A control system framework for the Hobby-Eberly telescope

Ramsey, Jason, Drory, Niv, Bryant, Randy, Elliott, Linda, Fowler, James, et al.

Jason Ramsey, Niv Drory, Randy Bryant, Linda Elliott, James Fowler, Gary J. Hill, Martin Landriau, Ron Leck, Brian Vattiat, "A control system framework for the Hobby-Eberly telescope," Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy IV, 991346 (8 August 2016); doi: 10.1117/12.2232724

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2016, Edinburgh, United Kingdom

A control system framework for the Hobby-Eberly Telescope

Jason Ramsey[a], Niv Drory[a], Randy Bryant[b], Linda Elliott[a], James Fowler[b], Gary J. Hill[a], Martin Landriau[a], Ron Leck[a], and Brian Vattiat[a]

[a]McDonald Observatory, University of Texas at Austin, 2515 Speedway, C1402, Austin, TX, 78712-0259, USA
[b]Hobby-Eberly Telescope, University of Texas at Austin, USA

ABSTRACT

We present the development framework for the distributed control systems, scripting frontend, and monitoring facilities of the recently upgraded Hobby-Eberly Telescope (HET). A common flexible control and data acquisition layer in C++, with message passing implemented on top of ZeroMQ, wraps the final designs of each new hardware component including tracking, metrology, instrumentation and calibration equipment. A homogeneous command, response and event layer normalizes the diversity of the lower level software interfaces easing the development of the Telescope Control System (TCS). Applications developed in the framework easily interface to the new tracker and legacy instrumentation of the primary mirror, weather, dome, and tracker support structure. The framework facilitates testing, vetting, and characterization of the telescope and TCS. Examples of the real-time monitoring capabilities and the Python scripting methods of various telescope components yield insight into overall system performance. Lessons learned along the way, future refinements, and anticipated enhancements, are detailed.

Keywords: Hobby-Eberly Telescope, HET, Control Systems, Software, Computer Programming, Integration

1. INTRODUCTION

The HET has recently completed a massive upgrade[1,2] of a majority of its primary tracking,[3,4] metrology,[5] and spectroscopy[6] instrumentation in preparation for the Hobby-Eberly Telescope Dark Energy Experiment (HETDEX). Driving, and abstracting, this new hardware is a diverse, yet thin and accessible, set of middleware layers built of a software framework wrapping open source tools providing consistent interprocess communication, multithreading, timing, configuration, and logging. The variety of hardware components now deployed at HET requires the flexibility of a distributed system to manage and impose the desired state of the telescope, for any science driven synchronization points, i.e. exposures, metrological sampling, positioning feedback, etc. The upgrade project has just entered the instrument commissioning phase and system integration is nearing the point where it sufficiently supports science grade data acquisition.

The initial effort to wrangle the software for the Wide Field Upgrade (WFU) took a hardware-centric approach focusing on abstracting the variety of the new hardware at a fairly low level.[7] This abstraction was then exposed to a Python scripting layer, via automatically generated methods. This produced many interfaces that went unused, adding unnecessary clutter and confusion to the code base. At the time, much of the new hardware, and the related interconnect, was still being determined and refined so a natural focus on a generic control system architecture, with the flexibility to easily replace or reconfigure the hardware abstraction layer, persisted with little development of the business logic required to coordinate an astronomical observation.

In this subsequent effort, motivators for imposing some standardization on the software for the next generation of HET control systems include increased developer efficiency, greater adaptability to requirement changes at the business logic level, thorough documentation of the entry points to each control system, and a maintainable and transition-able code base. The reliance on third-party libraries is minimal, attempting to keep things as lean and portable as possible.

Further author information: Send correspondence to Jason R. Ramsey
Jason R. Ramsey: E-mail: ramsey@astro.as.utexas.edu

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 991346 · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2232724

2. CONFIGURATION MANAGEMENT

The distributed nature of TCS, the complexity of the hardware represented, the need to reconfigure components for testing, and cyclic nature of system integration and debugging require a flexible approach to configuration. The applications comprising TCS (subsystems), each accept a configuration file containing attribute-value pairs which may be overridden at runtime by modifying the vales of the configuration with command line options corresponding to the supported attributes. The flat configuration file may be extended to pull in additional configuration files producing a configuration tree. However, in that case, only the root node is overridable by the command line. The Little Template Library (LTL), documented at http://www.as.utexas.edu/~drory/ltl, provides the API leveraged to support subsystem configuration and command line parsing. LTL is also used throughout TCS for numerics as well as FITS I/O.

Subsystems of TCS communicate with each other over configurable TCP/IP routes. A service provides runtime name resolution of these routes. In the case where a configuration file is given as well, the order of precedence for the three layers of configuration, from least to greatest, is configuration file, name resolution, then command line.

3. INTERPROCESS COMMUNICATION

An initial goal of the framework is to provide a dependable and robust communication channel that is easily accessible from the application layer, with only a few lines of C++, Python, or Bash. The subsystems and their clients, communicate using JavaScript Object Notation (JSON) strings, augmented by an optional untyped binary data block, as either broadcast events or in a command-response relationship. These two independent channels of communication provide a filterable publish-subscribe event interface and a blocking or non-blocking Remote Procedure Call (RPC) interface. The contents of the message and event strings are a depth one set of attribute-value pairs, a subset being reserved for metadata. These are organized by a hierarchy of system, source, and key, in the case of broadcast events and by acknowledgement, response, done acknowledgment, and exception, in the case of messages and responses. Aside from the legacy applications, all HET processes in the Wide Field Upgrade use these message-reply and broadcast protocols. These protocols, the datatypes, and their interfaces are encapsulated in a set of C++ classes and a subset of them exposed through the scripting layer.

3.1 Message interface

A TCS subsystem defines its set of available services inheriting from, and extending as needed, the message handler* class and registers those with a receiver. Each supported operation on, or query of, TCS has a corresponding handler implementation. Each handler may be registered as either reentrant or non-reentrant. In the later case, the receiver ensures that each invocation has mutually exclusive access to the handler. Once the handlers are registered and the receiver listening, the handler methods are available to receive messages from remote senders. The messaging interface is implemented on top of ZeroMQ DEALER/ROUTER socket pairs, see http://zguide.zeromq.org/page:all for further documentation of the ZeroMQ infrastructure.

Handlers may be invoked in either a blocking or non-blocking manner, referred to as synchronously and asynchronously respectfully. Handler process methods are executed in threads fetched from a statically allocated thread pool to reduce startup overhead. When called synchronously, control is returned to the caller once the done acknowledgement is received. When called asynchronously, control is returned to the caller as soon as receipt of the message is acknowledged. Any responses to asynchronous calls, including the done acknowledgement are made available to the caller through a queueing system that aggregates all responses to asynchronous messages associated with the sender. A handler that attempts to respond with more than a final done acknowledgement when invoked synchronously generates an exception so there is an explicit contract between the caller of a handler that responds with a sequence of responses, that it be called only asynchronously. This exception is generated on the sender side, see Sec. 3.4. The transaction diagram in Table 3.1 summarizes the relationship between sender and receiver in the synchronous case. In the asynchronous case, responses are either dropped by the sender

*Various C++ classes are referenced in this document by the abstract functionality they encapsulate, to avoid reimplementing class documentation in this paper.

Table 1.

Client		Sender	Network	Receiver		Service
Message	→	Send	→			
			←	Acknowledgment		
					→	Process
Response	←	Send	←		←	Response

upon receipt, or fed into a queue that can be consumed by the client. Scenarios exist for both use cases of this non-blocking interface.

The relevant performance metrics for such a system are less tangible than for those projects which strive to handle large numbers of concurrent requests. The telescope system, in general, moves fairly slowly and, at any given time, there can be only a single target state for the entire system. This results in very few truly concurrent invocations of the same underlying process. That aside, one method of evaluating the performance of the messaging interface is to examine the relative throughput of a handler that implements a NOOP. We define this metric as the number of messages handled by a receiver in a second, *mps*. The times at which messages are sent, received, responded to, and when the responses are received (M_s, M_r, R_s, and R_r respectfully) are integrated over messages sent to derive the *mps* throughput. For N messages sent, we determine a throughput, for the full command-response transaction, by

$$mps = \frac{N}{\sum_{i=1}^{N}(R_{r_i} - C_{s_i})}.$$

Such tests are highly system and setup dependent. Between a sending and receiving pair, both executing on the same laptop, we realize $\approx 2k$ *mps*, see Appendix A for example timings from alternate test setups. In Fig. 1 we see a sustained average command received time of .1852 Milliseconds and average complete response time of .6380 Milliseconds. In practice, the round trip component of this metric is dominated by the communication with the underlying hardware and completion of the requested action. This may vary wildly, depending on the behavior imposed by the done acknowledgement synchronization point defined by the handler.

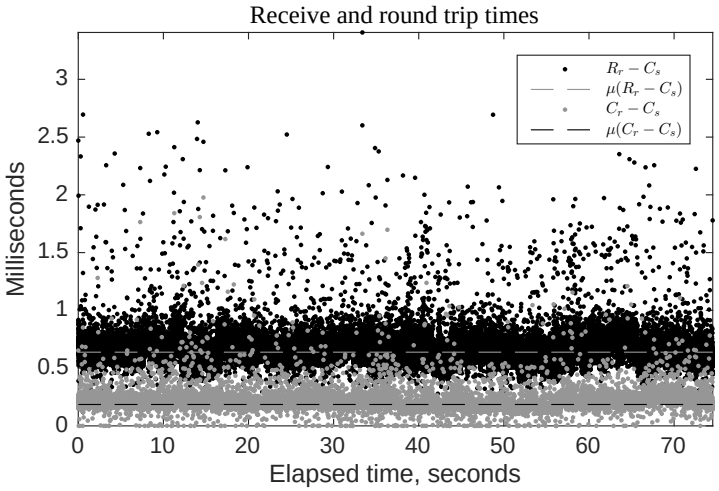

Figure 1. For each of 100k messages sent, the differences between send, receive and response times are shown along with the lines for the mean time-of-flight of the command and mean total transaction time.

3.2 Event interface

The event interface provides a publish-subscribe mechanism whereby a subsystem can broadcast a set of attribute-value pairs to all listening processes. Each subsystem is identified by a system name, and may instantiate multiple event sources on whatever contextual boundaries (global, class, method, singleton, etc) are appropriate for the particular application.

Each event broadcast is associated with a system, source, and key. The realtime event consumer side is implemented by instantiating an event sink, associating it with a set of event sources for which to listen, optionally applying independent filters over system, source, and key, then calling a method to wait on the next event that passes through the filter. The event sink must process the filtered events on an as received basis, calling a method when ready to handle the next event and blocking until received, or an optional timeout occurs.

Listing 1. An event broadcast in the scope of a class inheriting a C++ event source.

```
new_event( "my_key", fptime::now() )
        << make_pair( "attr1", 1 )
        << make_pair( "attr2", "two" )
        << make_pair( "attr3", 3.0f ) << ende;
```

From whatever necessary application scope, an event can be instantiated, parameterized with attribute-value pairs, then broadcast for system wide consumption including realtime plotting. The broadcast events are captured in persistent storage for post-analysis. This allows a subsystem developer to inject events wherever needed for debugging, profiling, and event driven coupling of subsystems.

3.3 The anatomy of the payload

Events and messages are derived from a base payload which provides thread-safe serialization and deserialization of the information over the wire. The payload provides access to the transmitted information via direct value lookup by attribute name and/or generalized, typeless, iteration over attributes. In the case of both events and messages, a suite of metadata accompanies each payload. This metadata provides timing information used to determine the timeliness of the communication, the timeliness of the information carried, and identifying information for the originator of the message or event.

The metadata of the payload is extended in the message, to include additional routing information. Each message response falls in to one of four categories; a message acknowledgement, an intermediate response to the original message, a final response the original message, and an exception. This tagging of responses is used to impose the implied state change on the (potentially blocking) sender and/or route the information to the appropriate queue. The message acknowledgment response is transparent to the caller, unless there is a timeout in sending the message, then an exception is thrown.

3.4 Exception migration

The framework provides the ability for called subsystems to respond with an exception. This is implemented such that any exception thrown within the scope of a handler's invocation is caught and relayed to the caller. Any exceptions that extend the base TCS exception have well defined types on both sides of the wire. All other exceptions are of a generic type. This design approach allows exceptions in third party lower level code to propagate back to the caller so that both sides of the transaction are able to adapt to the exception. The base class is extended to facilitate finer grained remote exception handling provided that both sides of the wire agree on the type. In general, compile time bindings at this level are avoided.

For asynchronous handler invocations, where responses may be one or many, any exception is queued in the same manner as non-exception responses. As the response queue is consumed on the sending side, each response must be queried by the caller for the exception flag in order to determine that an error occurred. This use case occurs sparsely and typically only for the transmission of image streams to a graphical user interface (GUI) and when scripting, see Sec. 6, non-blocking handler invocations.

3.5 Wiretapping

The messaging interface has been instrumented such that the traffic between any subsystems can be "tapped" for debugging or profiling purposes. The event interface is leveraged here to generate an event corresponding, less any binary data, to each message sent or received. Each subsystem exposes a handler that allows this functionality to be turned on and off. The feature is implemented such that the overhead of generating these tapping events is realized at all times, the broadcast of the event is the only thing that is runtime configurable. This approach allows for the feature to be turned on and off with no noticeable impact on the realtime system other than a manageable increase to the volume of information on the network.

4. MULTITHREADING

The framework presently imposes no constraints on how subsystems implement multiple threads but uses the TinyThread++ library internally, found at `http://tinythreadpp.bitsnbites.eu`, for a C++11 like threading mechanism. For TCS, the library is augmented with a thread worker class and a wait mechanism that attempts to handle spurious wakes, as well as return to the caller the time spent waiting. Initially, the GNU compilers in use were not C++11 aware enough to rely on the new threading interface. It is assumed, when so motivated, that transitioning to C++11 threads will not require a great deal of code modifications.

5. TIME

Time is sampled and represented in seconds since epoch, with fixed nanoseconds precision, throughout TCS. There are two timestamps associated with each message and event, applied by the framework layer. These are the time at which the information was sent, wire time, and the time that corresponds to the generation of the information in the payload, data time. Messages and events that have been deserialized have an additional receive time. Many pieces of information are aggregated, augmented, and then resent as broadcast events requiring the flexibility to adjust the data time while the wire time remains opaque to the application layer.

One example of this is the combining of positional information to derive a pointing on sky. The tracker and structure positions, information from two disjointly sampled realtime systems, are coupled in TCS to derive the telescope's current right ascension and declination. The data time placed on the subsequent RA/Dec event generated by TCS, prior to broadcast, carries that of the original tracker position, given that it is sampled at a much greater rate than the structure position, while the wire time of the new tertiary event represents its time of broadcast.

6. SCRIPTING

The primary supported scripting language for TCS is Python, with helper applications exposing a stateful scripting layer to Bash, or any other Linux shell. The Python bindings to subsystem handlers are direct, $1 \leftrightarrow 1$ with the C++ handler definitions, and are implemented at runtime by negotiation between the script and the subsystem(s) to which it connects. Each subsystem communicates the effective runtime API to the connecting Python object that then replicates the remote handlers as local methods. When called, the local methods resolve to a send of a message containing the parameters of the call to the connected receiver, and then the resulting handler invocation. The JSON string response associated with the final done acknowledgement is returned to the caller as a Python dictionary.

These handler bindings are all implemented in both a blocking and non-blocking manner. The non-blocking interfaces are callable by suffixing the relevant method name by `_async`. The call to the the non-blocking method replies with a message identifier with which the caller can wait for a response at a later synchronization point in the script. If an exception occurs on the remote end, it is held until the script waits on the initial message's identifier, at which point a Python exception is generated.

The state of the subsystem at the point where a done acknowledgement is sent implies a certain relationship between the caller and the subsystem. For instance, once a trajectory is loaded in to TCS it is activated, or sent on that trajectory, by a `go_next()` command. This command may be parameterized to request that not only the tracker be setup on trajectory, but that the dome, structure, guide probes, wave front sensors, etc., or any subset,

be setup as well, prior to returning. In such a case, the state implied by a successful return of the command is a function of the parameterization and success of the individual requests. Though currently handled through a GUI in practice, where state transitions are reflected in an event driven manner, and reaction timing handled by the user, scripts may choose to implement control flow and concurrency using this asynchronous invocation method.

7. DATA MANAGEMENT

The telescope state data and logging, apart from any imagery, spectra, or other instrument data, are represented as a timeseries by the events generated by the system. These are captured and recorded by one or more monitoring services. Though the monitor supports relay of the JSON documents to a MongoDB instance, the current practice is to store these in a nightly SQLite database with a minimal schema that exposes a NOSQL-like representation of the information supporting extension and removal of attributes from individual events without modification of the schema. The underlying schema is set when the first event, for any given system, source, and key, is received. Any deviation from that mapping, over the runtime of the monitoring process, is ignored.

Events are organized by identifier, unique to the monitoring process, over data time and context, along with their system, source, and key names and data time in one table. Another table stores the attributes and their values. These two tables are joined over event identifier to reproduce the event stream as a timeseries.

The SQLite interface, though cumbersome for live interaction, with a typical nightly database currently resolving to ≈ 7 gigabytes of information, is the preferred data representation at this point in commissioning. The nightly database files provide a portable and easily replicated data store that is immediately accessible to those familiar with SQL and having knowledge of the static schema. Realtime event consumers are currently assumed to acquire information via the C++ event interface.

8. FUTURE WORK

The HET currently struggles to manage and capitalize on the large volume of telescope state information gathered throughout the course of a night. Though a rich and fairly complete picture of each night's activities is captured, little exists in the way of user friendly tools for accessing this information in an efficient manner. Through the initial telescope commissioning phase, much of the analysis of system performance has been handled by laboriously sifting through information extracted in an ad hoc manner to common ASCII table format then massaged into the preferred environment of the analyst driving things. A well defined, standard, systematic and globally accepted approach to handling this information is needed. Current efforts to mitigate this include a Python API providing access to the event attribute-value pairs as Numpy arrays of timeseries and the application of reusable report generation tools that provide an analysis picture of a predefined subset of the night's information.

An initial requirement on the new control system was that the information being passed between subsystems be human readable. This was motivated by the user experiences with the legacy system's approach to interprocess communication which was primarily driven by global shared memory, so the information always remained in a binary C data structure form which hindered interprocess debugging. This was one of the primary factors in deciding to use JSON as a transport format but future efforts may investigate other mechanisms that avoid forcing the transmitted data structures through a string representation.

The broadcast mechanism employed by the framework relies on piecewise regular expression filtering of the system, source, and key. This architecture currently prevents the use of ZeroMQ's server side subscription filtering, which is implemented via a message prefix. This has little impact on the systems built of the framework aside from preventing the broadcast of large data volumes, as they would be filtered on the client side of each subscriber. This server side filtering functionality should be replicated at the level of the event sink to support the broadcast of events that include imagery and spectra.

9. CONCLUSION

The new control system framework has rapidly evolved to meet the challenges of the design, deployment, and commissioning of the next-generation of control and data acquisition systems at HET. It is successfully leveraged by each of the many distributed processes that make up the new TCS, without imposing idiomatic constraints. However, future refactoring efforts will, for the sake of maintainable, move to normalize the variety of design patterns that have been enabled. A generally applicable set of simple, robust, control system APIs has evolved as a side-effect of satisfying the requirements of the WFU and the HETDEX project.

APPENDIX A. MESSAGE TIMINGS

Round trip command-response times were profiled on a few combinations of systems. System A is a MacBook Pro (2015), B is a virtual machine hosted on system A, system C is a virtual machine hosted on hardware at the HET, system D is the LRS2 spectrograph readout computer at HET, and system E is the VIRUS spectrograph readout computer at HET. The specifics of the network interconnect between systems at HET are in flux and remain unknown for the purposes of this paper. The `ping` and `pong` applications used in this test are shown

Table 2. Average round trip times for 100k messages.

From	To	*mps*
A	A	2140.87
B	B	1567.48
C	D	939.72
D	E	1741.51

below, and provide minimal examples of the interfaces detailed in Sec. 3.

ping.cpp

```
#include <assert.h>
#include "messaging.h"

int main( int argc, char** argv )
{
    assert( argc == 3 );

    Sender s( argv[1], "ping" );

    for( int i = 0; i < atoi( argv[2] ); ++i )
    {
        Message* pong = s.send( "ping" );
        if( i == 0 )
            std::cout << "response_wire_time,response_receive_time" << std::endl;

        std::cout << pong->get_wire_time().as_string()
                  << ","
                  << pong->get_receive_time().as_string()
                  << std::endl;

        delete pong;
    }

    return 0;
}
```

pong.cpp

```
#include <assert.h>
#include <stdlib.h>
#include "messaging.h"

class pingHandler : public MessageHandler
{
public:
    pingHandler( int limit ) : limit_(limit), count_(0) {}
private:
    virtual void process( Message* m )
    {
        if( count_ == 0 )
            std::cout << "command_wire_time,command_recv_time" << std::endl;

        std::cout << m->get_wire_time().as_string()
                  << ","
                  << m->get_receive_time().as_string()
                  << std::endl;

        delete m;
        count_++;
        if( count_ == limit_ )
        {
            sleep( 1 );
            exit( 0 );
        }
    }

    int limit_;
    int count_;
};

int main( int argc, char** argv )
{
    assert( argc == 3 );

    // argv[1] == URL, argv[2] == message count
    Receiver r( argv[1], "pong" );
    r.registerHandler( "ping", new pingHandler( atoi( argv[2] ) ) );
    r.start();
    return 0;
}
```

ACKNOWLEDGMENTS

HETDEX is run by the University of Texas at Austin McDonald Observatory and Department of Astronomy with participation from the Ludwig-Maximilians-Universität Mnchen, (LMU) Max-Planck-Institut für Extraterrestriche-Physik (MPE), Leibniz-Institut für Astrophysik Potsdam (AIP), Texas A&M University (TAMU), Pennsylvania State University (PSU), Institut für Astrophysik Göttingen (IAG), University of Oxford, Max-Planck-Institut für Astrophysik (MPA) and The University of Tokyo. In addition to Institutional support, HETDEX is funded by the National Science Foundation (grant AST-0926815), the State of Texas, the US Air Force (AFRL FA9451-04- 2-0355), by the Texas Norman Hackerman Advanced Research Program under grants 003658-0005-2006 and 003658-0295-2007, and by generous support from private individuals and foundations. We thank the staffs of McDonald Observatory, HET, AIP, MPE, TAMU, IAG, Oxford University Department of

Physics, the University of Texas Center for Electromechanics, and the University of Arizona College of Optical Sciences for their contributions to the development of the HET WFU and VIRUS.

REFERENCES

[1] Hill, G. J., Drory, N., Good, J., Lee, H., Vattiat, B., Kriel, H., Bryant, R., Elliot, L., Landriau, M., Leck, R., Perry, D., Ramsey, J., Savage, R., Allen, R. D., Damm, G., DePoy, D. L., Fowler, J., Gebhardt, K., Haeuser, M., MacQueen, P., Marshall, J. L., Martin, J., Prochaska, T., Ramsey, L. W., Rheault, J.-P., Shetrone, M., Schroeder Mrozinski, E., Tuttle, S. E., Cornell, M. E., Booth, J., and Moreira, W., "Deployment of the Hobby-Eberly Telescope wide field upgrade," *Proc. SPIE* **9145**, 914506–914506–19 (2014).

[2] Hill, G., Drory, N., Good, J., Lee, H., Vattiat, B., Kriel, H., Ramsey, J., Randy Bryant, R., Elliot, L., Fowler, J., Landiau, M., Leck, R., Odewahn, S., Perry, D., Savage, R., Schroeder Mrozinski, E., Shetrone, M., Damm, G., Gebhardt, K., MacQueen, P., Martin, J., Armandroff, T., and Ramsey, L., "The Hobby-Eberly Telescope wide-field upgrade," *Proc. SPIE* **9906-5** (2016).

[3] Good, J., Booth, J., Cornell, M. E., Hill, G. J., Lee, H., Savage, R., Leck, R., Kriel, H., and Landriau, M., "Laboratory performance testing, installation, and commissioning of the wide field upgrade tracker for the Hobby-Eberly Telescope," *Proc. SPIE* **9145**, 914546–914546–11 (2014).

[4] Good, J. M., Hill, G. J., Landriau, M., Lee, H., Schroeder-Mrozinski, E., Martin, J., Kriel, H., Shetrone, M., Fowler, J., Savage, R., and Leck, R., "HET Wide Field Upgrade Tracker System Performance," *Proc. SPIE* **9906-167** (2016).

[5] Vattiat, B., Hill, G. J., Lee, H., Moreira, W., Drory, N., Ramsey, J., Elliot, L., Landriau, M., Perry, D. M., Savage, R., Kriel, H., Häuser, M., and Mangold, F., "Design, alignment, and deployment of the Hobby-Eberly Telescope prime focus instrument package," *Proc. SPIE* **9147**, 91474J–91474J–12 (2014).

[6] Hill, G., Tuttle, S., Vattiat, B., Lee, H., Drory, N., Kelz, A., Ramsey, J., DePoy, D., Marshall, J., Gebhardt, K., Chonis, T., Dalton, G., Farrow, D., Good, J., Haynes, D., Indahl, B., Jahn, T., Kriel, H., Montesano, F., Nicklas, H., Noyola, E., Prochaska, T., Allen, R., Blanc, G., Fabricius, M., Landriau, M., MacQueen, P., Roth, M., Savage, R., and Snigula, J., "VIRUS: first deployment of the massively replicated fiber integral field spectrograph for the upgraded Hobby-Eberly Telescope," *Proc. SPIE* **9908-54** (2016).

[7] Rafferty, T., Cornell, M. E., Taylor, III, C., and Moreira, W., "New Control System Software for the Hobby-Eberly Telescope," in [*Astronomical Data Analysis Software and Systems XX*], Evans, I. N., Accomazzi, A., Mink, D. J., and Rots, A. H., eds., *Astronomical Society of the Pacific Conference Series* **442**, 285 (July 2011).