ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MODPL03

EXPERIENCE UPGRADING CONTROL SYSTEMS AT THE GEMINI TELESCOPES

A. Núñez[†], I. Arriagada, T. Gaggstatter, P. Gigoux, R. Rojas, M. Westfall,
Gemini Observatory Southern Operations Center, La Serena, Chile
M. Rippa, R. Cárdenes, Gemini Observatory Northern Operations Center, Hilo, Hawaii

Abstract

The real-time control systems for the Gemini Telescopes were designed and built in the 1990s using state-of-the-art software tools and operating systems of that time. These systems are in use every night, but they have not been kept up-to-date and are now obsolete and also very labor intensive to support. This led Gemini to engage in a major effort to upgrade the software on its telescope control systems. We are in the process of deploying these systems to operations, and in this paper we review the experience and lessons learned through this process and provide an update on future work on other obsolescence management issues.

INTRODUCTION

The Gemini Observatory consists of twin 8.1-meter diameter optical/infrared telescopes, which provide full sky coverage from their locations on Maunakea in Hawaii (first light 1999) and Cerro Pachón in Chile (first light 2000). Most of the control systems for these telescopes were developed in the late 1990s using the technology and techniques of that time.

The overall software architecture for both telescopes is identical, and organized in five logical groups: Observatory Control System (OCS), Adaptive Optics Systems (AO), Telescope Control System (TCS), Instrument Systems and the Data Handling System (DHS) as shown in Fig. 1.

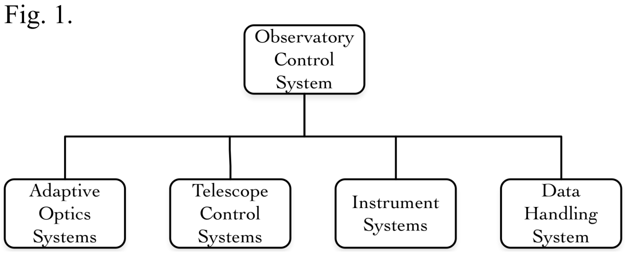

Figure 1: Gemini Software Architecture.

This architecture includes both the hardware and software necessary to operate the telescopes and its instruments, coordinated through the Observatory Control System (OCS) [1].

The Experimental Physics and Industrial Control System (EPICS) [2] was adopted as the standard control framework in which to run the facility subsystems, specifically for real-time control. A Standard Instrument Controller [3] on Versa Module Europa bus (VME) hardware was developed to ensure conformity between externally developed subsystems.

† anunez@gemini.edu

A more detailed view of the Telescope Control System and its components for both telescopes is shown in Fig. 2.

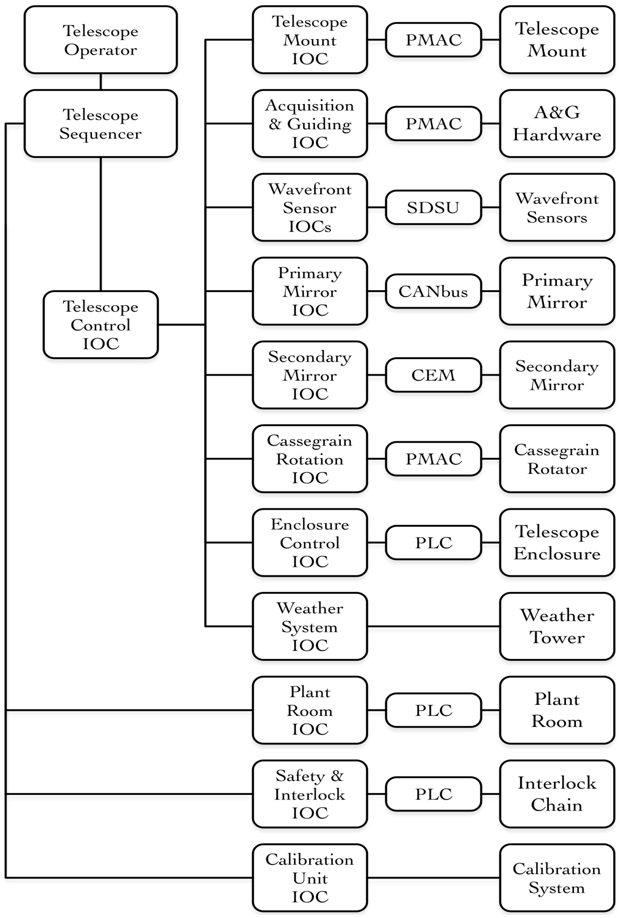

Figure 2: Gemini Telescope Control Architecture.

The Gemini software model makes a clear distinction between the low level software that directly controls and coordinates the telescope and instrument hardware and the high level software that interfaces with the users and sequences this hardware to perform queue scheduled science observations. All low-level Gemini software applications are classified as real-time even though the application may not have any hard real-time performance requirements.

While the high-level software has evolved over the years, the real-time control software has, with the exception of minor modifications to add features and fix problems, remained relatively unchanged. The real-time software development environment used to maintain these systems is likewise rooted in the 1990s. It has evolved organically over time, consisting of many different build processes; code repositories; development operating systems; customized hardware drivers; and a distributed support code base unique to each application.

Gemini has started a project to reduce long-term operational costs by upgrading and standardizing the real-time software for the Telescope Control Systems, its tools and development processes. This paper provides an overview of the project and its current status, lessons learned and next steps we plan to take to address other obsolescence issues in the observatory.

UPGRADE PROJECT

During 2011 and 2012, a study to determine the feasibility of upgrading the real-time systems was executed [4]. The motivation was to reduce ongoing real-time software development and maintenance effort by at least one FTE while reusing as much hardware, code and user interface infrastructure as possible. Also, we wanted to reduce licensing costs by using open-source tools wherever possible. Finally, we aimed to improve software development efficiency by adopting new standards based on successful practices developed by the telescope and particle physics community.

The Real Time Upgrade project was formally approved in January 2014 after this feasibility study was completed. Many aspects of our current operations were re-evaluated to create an effective environment for the future.

Feasibility Study Analysis

A complete review of the 40+ real time systems was performed and the results documented for subsequent analysis. This established, for each system, a baseline record of: basic details (system description and photographs), hardware configuration (architecture; processor board; peripheral boards), software configuration (boot image; operating system and version; hardware drivers; custom records, drivers, and support modules), and an overall system assessment (stability, fault history, performance issues; and maintenance issues).

In parallel, we reviewed the work done at other institutions performing similar upgrades, including Keck Observatory [5], Diamond Light Source [6] and the Canadian Light Source (CLS) [7]. This allowed us to identify common issues solved at these sites and come up with an approach that would address Gemini's objectives of reducing software maintenance effort by increasing system stability and standardizing systems and processes.

The following are the main conclusions of the situation analysis:

1. The upgrade project would focus on upgrading the Telescope Control Systems, as these are the ones that form the core of Gemini's real-time software architecture. Future projects can address Instruments and AO systems.
2. The EPICS framework is deeply embedded in every aspect of our operations. EPICS is a well-supported and adequate environment for our telescope control needs, and many of the operational issues currently experienced stem from different versions of EPICS (and more specifically its Channel Access component) operating simultaneously on the same network. The decision was made to retain it as the core real time control framework, but upgrade it to a current stable version.
3. VxWorks [8], the standard operating system used for the telescope control systems resulted in recurring licensing fees that have prevented us from keeping it up-to-date. As there are no plans to develop new VxWorks based systems in the future, exploring alternatives to VxWorks would provide an opportunity for substantial cost savings for both Gemini and its system developers.
4. The Gemini telescope subsystems and instruments have been, and continue to be, developed by different institutions. As a result, independent copies of common device support routines, drivers and libraries were shipped with each of these systems. Over time these copies have diverged, either because they were originally customized to fit a specific system need, customized over time to add new features, or because bug fixes were only applied to some of them. There are now many unique copies of drivers and support routines such as drvSerial, Allen-Bradley PLC, Delta-Tau PMAC, Xycom-240, VMIC, SDSU/ARC, and others. Resolving these differences and creating a common code base for all real-time applications would reduce the effort required to maintain existing systems and would provide a standard base for future development. A common code base would also allow bug fixes and EPICS upgrades to be applied to all systems in a coordinated manner.
5. The software development tools, frameworks and processes in use at Gemini were reviewed. We had a number of configuration control and build/deployment systems in use. To reduce cost and increase efficiency, we decided to standardize these tools and use a modern software development environment to perform the upgrades.

Upgrade Strategy

Our analysis suggested the upgrade strategy depicted Fig. 3 below.

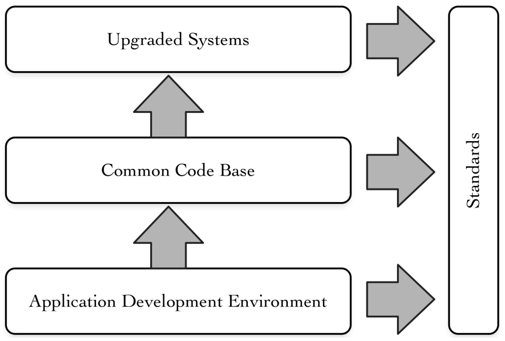

Figure 3: Real Time Upgrade Project Strategy.

The first goal was to create a unified real-time application development environment based on current operating systems, revision control procedures, packages and tools.

Using this environment, we would consolidate and standardize all the software modules to create a common, stable code base that can be reused by all the supported applications. This is our Common Code Base.

Finally, taking advantage of the common modules and the new software development environment, we would upgrade all the facility telescope systems.

In order for Gemini to avoid getting into the same situation that prompted the development of this project, we also need to produce new standards for future real-time software development.

Considering all these aspects, we broke the project down into the work packages shown in Table 1. In the following sections, we will review the status of each one of these, highlighting major achievements and lessons learned.

Table 1: Real-time System Software Upgrade Work Packages

WBS	Description
Application Development Environment	Create a real-time application development environment based on the latest operating systems, packages and tools.
Common Code Base	Replace obsolete software packages and merge divergent software libraries and drivers to create a common, stable code.
Telescope Facility Systems Upgrade	Upgrade the IOCs for all core Telescope Facility Systems to use the new environment and common code base.
Develop New Control System Standards	Develop real-time control software standards and architectures for future real time system development.
Project Oversight	Apply best practice project management processes to ensure project success and effective use of resources.

UPGRADING THE GEMINI APPLICATION DEVELOPMENT ENVIRONMENT

The Gemini Real-Time Application Development Environment (ADE, also referred to as "the environment") is used to develop and maintain software for the Gemini Telescopes. Upgrading the environment was the first phase of the overall upgrade project. The new development environment is the foundation for all system upgrade work and for new real-time systems developed in the future. We contracted out the implementation of the ADE to Observatory Sciences Limited (OSL) [9].

ADE Components

The new environment consists of an interrelated set of packages and components designed to allow efficient software development, as shown in Figure 4.

OSL completed a number of trade studies to determine the most effective set of tools that would form the ADE. Table 2 summarizes the main components that were selected. We discuss on the rationale for these decisions further in [4].

One important decision was to use the Operating System Independent (OSI) layer that has been introduced to EPICS [10]. This layer encapsulates specific operating system functions and resources (semaphores, threads, sockets, etc.), enabling EPICS applications to run on any operating system for which an OSI layer exists. With this, we add more flexibility to our systems by making them more portable in the future.

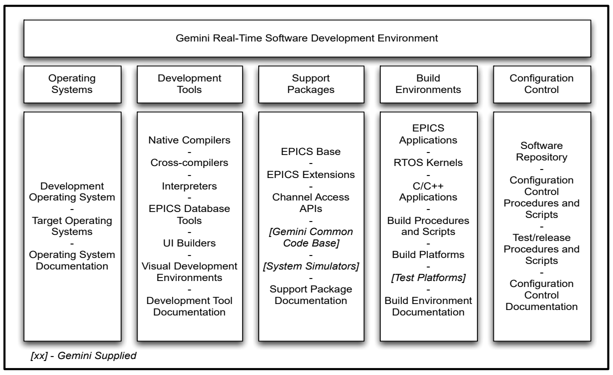
Gemini Real-Time Software Development Environment
Operating Systems
Development Tools
Support Packages
Build Environments
Configuration Control
Development Operating System - Target Operating Systems - Operating System Documentation
Native Compilers - Cross-compilers - Interpreters - EPICS Database Tools - UI Builders - Visual Development Environments - Development Tool Documentation
EPICS Base - EPICS Extensions - Channel Access APIs - [Gemini Common Code Base] - [System Simulators] - Support Package Documentation
EPICS Applications - RTOS Kernels - C/C++ Applications - Build Procedures and Scripts - Build Platforms - [Test Platforms] - Build Environment Documentation
Software Repository - Configuration Control Procedures and Scripts - Test/release Procedures and Scripts - Configuration Control Documentation
[xx] - Gemini Supplied

Figure 4: Real Time Application Development Environment.

Software Development Process Framework

The lack of a standard software development framework was a major contribution to maintenance overheads. Reviewing the number of configuration control, build and deployment systems we had in use, we concluded that a generic framework and set of processes for real-time software development should be similar to what is shown in Figure 5 below.

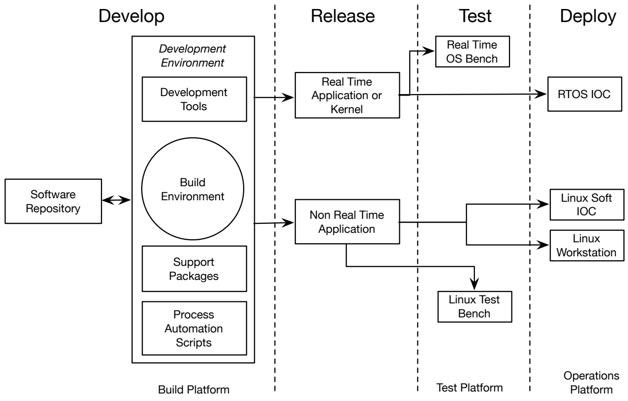

Figure 5: Gemini Software Development Framework and Processes.

After reviewing the software development frameworks and processes at various institutions, the SVN based Diamond [6] Application Development Environment was found to be the best match to the desired Gemini software development process.

Table 2: Development Environment Component Selection

Software Component	Current Configuration	Feasible Alternatives	Selected Option
Real-Time Operating System	vxWorks 5.4.2	vxWorks 5.5.1; vxWorks 7.0; RTEMS; QNX; Linux	RTEMS
EPICS Version	EPICS 3.13	EPICS 3.14 ; EPICS 3.15; EPICS 4	EPICS 3.14
EPICS database tool	Capfast	VDCT; TDCT; Custom	TDCT
Display Tool	DM and EDM	CSS/BOY; epicsQt; caQtDm	epicsQt
Configuration Management tool	CVS	SVN; GIT	SVN

Diamond's features include a comprehensive, standard directory structure; This structure defines the areas where different types of software modules can be accessed and built. It also provides standard locations, which reflect the software's functionality and maturity. Also, Python scripts standardize the processes in the software development cycle. Finally, an automated build server ensures that a comprehensive set of built software is kept up-to-date. The standard EPICS Build conventions are adopted (using GNU Make), with enhancements including new rules, additional templates, macros, configuration files and consistency checking features.

OSL delivered the Application Development Environment at the end of 2014. Since then, the ADE has been a useful tool for Gemini to manage its release process from development to production, and has helped to standardize the way real time software is managed in the Observatory.

More than two years after its introduction, two main improvements to the Gemini ADE have been identified:

Incorporate a way to manage deployment to different sites more explicitly. Gemini operates two telescopes located in different locations. The systems that run at each location are almost identical, but due to small differences in hardware, the software needs to be adjusted slightly to operate at each site. Right now, we maintain copies of the same software per site. This requires manual effort to keep both sites synchronized with the latest bug fixes, and it is not sustainable. We are exploring options to manage all common software in just one location, and relegate only the specifics to each site.

There are many ways this can be achieved, but due to the project schedule we decided to treat it as an improvement that will be done after the project is completed.

Use GIT instead of SVN. SVN came with the framework we adopted from Diamond. Although SVN is a good tool for this project, GIT provides several advantages that we could benefit from. In particular, as Gemini is a geographically distributed organization, GIT would simplify the source configuration control by simplifying the management of distributed development.

In addition, GIT provides simpler ways of branching and local experimenting that would be beneficial for some of the upgrade work we have done so far.

CREATING THE COMMON CODE BASE

With the ADE commissioned we were ready to start the next phase, the Common Code Base (CCB). The CCB represents the foundation of the entire upgrade project, since it encompasses all the common libraries and drivers used by all the telescope systems that were subject to upgrade.

This work required an exhaustive analysis of all the existing system software on a module-by-module basis to identify which have diverged, and why. Identical copies of modules were removed from each system and put into a common library. Modules that have evolved over time because of upgrades and bug fixes were brought up to the latest version and put into the library. For drivers or libraries that were customized to accommodate special system needs we used three different strategies depending on the specific case:

- Modify the telescope system software to remove the need for this customization.
- Move the customization to a separate and smaller module and move the non-customized version to the common library.
- Redesign the software to make the module configurable so customization will be achieved by loading a configuration at run time.

One issue we ran into while implementing the CCB is that we did not plan properly for the hardware components and troubleshooting tools we needed to complete this work. As we were developing these modules we looked at them from a pure software perspective and realized that we needed some specialized hardware to verify its functionality (in addition to the obvious CPUs to execute the code). We were able to get these missing components from our telescopes, but since not everything was readily available, this caused schedule delays that we could have avoided. In addition, we discovered that in order to troubleshoot many of these components we had to have several tools in our labs (these existed at Gemini, but normally at the summit), so these were additional procurements we had to make while doing this work.

This work was completed in August 2016. Our CCB provides 19 common libraries and drivers to support all our needs. In addition, we produced a set of requirements for every component, and a comprehensive test plan that was used to verify and accept each CCB component as completed. These documents and procedures have been instrumental as we moved forward during the system upgrades stage. In many cases, while testing systems, we had to go back to the CCB to ensure the components were working as intended, and in a few cases, we discovered that we were missing corner cases. In these instances, new unit tests were developed and/or the test procedures were updated, and consequently new versions of the components were released.

TELESCOPE SYSTEMS UPGRADE

With the Application Development Environment and the Common Code Base completed, we started the process of upgrading the telescope systems during the second half of 2016.

A complete upgrade of each system consists of replacing direct VxWorks system calls with their OSI equivalents, converting the real-time operating system from VxWorks to RTEMS and ensuring that the CAPFAST schematics can be maintained with TDCT. Any system with minimal hardware connections will be considered for conversion to a Linux-based Soft IOC.

Strategy for the Upgrades

The first questions we faced were in terms of what order to follow for upgrading the systems. Where do we start? Do we upgrade the easy systems first? The hard ones? And what makes a system easy or hard?

Our approach was to first tabulate the systems by their complexity. For this, we defined complexity as the number of Common Code Base components every system needed in order to operate. This was somewhat arbitrary, but it worked well in that it gave us an indication of the magnitude of the required modifications needed to port each system from the legacy code base to the new environment. A system with just a few dependencies to CCB generally meant that it required few changes to be supported in the new environment, and the testing would be focused mostly on those interfaces. A system with a variety of dependencies normally implied a system that has many hardware interfaces (therefore drivers) and/or it required significant refactoring to use consolidated libraries.

Then, with this information, we defined a complexity threshold and separated the systems into two categories: "Easy" for systems with dependency number lower than the threshold value, and "Hard" for systems with a higher dependency value. We decided to start by upgrading the simplest system we had. This would be useful, as it would allow us to hone in on the upgrade process and to ensure that our strategy to define good test plans was robust; This would, in turn, allows us to focus on solving the complex technical issues that would come up in other more challenging systems, without spending time on commonplace issues present on every system. It would also be good for the team morale, as it would give everyone confidence in the work being done by putting upgraded systems in operations right off the bat and without major difficulties.

The next challenge was to figure out the strategy for testing and verifying the upgraded systems. The telescope and all its components are used every night, and therefore access for testing and verification is limited. Gemini has scheduled maintenance shutdowns periods, but those are normally once a year, and therefore restricting testing to just those windows would impose serious risks to the project schedule.

In order to address this, our initial approach was to develop simulators for each system. This would allow us to upgrade each legacy system, test it in the laboratory, and confirm that it would work prior to its re-commission in the telescope. The intent was to reduce the amount of time required on the telescopes for testing and verification.

This did not work for many reasons. First, one of our assumptions was that we had system design documents that were current, and therefore it was possible to use these to extract system requirements and test plans to develop the simulators. Unfortunately, this was not the case. Several systems have evolved over the years, but their design documents have not been kept up to date, so we didn't have useful documentation for many of our systems. Reverse engineering the systems to extract these requirements from the source code is possible, but would take a significant amount of effort.

We also discovered that having all the hardware required for system testing in a lab environment would be prohibitive. We concluded that with the schedule and available resources we had, the aspects that we would be able to simulate in the lab were very limited.

With this information, we abandoned the idea of producing systems simulators, and instead focused our effort to develop comprehensive test plans for each system, to perform testing at the telescopes.

Developing System Test Plans

As the original software design documents were mostly lacking or incomplete, we decided to derive these test plans from requirements that we extracted from use cases, as illustrated in Fig. 6:

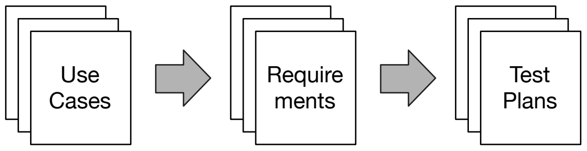

Figure 6: System Test Plan Derivation.

For each system, the first task was to identify, categorize and describe every interaction in the system. We used standard UML Use Cases [11] for this, and we interviewed key stakeholders for every system. These interviews included not only direct users of the systems, but also engineers and technicians doing maintenance and troubleshooting support, and other software engineers that had provided support to these systems in the past.

This activity provided the additional benefit of increasing key stakeholder awareness and buy-in. The use of Google Docs [12] to write these documents was very useful to get quick feedback and review. We also had to spend some time providing basic training to the interviewees, to ensure they would follow the UML diagrams and for them to understand the scope of this exercise. As we expected, there were instances where users were asking for new capabilities and improvements to the system in question – so we had to be clear about the goals we were trying to achieve in this stage. One interesting area was of the user interfaces – as users were thoroughly reviewing how they operate the systems, it became obvious in many cases that some user interfaces could be simplified and cleaned up. We decided *not* to do that in the project, to retain our schedule and focus the effort on the upgrade itself, but nevertheless to collect this information for future operational improvements.

With the use case documents completed, the next step was to extract system requirements. For this, we documented requirements in tables, with unique identifiers. We also held a review with key stakeholders to ensure the requirements were complete.

Based on the use case documents, we realized that we were covering *functional* requirements well, but *performance* requirements were somewhat lacking. For this, we worked with hardware engineers, experts for each system, to identify these and come up with their corresponding test plans.

With the requirements document in place, the final step was to put together a test plan for each system. The test plan contains a test that verifies every requirement in the system and it is used *every* time we plan to verify a new version of the software.

Upgrading the Systems

With systems test plans in place and a strategy defined, we went ahead and followed the procedure to port the legacy systems.

As dictated by our strategy, we started with the simplest systems first – we selected the Gemini weather system (GWS) and our calibration system (GCAL).

The upgrades of these systems went smoothly. As the GWS did not have any direct hardware connections, we replaced that system with a soft IOC running Linux. GCAL on the other hand controls motors and lamps, so we moved it to RTEMS on a VME CPU.

The approach to upgrade the systems at night was agreed with Operations to follow these rules:

1. The system should have passed its test plan at the telescope during the day.
2. We will have two people on the summit at night so that in case something goes wrong, we can revert the system back to its previous state, to avoid losing night time.
3. Scheduling of these upgrades will be coordinated with science operations to minimize the impact on some high-priority science programs.

In April 2017, both systems were successfully commissioned at Cerro Pachón. This gave us confidence and we started to work on the next set of systems, the Interlock System (GIS) and the Primary Control System (PCS).

Initial tests with GIS were promising and we decided to deploy it at night. Unfortunately, we found strange issues with the system, that forced us to revert back to the legacy system. This proved that the plan to have engineering effort at night to provide support in these cases was a good thing.

The GIS team went back to the lab, and further testing demonstrated that the CPU we were using was not behaving correctly.

In the meantime, the PCS was having significant problems in the porting – as it was using specific system calls in VxWorks that weren't directly available in RTEMS. Contract work with Osprey Distributed Control Systems [13] provided us with a module to replace these calls and we were able to solve this issue.

Unfortunately, the PCS presented additional problems especially with performance during initial testing. To make things worse, as we were approaching winter in Chile, access to the summit was very limited, and as such our possibilities to continue troubleshooting the system were reduced.

In July 2017, we had a planned shutdown window at Gemini North, for seven weeks. We took advantage of this shutdown to test and troubleshoot our systems there. We were able to solve the issues with the PCS, GIS, and also ported our Cassegrain Rotator Unit (CRCS). This showed that uninterrupted time to troubleshoot issues was the most effective way of resolving these issues.

With this work completed, back at Gemini South the team focused on upgrading the Mount Control System (MCS), which was completed in September 2017.

At the time of this writing, we were able to put in operations the GWS, GCAL, PCS, GIS, CRCS and MCS at Gemini South, and the GWS, PCS, GIS and CRCS at Gemini North. We are looking for a window to upgrade GCAL and MCS at Gemini North. We are confident we can have this completed in October 2017.

It was clear from this experience that although lab testing is important, telescope testing is critical and dedicated access to it for troubleshooting is key. Unfortunately, as the telescopes are in regular operations, this is a challenge that we will continue to face. Future telescope shutdowns are windows that we are looking at for additional opportunities for extensive testing. Planning in advance for those is critical. Considerations to winter weather conditions also need to factored in the schedule.

PROJECT MANAGEMENT LESSONS

The Real Time Upgrade Project has been a multi-year project with a distributed team working towards an ambitious goal.

The project has been organized using the PRINCE2 methodology [14], using a product-based planning, with clearly defined stages and work packages. This has been particularly useful for this project. In fact, this project was put on hold for almost a year to allow engineers to complete a different higher priority project in 2015. Doing that was simple, as there were clear boundaries in the project plan when stopping work would make sense.

In any distributed team work, especially when dealing with remote locations and different time zones, communication is key to achieve good performance, coordinate activities, and align goals. During the project we have extensively used a messaging client platform[15] that allows everyone in the team to join a room to discuss relevant topics to the project. This has expedited collaboration and simplified communication that otherwise would have occurred via email or phone, and most likely would have taken days to achieve results.

In addition, documentation has been produced using Google Docs [12], which has proven to be very effective for multiple people to work together on the same document. We have discovered some limitations – perhaps the most important is that once a document becomes too complex (some of our test plans/requirements documents were 100+ pages long, with tables and figures), editing these documents is painfully slow. Breaking those documents down in smaller pieces would be preferable.

As the systems upgrade stage started, testing moved forward and new issues were appearing, we needed a way to prioritize tasks and ensure the team would remain focused. We started to use an Agile process for planning the project activities, based on Scrum [16], using one week sprints. This has proved to be very effective, allowing us to better visualize our workload and make more timely decisions on work to be completed. We use software tools that support this model [17]. Gemini has used this methodology extensively in other high-level software development projects, but this was the first time we used it in a real-time software project. On a weekly basis, a review of the main accomplishments of the week is done, detailed plans are formed for the upcoming week, and new tasks that have been discovered are also discussed and prioritized for the future. In retrospect, we should have started to use a system like this from the very beginning.

Another important aspect of this project is coordination with operational activities. We are in constant communication with science and engineering operations, in order to keep them informed of our progress and also to plan for any activities that will require telescope time, be it for testing or when new systems are ready for operations.

In addition, communication with end users has been critical. Before introducing any new change to operations, we have produced training material for both end users and technical staff that will be impacted by these changes. As we have multiple sites (Hawaii, Chile and in each case, staff at the base facilities and the summits), we have repeated these talks to ensure all staff are well aware of the changes.

The strategy used in the project to start with the simplest systems first was very useful. Perhaps the only problem is that we became too optimistic after the first two systems were released to operations, making us believe that the rest of the systems would be equally simple. This wasn't the case. However, and as intended, starting with simple things allowed us to polish up our internal process, iron out coordination details with operations and helped us to boost the project morale.

Finally, as in any development project, we had to be very strict to avoid scope creep. As we move forward in the upgrades, we continue finding areas that are interesting to improve in our systems. We take note of those, register them as future operational improvements, and move on. Once this project is closed, we will review those items with operations so future projects can address them. Having an issue tracking system in place is particularly useful for this kind of housekeeping.

FUTURE WORK

At the time of this writing we are completing the commissioning of the first set of systems and we are starting the process to develop the system test plans for the next set. We expect to complete all systems by April 2018.

In addition, we need to update the documents that define our new software standards for real time and control systems development. We expect this work will be completed by end of Q2/2018.

Once this project is closed, Gemini plans to continue with other obsolescence management projects. In particular, we are looking at ways to upgrade our reflective memory system, upgrade obsolete CPUs from our VME-based systems and define a strategy to upgrade our secondary mirror control system, among others.

CONCLUSIONS

This project is providing a systematic upgrade to the Gemini Telescope real-time software, based on a thorough analysis of both the software and the development processes used to maintain it. The approach has allowed us to successfully upgrade operational systems that are used regularly, without major impact on our scientific operations.

With the experience gained so far, we are confident that by the end of this project we will have a unified software development environment, updated standards and a collection of telescope systems that can be maintained and kept current efficiently by the observatory staff.

In addition, this project has served as a concrete example that dealing with obsolete components (software and hardware) is possible. As we did not have a systematic process to upgrade these systems since they were commissioned more than fifteen years ago, we are now catching up. We expect to continue doing planned maintenance and upgrades of these systems in the future, to keep these systems up to date for years to come and avoid getting into the same situation in the future.

REFERENCES

[1] K. Gillies, and S. Walker, "The Design of the Gemini Observatory Control System," ADASS V, Tucson, AZ, 1996.
[2] Experimental Physics and Industrial Control System, http://www.aps.anl.gov/epics/index.php/.
[3] B. Goodrich, A. Johnson, and C. Boyer, "Standard Controller," Document ICD-13, Gemini 8M Telescope Project.
[4] W. Rambold, P. Gigoux, C. Urrutia, A. Ebbers, P. Taylor, M. Rippa, R. Rojas, T. Cumming, "Upgrade and Standardization of Real-Time Software for Telescope Systems at the Gemini

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MODPL03

Telescopes", SPIE Astronomical Telescopes + Instrumentation 2014, Montreal, Canada.

[5] J. Johnson, K. Tsubota, and J. Mader, "KECK Telescope Control System Upgrade Project Status," in *Proc. ICALEPCS'13*, San Francisco, CA, USA, 2013.

[6] M. T. Heron, T. Cobb, R. Mercado, N. Rees, I. Uzun, and K. Wilkinson, "Evolution of Control Systems Standards on the Diamond Synchrotron Light Source," in *Proc. ICALEPCS'13*, San Francisco, CA, USA, 2013.

[7] E. Matias, R. Berg, T. Johnson, R. Tanner, T. Wilson, G. Wright, and H. Zhang, "CLS: A Fully Open Source Control System," in *Proc. ICALEPCS'07*, Knoxville, TN, USA, 2007.

[8] Wind River VxWorks RTOS, `http://www.windriver.com/products/vxworks/`.

[9] Observatory Sciences, `http://www.observatorysciences.co.uk/`.

[10] M. Kramer, "EPICS: Porting iocCore to Multiple Operating Systems," in *Proc. ICALEPCS'99*, Trieste, Italy, 1999.

[11] Unified Modeling Language (UML), `http://www.uml.org/`.

[12] Google Docs, `https://www.google.com/docs/about/`.

[13] Osprey Distributed Control Systems, `http://ospreydcs.com/`.

[14] PRINCE2 Methodology, `https://www.axelos.com/best-practice-solutions/prince2`

[15] Atlassian Hipchat, `https://www.atlassian.com/software/hipchat`

[16] Scrum Process, `https://www.scrumalliance.org/`.

[17] Atlassian JIRA Software, `https://www.atlassian.com/software/jira`