signal from high priority thread when the port mutex is acquired by our process.
5. Once the high priority semaphore is non-zero, do long running serial transaction.
6. Call pthread_cond_broadcast() on the semaphore to wake up any other sleeping low priority threads.
7. Finally, unlock the port mutex to release the port.

### *Shared Memory Layout*

A note about shared memory is that it is assumed any process has already acquired a pointer to this section by supplying a common handle identifier across processes.

```
struct stSharedMemVars
{ pthread_mutex_t mutex;
  pthread_cond_t conditionVar;
  sem_t semHighPriority;
};
```

The purpose of the mutex is to represent "mutual exclusive" access to a resource like a serial port. The purpose of

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA119

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

the condition variable is to signal low priority threads to sleep and be woken up asynchronously when a higher priority thread completes. The purpose of the high priority semaphore is to maintain a count of the number of processes system wide that are high priority, and then asynchronously signal the other processes to wake-up and resume attempting to lock the shared resource (port).

## IMPLEMENTATION AND TESTING

The asynchronous and multi-threaded method described above has been implemented in the ACS in two different pieces: 1.) An example high-level handler for Keithley Picoammeter 6514 device driver has been re-written such that blocking calls to the lower serial layer are moved to their own "threadEntry" function with new calls to spawn those threads, and 2.) The lower level serial device code has been modified to implement the locking algorithm based on shared memory described in the "Algorithm" sections above.

****For the following tests an artificial 0.5 second delay was added to the serial port driver to simulate a particularly low baud rate or long running process.****

- Send Qty 4 high priority commands to the device simultaneously, verify main thread returns quickly.
- Send Qty 4 high priority commands to the device simultaneously, and verify the low priority threads wait.

```
controlsys@acs082:/atlas/programs/ATLAS_Native_Serial_Handlers
NEW Thread Index[0] = Chix 18, Thread ID b5a80b70
        PARENT THREAD Returned in 0.000081 seconds.
NEW Thread Index[1] = Chix 19, Thread ID b4cffb70
        PARENT THREAD Returned in 0.000126 seconds.
NEW Thread Index[2] = Chix 20, Thread ID b42feb70
        PARENT THREAD Returned in 0.000130 seconds.
NEW Thread Index[3] = Chix 21, Thread ID b38fdb70
        PARENT THREAD Returned in 0.000131 seconds.
        0.500521 seconds for command SENS:CURR:RANG 2E-6

        0.507705 seconds for command :SYST:ERR?

Thread Done: ID b5a80b70
        0.500523 seconds for command SENS:CURR:RANG 2E-5

        0.507940 seconds for command :SYST:ERR?

Thread Done: ID b4cffb70
        0.500524 seconds for command SENS:CURR:RANG 2E-4

        0.507969 seconds for command :SYST:ERR?

Thread Done: ID b42feb70
        0.500524 seconds for command SENS:CURR:RANG 2E-3

        0.507998 seconds for command :SYST:ERR?

Thread Done: ID b38fdb70
```

Figure 2: Main OPI thread returns in <1mS.

```
controlsys@acs082:/atlas/programs/ATLAS_Native_Serial_Handlers
Thread Done: ID b53ffb70
        PARENT THREAD Returned in 0.000012 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
        0.830733 seconds for command READ?

Thread Done: ID b49feb70
WARNING - Thread b53ffb70 sleeping inside Cond. Wait...
        PARENT THREAD Returned in 0.000012 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
WARNING - Thread b53ffb70 sleeping inside Cond. Wait...
        PARENT THREAD Returned in 0.000012 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
WARNING - Thread b53ffb70 sleeping inside Cond. Wait...
WARNING - Thread b49feb70 sleeping inside Cond. Wait...
        PARENT THREAD Returned in 0.000012 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
WARNING - Thread b53ffb70 sleeping inside Cond. Wait...
WARNING - Thread b49feb70 sleeping inside Cond. Wait...
        PARENT THREAD Returned in 0.000013 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
WARNING - Thread b53ffb70 sleeping inside Cond. Wait...
WARNING - Thread b49feb70 sleeping inside Cond. Wait...
        PARENT THREAD Returned in 0.000013 seconds.
        PARENT THREAD Returned in 0.000006 seconds.
        0.510531 seconds for command SENS:CURR:RANG?
```

Figure 3: Low priority threads sleeping.

Figure 2 is a debugging printout from the higher priority control process. Here we can see 4 threads being spawned with different thread IDs. Next, we see one thread at a time acquire a lock on the serial port and execute several long-running commands which would normally block the parent thread. At the bottom we see each thread in turn complete with no errors. Additionally, the parent thread returns generally in the microsecond range.

At the same time this is happening, Figure 3 shows a second monitor process polling 2 values from the device. When the high priority threads run, the block execution of the lower priority threads. We can see the low priority threads sleeping, but still returning control to the calling thread within several microseconds. Eventually the low priority threads get released from their conditional wait and begin sending messages once again.

## CONCLUSION

The purpose of this work is to implement asynchronous and priority based threading on top of the regular serial port driver code. While this type of layer is common in control systems like EPICS which have a single process per port, it does not come for free on other software packages. We have demonstrated that by using a combination of shared memory space, shared mutex and condition variables, and careful thread management, a similar feature to EPICS Asyn can be implemented and drastically decreases perceived execution time of the control system to lower than at least the ~13mS theoretical latency of the human eye itself [5].

## ACKNOWLEDGEMENTS

This work was supported by the U.S. Department of Energy, Office of Nuclear Physics, under Contract No. DE-AC02-06CH11357. The research used resources of ANL's ATLAS Facility, a DOE Office of Science User Facility.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA119

## REFERENCES

[1] "Stable Beams Available from ATLAS", www.phy.anl.gov/atlas/facility/stable_beams.html, retrieved October 12, 2015.

[2] *IEEE Std. IEEE Std. 595-1982, Standard Serial Highway Interface System,* The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.

[3] Vista Control Systems, Inc. https://www.vista-control.com/

[4] EPICS Asyn Driver, https://epics.anl.gov/modules/soft/asyn/

[5] Potter, M.C., Wyble, B., Hagmann, C.E. *et al.*, Atten Percept Psychophys (2014) 76: 270. doi.org/10.3758/s13414-013-0605-z

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.