ASEPS-0 Testbed Interferometer control system

Brad Hines

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, California 91109, USA

ABSTRACT

The ASEPS-0 Testbed Interferometer control system design presents new challenges as compared to previous generation instruments. Increased instrument complexity due to narrow-angle astrometric observing techniques, longer baselines, and increased user expectations due to computer technology advances all contribute to the size of the effort required to bring the instrument on-line.

This paper discusses the design objectives for the computer systems and software for the instrument and how the architecture selected was driven by the objectives and by the resources available; one of the major design objectives was to come up with an architecture that keeps software, network, and communications issues separate from scientific and subsystem implementation issues, making the most effective use of both scientifically-oriented and computer-oriented developers. The paper then presents the architecture that has been implemented in detail.

1. INSTRUMENT OVERVIEW

The ASEPS-0 Testbed Interferometer, formerly known as the TOPS 0 Testbed Interferometer, is designed especially to make high-precision astrometric measurements of stars separated by narrow angles of 10 arcminutes or less. The overall architecture and design of the instrument has been presented in a paper by Colavita et al at this conference.[1] There are a number of properties of this instrument that lead to novel requirements on the control system for the instrument as compared to previous-generation instruments such as the Mark III,[2] including the presence of an camera for initial acquisition, the delivery of two stellar beams from one siderostat, infrared operation, and the long (100 meter) baselines.

2. PROGRAMMATIC INFLUENCES ON THE CONTROL SYSTEM ARCHITECTURE

The complexity of the instrument and its user interface is such that, in its final form, the instrument control software is expected to consist of about 100,000 lines of real code (not including comments and whitespace). The traditional approach to software engineering a project of this magnitude would be to hire staff specifically to do the software development for the project.

However, the ASEPS-0 Testbed Interferometer project does not have budgetary resources to support a dedicated staff of software specialists. This means that people who are not software specialists are required to perform a number of the software development tasks that are required. Unfortunately, the increasing software and hardware complexity of the current generation of interferometers is such that it is no longer a safe management practice to develop individual instrument subsystems in isolation and then worry about the software integration later. At the 100,000 line level, a software/integration effort requires software specialists in order to ensure success. Without the prudent application of accepted computer science concepts and sound software engineering techniques, the complexity of the software can easily grow to overwhelm the manpower available.

It is essential, then, to make use of a computer and software architecture that leverages the efforts of the software specialist(s) that are available so that the people who are not software specialists can develop subsystems in parallel that will integrate readily into the system as a whole. In particular, it is desirable to minimize the complexity of the software from the point of view of the people who are not software specialists. Where complexity is necessary, it should be moved into the realm of the software specialist whenever possible. In general, the main goal to be achieved by the selected software and computer architecture is the minimization of complexity and programmatic risk.

3. CONTROL SYSTEM ARCHITECTURE OVERVIEW

3.1 Philosophy

The overall approach to the ASEPS-0 Testbed Interferometer control system architecture design was to borrow from previous

work wherever possible, minimize complexity through judicious use of tested computer science tools where necessary, and effect a clear partitioning of the control system into manageable and natural subsystems. As a result, the ASEPS-0 software architecture borrows substantially from the design philosophy of the Mark III and from the actual hardware, code, and communications techniques developed for an earlier project at JPL to develop an optical delay line for the BOA interferometer.[3,4] For this reason, a VMEbus-based system using 680X0 CPUs running the VxWorks realtime operating system was selected.

The control system architecture and the overall instrument architecture are tightly integrated with one another, and the design of each affected the other. The control system architecture divides the instrument into functional subsystems such that each subsystem is a self-contained entity consisting of all the optics, electronics, computers, and software needed to be able to operate the subsystem. This subsystem as a whole is then the responsibility of a "subsystem developer" to implement and make work.

In order to provide the necessary computational horsepower, and to make each subsystem truly self-contained, a multiprocessing architecture was selected, with each subsystem being allocated one or more of its own 680X0 CPUs; this way, subsystems do not have to share CPUs with other subsystems and problems can be more readily isolated.

The various subsystems all fit into a master command, control, and communications software framework that is provided to each subsystem "for free" by the overall control system architecture (i.e., the subsystem developers are able to access these functions without concern for the details of their implementation). This overall framework is the responsibility of the software specialist, who hides the complexity of the implementation of these functions so that the subsystem developers can concentrate on getting their subsystems working.

The ASEPS-0 Testbed Interferometer is a physically distributed instrument. Such a layout lends itself naturally to a distributed computer architecture; however, distributed computing is by its nature a complex undertaking and it was determined that it was desirable to avoid this complexity where possible. For this reason, the hardware portion of the control system architecture was selected so as to synthesize an apparently lumped computer system even though the elements of that computer system are physically separated. As far as the interferometer's software can tell, it is as if all the computers and input/output electronics are located in a single VMEbus cardcage, although that is not physically the case.

3.2 Properties of individual subsystems

Each subsystem consists of one or more 680X0 single-board computers communicating with optical and electronic hardware through a combination of off-the-shelf and custom VMEbus I/O cards. Each subsystem fits entirely within a single VMEbus cardcage. Each CPU runs a number of periodically scheduled real-time tasks, such as servos or state machines. CPU time is partitioned among the tasks based on their priority and specified run frequency. In addition, each CPU may also run asynchronous "background" tasks that perform functions such as waiting for new commands or performing lengthy or non-deterministic calculations.

Each subsystem has a number of operating modes. For example, the star tracker subsystem can be in acquire mode when it is first searching for a new star, track mode when it is actually servoing the stellar image to the instrument boresight, or off mode when it is idle. In each mode, the subsystem runs a finite state machine (FSM), cycling among various states as conditions change. So, by our terminology, a *state* is a property of a subsystem that it can change by itself as it performs some task such as acquiring a star, while the *mode* of a subsystem, which cannot be changed by the subsystem itself under normal conditions, selects which of the available state machines the subsystem should run.

3.3 Things subsystems need from their environment

One of the properties of the software architecture is that the subsystem developers should not have to worry about how their subsystems hook to the outside world; they should only have to worry about making their subsystems work. In order to accomplish this goal, a friendly environment must be provided to each subsystem that can take care of the majority of this detail. This environment must run on the subsystem's CPU(s) alongside the subsystem's own code, but it must be independent of and not interfere with the operation of the subsystem itself.

The environment must provide the following services: command input, data recording to a host, status reporting to a host or sequencing computer, periodic scheduling of the tasks needed to make the subsystem function, data transfer to and from other

subsystems, and control of and by other subsystems. A successful implementation of this environment reduces the use of these services to simple function calls from the subsystem code itself whenever a service is desired. This will be discussed in more detail in section 5.

4. HARDWARE ARCHITECTURE

The control system hardware architecture as discussed is shown in Figure 1. The following sections discuss the various elements of this architecture. As discussed in section 3.1 above, the goal of the control system hardware architecture was to synthesize one large virtual VMEbus cardcage from the point of view of the control system software. In order to do this, it is necessary to identify the functions which are common to all subsystems and then transparently redistribute these functions among multiple cardcages.

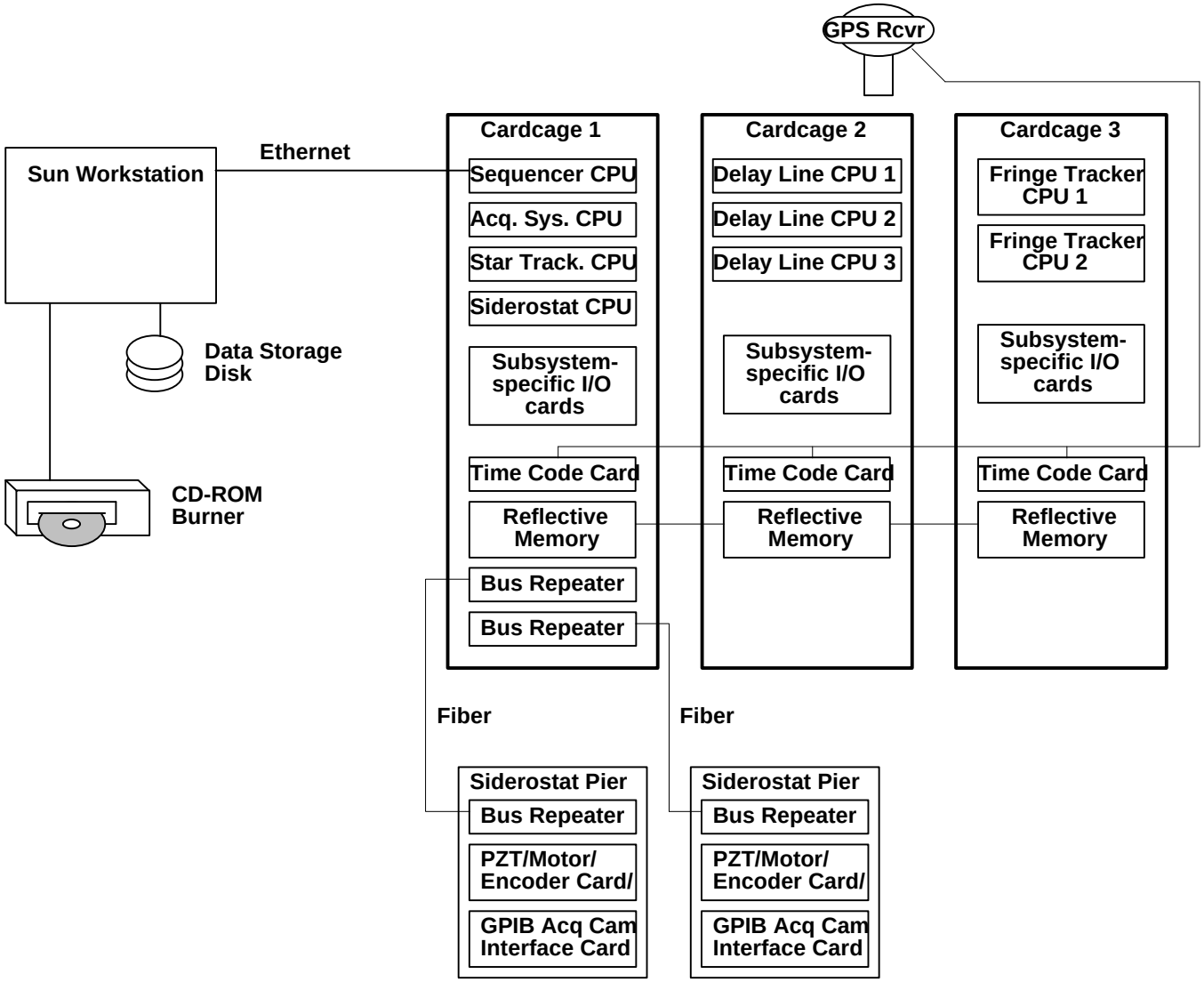

Figure 1. Control system hardware architecture

4.1 Timing and synchronization

The first of these functions is timing and synchronization. The ASEPS-0 Testbed Interferometer makes use of a GPS satellite receiver system which then can distribute IRIG-B timecode via coax to one or more VMEbus IRIG time code reader cards. When located in separate cardcages, these time code reader cards maintain synchronization with one another to the few-microsecond level. We therefore place one IRIG time code reader card in each cardcage. This card is then programmed to

periodically interrupt one CPU in each cardcage, providing deterministic timing for the interferometer's subsystems. In response to the interrupt from the time code reader card, this CPU, referred to as a "dispatch CPU," can then turn around and interrupt any other CPUs in the cardcage as needed via a so-called mailbox interrupt. The dispatch CPU is not dedicated solely to the dispatch function; rather, it is part of one of the interferometer's subsystems that has been assigned the additional responsibility of dispatching interrupts to other subsystems. By this scheme, then, every CPU in the interferometer receives deterministically timed interrupts on a regular basis, regardless of which cardcage it is in or how many other CPUs or subsystems are in its cardcage.

4.2 Communications

A second function that is common to all subsystems is communication with the outside world, including communication of status, data, and commands. There is an abundance of communications techniques that could be used, but the fastest and most reliable form of communication available is simply a block of memory that is shared among all the CPUs in the interferometer with predefined regions of memory being used for predefined purposes.

Even with shared memory, communications techniques can be quite complex. For example, the VxWorks operating system itself is available with an optional module which provides such high-level concepts as pipes, message queues, and semaphores via a shared VMEbus memory. However, this level of sophistication is not needed for this application and carries significant overhead and leads to a lack of determinism, because the services provided by the operating system are appropriate to a dynamically changing environment, inasmuch as the number and size of message queues, pipes, or other communications constructs that will be needed cannot be known at the time the operating system is compiled.

In the case of the interferometer, however, we know exactly how many subsystems there will be and what communications mechanisms will be needed between each, allowing us to *statically* define, at compile time, how the shared memory will be used. With this restriction in place, shared memory becomes an extremely efficient and rather simple communications medium.

Unfortunately, the number of VMEbus cards needed to implement the interferometer exceeds the number that will fit in a single VMEbus cardcage (we have three cardcages), so it is impossible for all the CPUs in the interferometer to actually share the same VMEbus memory. However, an off-the-shelf product known as a "reflective memory" solves this problem. Identical reflective memory cards reside in each of the three cardcages, and the three cards are connected by a cable. A reflective memory card behaves just like a normal memory card, in that CPUs can read and write to the memory on the card just like they would to any other VMEbus memory, but, in addition, whenever a CPU writes to a location in its local reflective memory, the reflective memory card immediately broadcasts the updated information to the other two reflective memory cards, which update their own memories to be the same as the first card's. In this way, information is transparently propagated from cardcage to cardcage. As far as the CPUs in each cardcage can tell, the information in the reflective memory may as well have come from a local CPU as one in a different cardcage. In effect, although each VMEbus cardcage has its own separate and distinct address space, the three cardcages share the portion of the address space containing the reflective memory. This works out well in terms of efficiency of communications and available bus bandwidth, because it means that VMEbus transactions with the shared memory, which are of interest to all CPUs, are transmitted to all cardcages, while VMEbus transactions that are specific to a particular subsystem, such as reading the value of an A/D converter, remain confined to a single cardcage.

4.3 Remote control and sensing

The last remaining hurdle for the control system hardware architecture is how to deal with the large physical separation of the components of the interferometer to which it must interface. With an instrument baseline of 100 meters, the control system must be capable of controlling motors and reading cameras at the instrument's siderostat piers that are physically quite far from the CPUs that are controlling them.

One option would be to just have long cable runs from the hardware at the pier to the interface electronics in the main interferometry lab. The nature of the connections needed make this impossible, however (e.g., a GPIB link from the acquisition camera to its controller), so the VMEbus interface electronics that control the pier hardware must be located at each siderostat pier.

It would be desirable, then, to extend the reflective memory out to each siderostat pier so that the remote VMEbus cardcages can be part of the single "virtual cardcage" that the software sees. However, the reflective memory cannot operate reliably over such

a distance, and problems with ground loops and the worry of lightning strikes make this an undesirable solution. In addition, this solution would require at least one additional CPU in each remote cardcage. In fact, since the siderostat piers contain elements of the star tracker, siderostat, and acquisition subsystems, it would take three CPUs in the remote cardcage to remain in conformance with the desired architecture already set forth.

Instead, a scheme using a so-called bus extender was used. The "bus link," as we refer to it, consists of two VMEbus cards with a pair of optical fibers connecting them. One of the two bus-link cards goes into the "local" cardcage inside the interferometry lab, the other goes into remote cardcage out at the siderostat pier, and the optical fiber connects the two. The only other cards in the remote cardcage are the slave I/O cards which read the acquisition camera and which control the motors and piezoelectrics for the siderostat and star tracker. When the siderostat CPU, for example, wishes to move the siderostat motors, it accesses the motor controller card as if it were in the local cardcage. The local half of the bus link responds to the access as if it were the motor controller card. It then causes the remote half of the bus link to actually access the motor controller card as commanded and return the result to the local half of the bus link, which then reports the result to the siderostat CPU as if it itself were the motor controller card.

This system is transparent, then, to the local CPU, and the notion of one large virtual cardcage is preserved. In addition, since optical fiber is the communications mechanism, the 100-meter baseline length is not an issue. Also, optical fiber is now the only connection between the remote siderostat piers and the interferometry lab, eliminating any potential problems with ground loops or ground differentials caused by lightning strikes.

4.4 External communications

Communications from the interferometer control system to the outside world, including user interface command and control, data recording, and status updates, are carried out via Sun-style Remote Procedure Calls over an Ethernet connection between the Sun and the control system's sequencer CPU, which handles sequencing of all the other subsystems in addition to carrying out communications with the outside world.

5. SOFTWARE ARCHITECTURE

5.1 Overview

The overall software architecture consists of five subsystems (siderostat, acquisition system, star tracker, delay line, and fringe tracker) communicating with each other and with the Sequencing computer through shared memory. Data and status information are relayed to a host Sun workstation using Sun-style Remote Procedure Calls. On the host end, the incoming data is recorded to disk, while the incoming status information is routed to a Graphical User Interface for display. An operator can also use the GUI to send commands to the VME systems; these commands are sent via RPC to the Sequencing CPU, which carries them out.

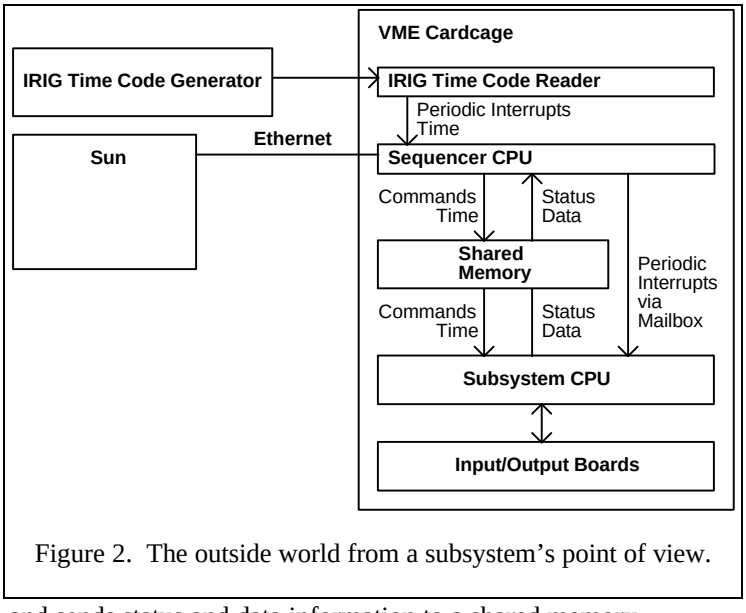

Figure 2. The outside world from a subsystem's point of view.

From the point of view of a subsystem (or a subsystem developer), the outside world can be viewed as shown in Figure 2. The subsystem's CPU interfaces to its associated I/O boards over the VMEbus to control the hardware it needs to control, and it receives commands from and sends status and data information to a shared memory. Synchronization with the rest of the interferometer is maintained via mailbox interrupts from the sequencer CPU, which receives interrupts from the IRIG time code reader card which is synchronized to a GPS receiver.

In general, the main goal of the software architecture is to give to subsystem developer as many of the services he needs as possible "for free," so that the developer's main concern can be getting his code working and not getting data and commands in and out of his subsystem.

5.2 Services provided to subsystems

There are six basic services that are common to the various subsystems that the architecture provides: command input, data recording, status reporting, periodic task scheduling, input of subsystem-specific data (sometimes referred to as handoffs), and control of other subsystems. The paragraphs that follow explain more about what these are.

Command input, data recording, and status reporting are all available via a simple function call mechanism. Whenever a new command arrives for a given subsystem, the software environment calls a function in the subsystem developer's software. A skeleton for this function is provided as part of the environment; by adding meat to the skeleton, the developer gains the capability to process commands. Status reporting and data recording are similar. When data or status information is required, the subsystem environment calls a skeleton function to collect the necessary information; the developer adds the necessary code to this skeleton to provide meaningful information. Alternatively, the subsystem can be configured to provide status and data proactively (whenever it determines a need) rather than in reaction to calls from the subsystem environment.

Periodic task scheduling is provided in only the coarsest form in the VxWorks operating system, so a periodic task scheduling library was developed to provide this facility to the subsystem developer. The subsystem developer can add or delete tasks as he wishes, specifying their execution rates and priorities. All management of the tasks and error reporting and recovery is taken care of by the environment.

The implementation of data handoffs ("handoff" is a term we use to describe a chunk of information passed between two subsystems) and the ability to control other subsystems is discussed in detail in the next section. The basic idea is that affecting a change in another subsystem or retrieving information left by another subsystem is as simple as a single subroutine call. As far as the subsystem developer can tell, the subroutine runs entirely locally to his own CPU.

5.3 Public and private interfaces

One of the abstractions that is central to the control system software architecture is the concept of public and private interfaces to each subsystem. My use of the terms public and private here is slightly different from their use in the context of object-oriented programming languages. In the context of a programming language, a public interface is that set of functions and variables which is accessible by any part of the software, whereas the private interface is only accessible to code that provides the actual implementation of the object being defined, thus isolating the details of the implementation of an interface from the interface itself.

While other details are different from the programming language meaning of the terms, the concept of implementation hiding does carry through to the public and private interfaces used in the control system software architecture. The public interface to a subsystem is the set of functions that can be requested by any other subsystem in the interferometer. The public interface is simply a set of C-language functions that abstract away the underlying shared-memory communications mechanism.

For example, the public interface to the siderostat subsystem includes a function sidTargetSet which is used to set the right ascension and declination coordinates at which the siderostat should point. This function is available to any CPU in the interferometer that needs to set the siderostat target. The implementation of this function would be in the file sidPublic.c. In the case of sidTargetSet, which is an "on-command" type of handoff, the function must do two things: it must place the new target coordinates into the designated area of the shared memory, and it must place a command into the specified siderostat's command queue in shared memory telling it to

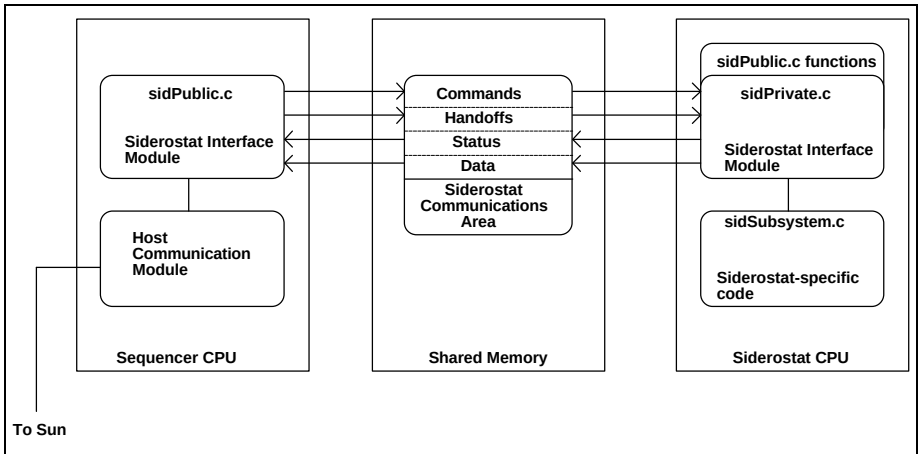

Figure 3. Public/Private interface

retrieve the new target coordinates from the shared memory. The lines of communication used to do this are shown in the left half of Figure 3.

The siderostat's private interface, in the file sidPrivate.c, contains the routines necessary to retrieve the new target information from shared memory; in this case, it would be a function called sidTargetGet. The command input functions of the subsystem environment for the siderostat will notice the new command in the shared memory command queue and will execute the new command by calling a routine specified by the subsystem developer. This routine should then call sidTargetGet to retrieve the new siderostat target position handoff from the shared memory and then do whatever else is necessary to put the new target into effect. These lines of communication are shown in the right half of Figure 3.

Thus, the public interface to a subsystem is the set of interfaces which the subsystem provides to the other CPUs in the control system. The private interface for the subsystem is the set of functions used to complement the public interface by retrieving from shared memory the information that is placed there by CPUs which call the public functions. In this way, the actual interface to the shared memory is abstracted away both for the CPU that calls the public function and for the CPU whose public function is being called. It would be entirely possible to change the communication medium from shared memory to Ethernet, for example, without changing a single line of the subsystem developer's code.

In practice, then, each subsystem has a xxxPublic.c file containing the code that makes up its public interface. The public interfaces for all the subsystems are linked together into a single object file which contains the public interfaces for all subsystems. This object file is then loaded onto every CPU in the control system, thus giving each CPU access to all the public functions of all the CPUs in the control system. The advantage of this architecture is that there are no limitations on communications routes imposed by the architecture of the communications interface, allowing for maximum flexibility as development and integration progress and as previously unforeseen communications requirements arise.

5.4 How the environment provides these things

There are several software modules that are linked in with each subsystem's code and are resident on each subsystem's CPU that provide the services that have been described. For example, in the case of the siderostat, these include modules called sidPrivate and genPrivate which handle I/O needs, schedLib which handles task scheduling, and a set of modules with names like acqPublic which are the public interface to other subsystems. The siderostat software is based in a module with a name like sidSubsystem. The siderostat code does not all need to be located in this file, but, as part of the software framework, a skeleton sidSubsystem.c is provided. All interfaces with the outside world, therefore, are through this module, but the code placed into the sidSubsystem file is free to call routines in any other modules that the developer wishes to write.

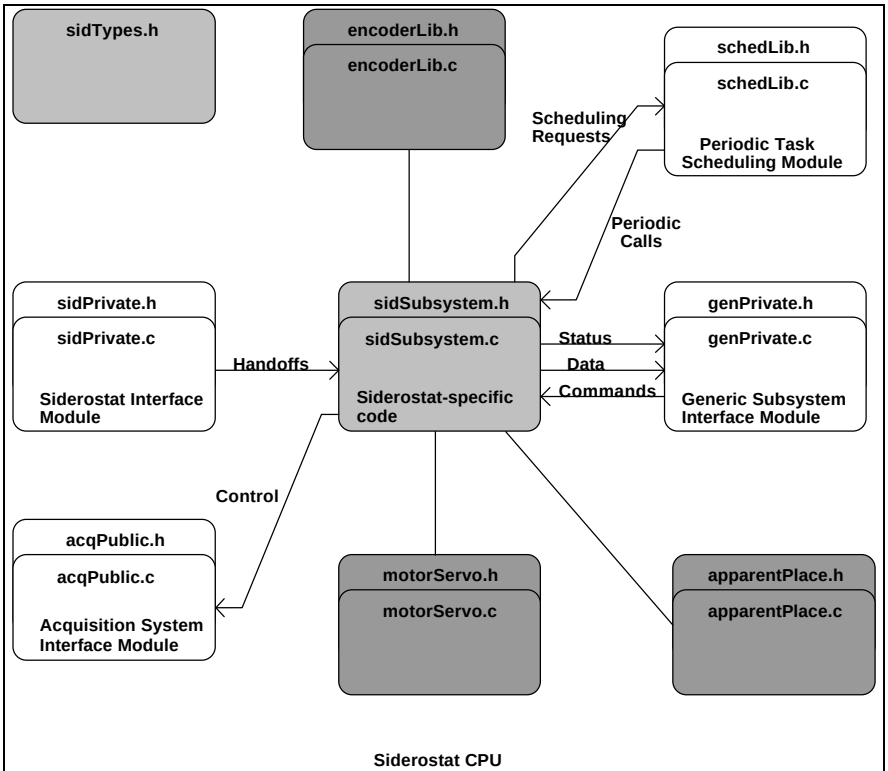

Figure 4. Subsystem software architecture (siderostat example).

This is illustrated in Figure 4. Modules which the subsystem developer would write from scratch are hatched and skeletons which the subsystem developer would modify are shaded, while the unshaded modules are those provided as part of the subsystem environment and should not be changed by the subsystem developer.

In the diagram, all interface functions are routed through sidSubsystem.c, but sidSubsystem.c accomplishes much of its work by calling routines that are located in motorServo.c, apparentPlace.c, and encoderLib.c. In addition to filling in sidSubsystem.c, the subsystem developer adds data type definitions to sidTypes.h for any siderostat-specific structures which will be imported or exported as status, data, or handoffs. This file can then be included by any modules that wish to interact with the siderostat subsystem.

The bulk of the services provided by the subsystem environment are provided in the genPrivate module; this contains the code for command input, status reporting and data recording, and hooks into the periodic task scheduling system. The periodic task scheduling system itself is contained in the schedLib module; this module handles the creation, deletion, and deterministic prioritized scheduling of the real-time tasks needed by the subsystem. Handoffs, or the receiving of data from other subsystems, are handled by subroutines in the sidPrivate module, which can be called from anywhere in the subsystem code as needed.

5.5 Instrument configuration

For maximum flexibility during development and integration, many parameters concerning instrument configuration are not compiled into the code, but are rather determined at run-time from special configuration files that we refer to as "INI files." These files are patterned after the WIN.INI and SYSTEM.INI files used in Microsoft Windows, and will be familiar to most Windows users. This technique allows for quick turnaround when making changes to the system and for an easy way for subsystems to keep records of data that is important to them but may vary with time, such as a pointing model for the siderostat.

5.6 Sequencing and Communications

Sequencing of the instrument and communications via Ethernet to a host computer where the instrument operator sits are handled by the sequencer CPU. The sequencer is similar to the other interferometer subsystems, but has some substantial differences that merit that it be considered separately. For example, the sequencer CPU does not have a public interface, because it is the sequencer's job to control the other subsystems, not vice-versa. Communications and instrument sequencing are only loosely coupled on the sequencer CPU, so they will be discussed individually.

Like the other subsystems, the sequencer CPU operates by moving through the states of a finite state machine. The sequencer has more than one FSM that it can use; which FSM it uses depends on which mode the sequencer is in. For example, one could have separate FSMs for astrometric and imaging observing modes, if that were desired.

Current plans call for the sequencer to have a single Observing mode, a Manual mode, and an Off mode. In Manual mode, the sequencer does not exercise much intelligence, but rather it simply acts as a relay in ferrying low-level commands sent by the host computer to individual subsystems. This is extremely handy during debugging and integration and during maintenance procedures such as removing the siderostat mirrors for recoating. An operator can sit at the console of the host computer and use the Graphical User Interface to view and control the details of individual subsystems, almost as if the sequencer weren't there.

In Observing mode, on the other hand, the host computer provides little guidance other than a list of stars to observe, and the sequencer CPU handles all the details of commanding the individual subsystems through the normal observation sequence. This process consists of two parts. One part of the process is to monitor the error and status streams coming from each subsystem and maintain a model of the state of the interferometer as a whole so that the sequencer will know when to move from state to state and will be able to detect and handle anomalous conditions such as temporary cloud cover or hardware failure. The other part of the process involves controlling of the interferometer's subsystems.

The major part of the control task involves causing the various subsystems to switch into their various modes at the appropriate points during an observation sequence. For example, once the acquisition system has acquired a star such that it is centered on its camera face (ideally falling onto the star tracker's detector), the sequencer commands the star tracker to enter its star acquisition mode. The other part of the control task involves sending the necessary handoffs and commands to the individual subsystems, such as sending a siderostat target handoff to the siderostats so that they will point at the star to be observed.

In addition to the standard Observing mode state machine, the sequencer is designed to be flexible, so that other state machines can be developed and implemented during the integration process. For example, one of the early tasks in integrating the interferometer is to demonstrate integration of the siderostat and the acquisition system, and there will be a special sequencer mode implemented to perform just this integration, and it will be left in the system as a special diagnostic mode for the

sequencer once integration is complete. Thus the sequencer will have the capability to operate the instrument as a whole or to operate portions of it as desired for testing.

One aspect of the interferometer that places some unique requirements on the sequencer is the dual-star operation of the instrument. A single observation typically consists of pointing to, tracking, and acquiring fringes on both a bright primary star and a dim secondary star. For most of the subsystems, a given operation on the primary star must complete before the corresponding operation on the secondary star can complete. For example, the star tracker must be tracking the primary star before it can begin acquisition of the secondary star. One effect of this is that it creates an opportunity for parallel execution of certain sequences. For example, once the star tracker has completed primary star acquisition, two things can happen: the star tracker can begin secondary star acquisition and the fringe tracker can begin primary fringe acquisition.

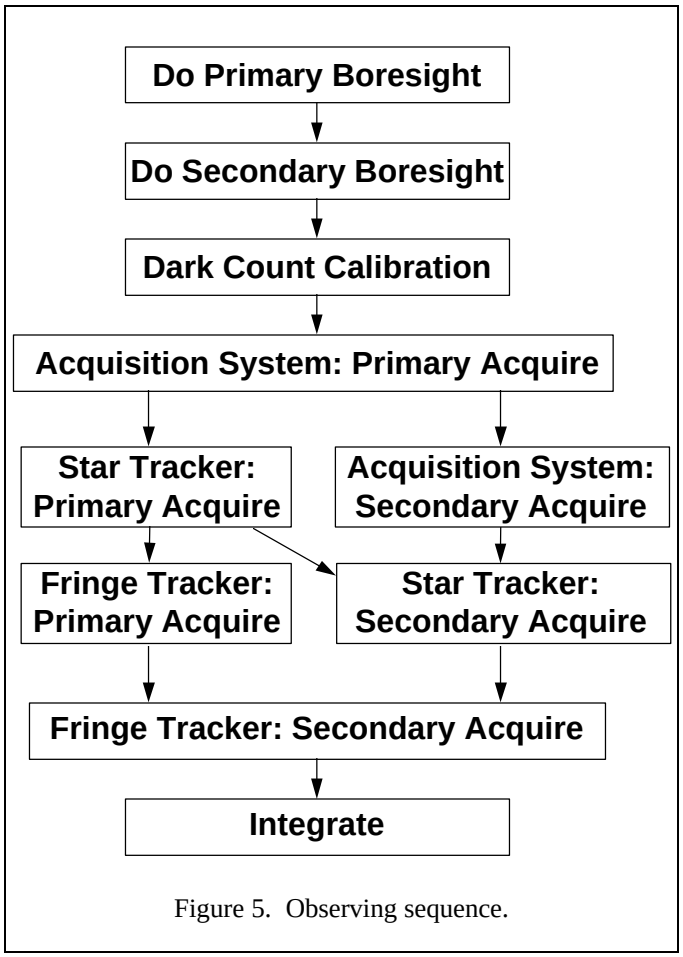

Figure 5. Observing sequence.

Another way of viewing this "opportunity" is to note that, for maximum observing efficiency, the instrument must take advantage of the inherent parallelism and perform operations simultaneously when possible. When seen in that light, taking advantage of the parallelism is more of a requirement than an opportunity. For this reason, the sequencer has a multi-threading capability. That is, the sequencer's FSM can be in more than one state at once. This is illustrated in a coarse way in Figure 5, which depicts a simplified observing sequence for a single star pair. From this, you can see, for example, that once the acquisition system completes its acquisition of the primary star, the star tracker can commence its primary acquire operation while the acquisition system begins its secondary acquire operation, so the sequencer can be in both the star tracker primary acquire state and the acquisition system secondary acquire state at the same time. When the acquisition of the primary star is complete, the sequencer splits the current thread of execution through the state machine into two threads of execution, and the two threads move on independently from that point until they are scheduled to rendezvous again.

In the example in the diagram, the thread that goes to the star tracker primary acquire state splits again into two threads when complete. One of the these moves on to the fringe tracker primary acquire state, while the other moves to the star tracker secondary acquire state, where it rendezvous with the thread leaving the acquisition system secondary acquire state. By rendezvous, I mean that the two threads merge, so that only one thread continues on to the star tracker secondary acquire state.

The issue of how to implement the rendezvous is interesting; the thread that arrives at the rendezvous first must apparently wait for its mate in a kind of no-man's land, with no well-defined state, before continuing on to the star tracker secondary acquire state. In our system, what actually happens is that the first thread to reach a rendezvous is killed, and the second thread to reach the rendezvous is the one that actually continues on through the state machine. This is accomplished by associating a *rendezvous count* with each state in the state machine. When a thread prepares to enter a new state, it decrements the rendezvous count for that state. If the rendezvous count has reached zero, the thread enters the new state and resets the rendezvous count to its original value. If the rendezvous count for the state has not reached zero, the thread terminates. Most of the states in the sequencer FSM, then, have rendezvous counts of one, but states where threads join, such as the star tracker secondary acquire state, have a rendezvous count of two. While the sequencer does not have any states where more than two threads merge at once, the rendezvous count mechanism is capable of managing rendezvous of arbitrary numbers of threads at a single state and can easily be used for more complicated state machines.

The VxWorks realtime operating system which runs on the VMEbus computers provides native support for Unix-style networking, so Sun-style Remote Procedure Calls[5] were a natural choice for all communications between the sequencer and the

host computer, which is a Sun workstation. RPCs do exact some overhead compared to lower-level communications techniques such as sockets, but much of this overhead can be avoided by using care in the development of the RPC mechanisms. The big advantage of RPCs is that they are simpler and more robust than socket-level programming. (With a lot of work, a socket-level programmer can write code that is robust in the face of network anomalies, but with RPCs, someone else has already done it for you.) Specifically, RPCs make use of a facility whereby the specific socket number over which communications will be carried out is not specified until runtime, making for better reliability in the event of system crashes, which are not infrequent during debugging and can lead to "stuck sockets" which can result in a lot of lost time during development.

There are two details about our use of RPCs that are probably interesting. One is that all our RPC communications use the RPC "opaque" data type, thus avoiding the overhead of the RPC XDR (eXternal Data Representation) layer. This is referring to the fact that RPC is a protocol designed for communications between arbitrary types of computers, with any arbitrary word length or byte ordering. In order to allow all types of computers to communicate, a canonical representation of data must be defined for all transactions across the network. This representation is the eXternal Data Representation. All remote procedure calls pass all their arguments and results through an XDR layer on the way into and out of the machine in order to ensure compatibility on both ends of the link. However, in a case such as ours, where the host computer is a Sun Sparcstation and the embedded computers are Motorola 680X0 machines, the native data representations of the two machines are very similar (word length and byte order is the same for both machines, although there are some minor differences in the way structures are packed), so the XDR layer is not essential. By defining our own do-nothing XDR procedure on both ends of the link, we avoid the overhead of XDR and can achieve substantially improved performance with minimal added complexity.

The other detail of interest is our use of RPC procedures. RPC defines the notion of a "protocol." For example, there are four protocols set up between the sequencer and the host computer, one for the transmission of commands from the host to the sequencer, and one each for the transmission of error, status, and data information form the sequencer to the host. An RPC protocol may support multiple "procedures." For example, we could define one RPC procedure for each type of command we wish to send to the sequencer. However, in keeping with strategy that led us to choose opaque data transmission, we use only a single RPC procedure within each protocol. For example, the command protocol contains a single procedure called RPC_PROCESS_COMMAND. The argument to the RPC call, which we assemble ourselves, is a structure which includes information on what the command to be executed actually is. This approach has no real disadvantage in terms of complexity; it is about equally complex to implement a set of RPC procedures as it is to interpret an incoming structure and take action depending on its contents. There is a big advantage, however, in maintainability of the code by people who are not experienced network programmers. Since the RPC layer is generic and static, new commands can be added to the interferometer without having to modify the network communications layer.

As the communications relay for the entire realtime computer system, one of the sequencer's main roles is simply to relay all status, data, and error information placed in shared memory by the other subsystems onto the host computer. In addition, the sequencer sends reports of its own status and activities. The host computer can then decide which information to display for the operator, which information to record as data, etc.

The other main function the sequencer performs is the processing of commands from the host. During startup, the sequencer spawns an RPC server process as an asynchronous background task. This process is idle until a command arrives from the host. When a command arrives, the server process wakes up, interprets the command, processes it (by inserting a command into one of the other subsystems' command queues in shared memory, in the case of a command for another subsystem, or by changing the state of some variable on the sequencer, such as the sequencer mode, for commands that are intended specifically for the sequencer), and then returns to its idle state.

5.7 Real-time control system software summary

Figure 6 is a fairly complete diagram illustrating the interactions among all the interferometer subsystems. This diagram is a fairly accurate representation of a subset of the interferometer consisting of just the siderostat, acquisition system, and star tracker subsystems together with the sequencer. Only the modules residing on the sequencer and siderostat CPUs are shown; the acquisition system and star tracker CPUs are not shown, although the modules used for communications between the siderostat and sequencer CPUs and the acquisition system and star tracker CPUs are shown.

As in Figure 4, modules which the subsystem developer would write from scratch are hatched and skeletons which the subsystem developer would modify are shaded, while the unshaded modules are those provided as part of the subsystem

environment and should not be changed by the subsystem developer. This drawing appears complex because it illustrates all the things that the subsystem environment provides to the subsystem developer (the white boxes) without requiring him to write any code.

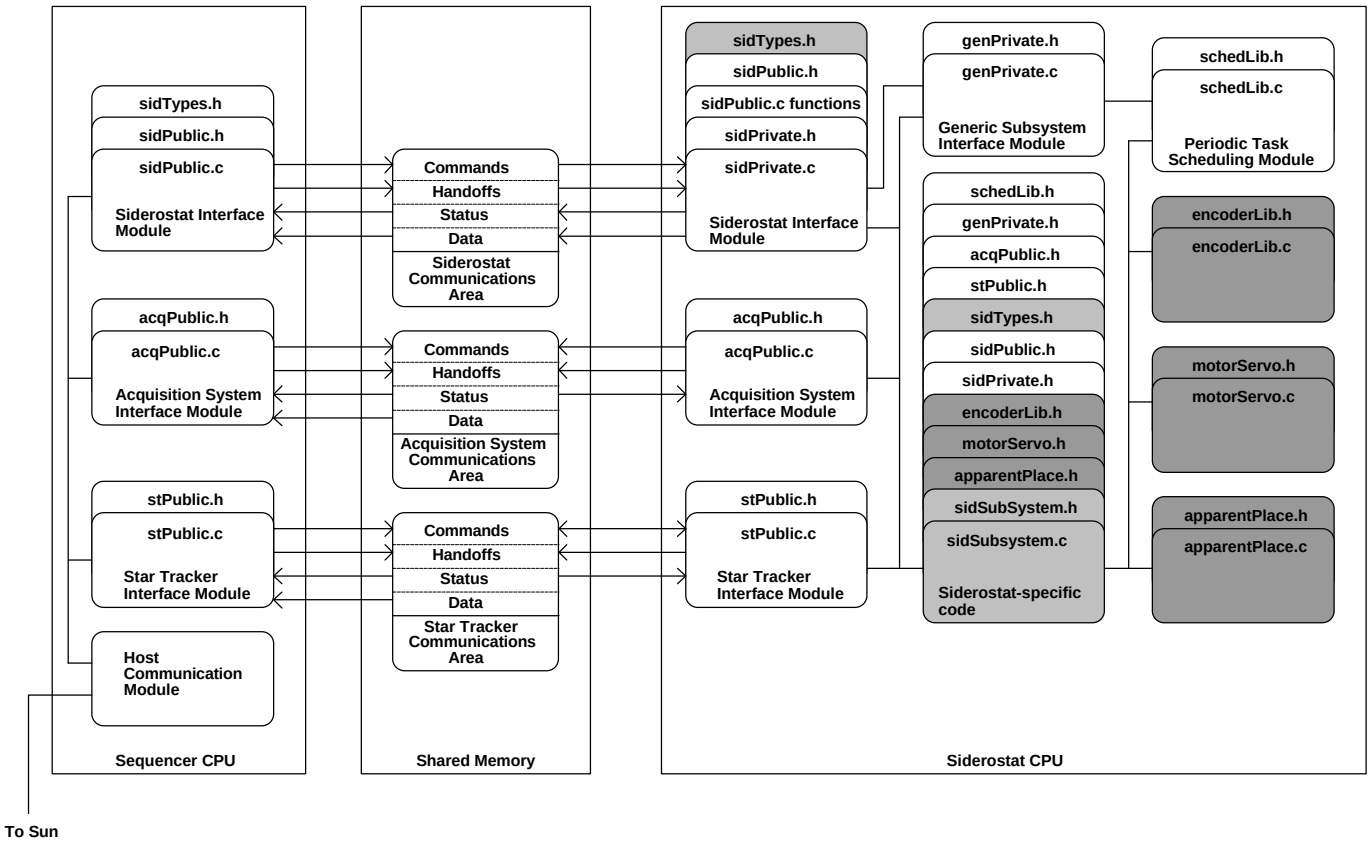

Figure 6. Software architecture detail.

5.8 Host software: user interface and data recording

All data, status, and error information from the realtime computers eventually makes its way to the host computer, where RPC routines respond to the incoming data. The operator starts the host-side software by running a single program; this program quickly splits itself into five independent processes. Three of these are the RPC server processes for incoming data, status, and error information. A fourth process has the job of relaying commands to the sequencer when requested by the operator, and the fifth process manages the user interface and data recording.

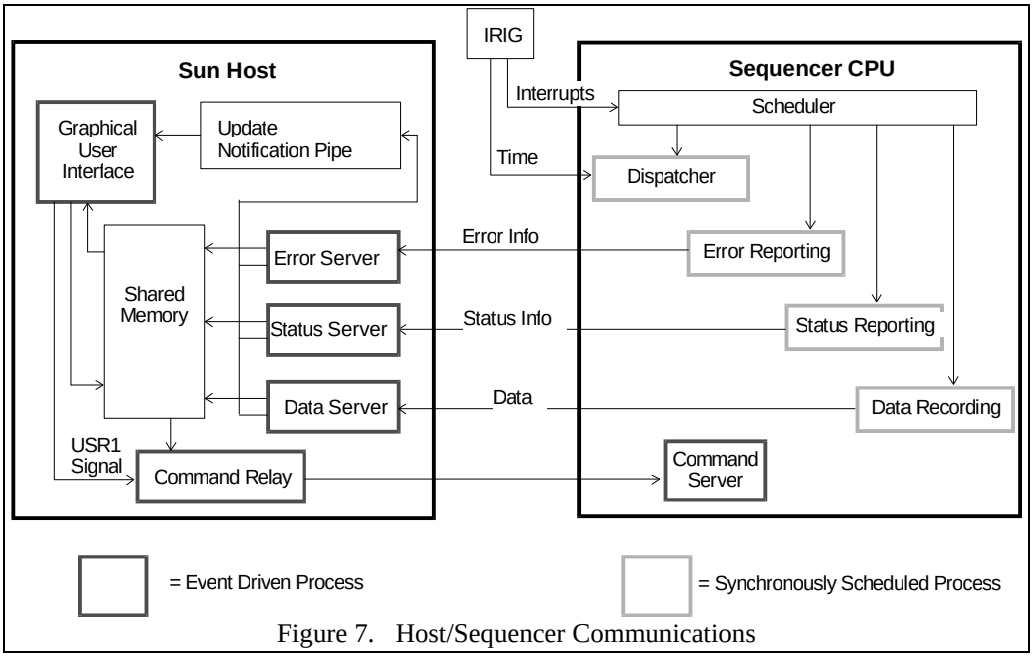

Figure 7. Host/Sequencer Communications

The five processes all communicate with one another through shared memory on the Sun (there are operating-system routines for allocating blocks of shared memory).

The three RPC server processes, shown in Figure 7, normally sit idle until an RPC call from the sequencer occurs. At this point, the appropriate server is awakened by the operating system. It processes the incoming RPC call by placing the newly arrived information into a predetermined location in the shared memory, and then it notifies the user interface/data recording process to process the new information. It does this by placing a brief message into a pipe,[6] which I refer to as the Update Notification Pipe, specifying what sort of information was just placed into the shared memory.

The user interface/data recording process normally sits idle, waiting for input from the operator or for new information to arrive in its Update Notification Pipe. When information arrives in the pipe, the user interface/data recording process wakes up and retrieves the newly updated information from the shared memory. If the new information is data to be recorded, then the process appends the new data to a disk file containing the night's observing data. If the information is new status or error information, then the process updates any open windows on the screen that contain information that is obsoleted by the new information. Optionally the status or error information may also be recorded to the observing data file.

When the user provides command input to the user interface (by clicking a command button with the mouse), the user interface process assembles a command structure to pass to the sequencer and places it in shared memory. It then signals the command relay process, which retrieves the command structure from shared memory and sends it on to the sequencer via a RPC. Under normal circumstances, the user interface process itself is capable of sending the command on to the sequencer directly, but in abnormal situations, such as if the sequencer crashes during debugging, the process performing RPC to the sequencer can get hung. If the user interface process hangs in this manner, the entire Sun workstation console locks up. To prevent this from happening, a separate command relay process is used. If this process hangs, the operator still maintains full control of the console and can cleanly exit the program.

6. CURRENT STATUS AND CONCLUSIONS

At this writing, about 2/3 of the ASEPS-0 Testbed Interferometer control system software has been written (about 60,000 lines of code). About 90% of the code that runs on the real-time computers has been written, and all of the subsystems have demonstrated some basic functionality. Integration of the various subsystems with the sequencer has been tested and is working, and integration of the subsystems with one another will begin in the next month or so. Communications with the host computer and the user interface are working, and end-to-end communications and control have been demonstrated for the delay line subsystem, and other subsystems are currently being integrated into the user interface. In short, all the pieces that make up the software and hardware of the control system architecture have been tested and are functional, having been tested with at least one subsystem. The next major step involves integration of the pointing system, consisting of the acquisition system, siderostat, and star tracker.

7. ACKNOWLEDGEMENTS

The research described was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

8. REFERENCES

1. Colavita, M.M., Shao, M., Hines, B.E., Wallace, J.K., Gursel, Y., Beichman, C.A., Pan, X.P., Nakajima, T., and Kulkarni, S.R., "ASEPS-0 Testbed Interferometer," this conference, Kona, Hawaii, 1994.

2. Shao, M., Colavita, M.M., Hines, B.E., Staelin, D.H., Hutter, D.J., Johnston, K.J., Mozurkewich, D., Simon, R.S., Hershey, J.L., Hughes, J.A., and Kaplan, G.H., "The Mark III Stellar Interferometer," Astronomy and Astrophysics **193**, pp. 357-371, 1988.

3. Mozurkewich, D., Johnston, K.J., and Simon, R.S., "Big Optical Array", *Proc. SPIE*, 1237:120-127, Tucson, 1990.

4. M. Colavita, B.E. Hines, and M. Shao "A high-speed optical delay line for stellar interferometry," High Resolution Imaging by Interferometry II, ESO, Garching, Germany, 1991.

5. Sun Microsystems, Network Programming Guide, SunOS 4.1 manual set, 1990.

6. Sun Microsystems, SunOS Reference Manual, SunOS 4.1 manual set, 1990.