Large Binocular Telescope

Instrument Data Interface Definition

CAN 481s016
Version c
Contents

1. Introduction ........................................................................................................................................ 3
2. Telescope Command Interface ......................................................................................................... 3
3. Web-Based Data Access ..................................................................................................................... 4
4. FACSUM ........................................................................................................................................... 8
5. Alarms............................................................................................................................................... 9
6. Telemetry .......................................................................................................................................... 12
7. Instrument Data Requirements ........................................................................................................ 14
8. Observing Tool Interfaces ............................................................................................................... 15
9. Abbreviations and Acronyms ......................................................................................................... 16
10. References / Links............................................................................................................................ 17

Appendix A – Sample INDI Service ..................................................................................................... 18
Appendix B – Simple Telemetry Interface Program ............................................................................ 22
Appendix C - Current TCS Data Dictionary Definition for Facility Instruments ................................. 27
1. Introduction

Instruments at the Large Binocular Telescope Observatory (LBTO) have their own control systems and GUIs to drive telescope observations. Using these control systems and GUIs (or scripts), instruments send commands to the telescope and read status information via the Telescope Control System (TCS) instrument interface.

To provide insight into observatory operations, instruments need to provide performance, alarming, and telemetry information to observatory software infrastructure. This allows cross-correlation of instrument data across the full spectrum of gathered observatory telemetry, provides a common alarming mechanism, and allows for integration of instrument status information into operator displays.

In 2015, the TCS shared-memory data dictionary was modified to include some instrument data placeholders. The LBC and LUCI facility instruments were updated to send data through the TCS interface to populate this instrument information for use by the facility summary GUI and for alarming. While this was temporarily sufficient, a better infrastructural concept is needed for general use. In 2017, the observatory will implement a web-based status information display. The design incorporates a new, more standard instrument data access mechanism that should serve both current and future instrument needs.

This document is a guideline to instrument teams for what should be published from instrument software to the observatory software infrastructure. This document will be a living document, as LBTO software personnel work with instrument developers to develop detailed instrument-specific data interfaces.

2. Telescope Command Interface

Instruments drive the telescope either via operator GUIs (generally discouraged), or via scripts. Instrument scripts are created either manually (discouraged), or are automatically generated via use of common observatory observation preparation tools. The LBT Observing Tool (OT) provides a common look and feel for defining observations, including aids for field orientation, guide star selection, specification of instrument configuration settings, and sequence definitions for offsets and exposures. The end result of a user session with the OT is a generated instrument observation script. Instrument software is responsible for script parsing (or GUI command execution), and then translation into telescope commands. More information about the OT is provided in paragraph 8.0.

The telescope interface is accomplished via low level use of TCS Instrument Interface (IIF) software. In addition to the control interface, status information is passed both directions.

The ICE Instrument Interface Control Document (http://abell.as.arizona.edu/~hill/xlbt/cgi/ican.cgi?481s013) describes:

- TCS fundamentals – instruments, focal stations, authorization, coordinate systems, modes
- the IIF programming interface in detail with examples
- details for installing Ice and some descriptions of its use
• the data dictionary – shared memory maintained by the TCS with public data available through the IIF for reading and writing

Historically, some instruments used the C-interface or C++ interface to the TCS. These interfaces are obsolete now. All future instruments should use the Ice interface (https://zeroc.com/products/ice)

3. Web-Based Data Access

LBTO is in the process of implementing a web-based server for viewing observatory-wide status. In the process of creating a viewer, the question of where the viewer should get its data arises. The TCS data dictionary (DD) is designed to be a real-time shared memory segment for use by the subsystems of the TCS software. It was a convenient place to read data from when the facility summary GUI (FACSUM, see next paragraph) was initially developed. However, it is generally undesirable to couple instrument data to TCS builds. Data required to provide web-based status does not require real-time updates. A much slower and simpler data access mechanism is required for overall observatory status updates.

LBTO’s initial approach to web-based data access will be to use INDI (Instrument Neutral Distributed Interface) protocol. INDI defines a simple protocol for “devices” to provide “properties” for use by INDI clients. The devices control everything about instrument properties. The properties are “self-describing” and are published by devices on-demand. Devices can be software drivers for actual hardware, or pure software services.

Because LBTO has a varied architecture – some instruments are more than 10 years old, others are just being built – it is desirable to implement a simple interface that will not impact older instruments significantly, and can be incorporated by instruments developed with more modern technologies (Python, Java). An INDI-compliant instrument can be written in any language.

The LBT Interferometer project, LBTI, has used INDI since the project’s inception and the project GUIs are now all web-based. Barring any unforeseen technical complications in implementing INDI compliant device drivers generally at LBTO, it is expected that INDI will be a supported instrument data access mechanism going forward.
The diagram below (from the Introduction to INDI document) illustrates how INDI drivers and services and the indiserver processes interact with INDI clients like GUIs or scripts. indiservers run on the same machine as the drivers they are controlling. Servers can be “chained” to communicate with servers on other machines.

Some instruments (LBTI, iLocater) are using INDI as a basis for internal control of the instrument, accessing LBTO infrastructure data, and GUIs. The LBTO web-based Status Server will be able to access those instruments directly through their indiserver. In the future (2018, forward), this will allow for real-time graphing interesting instrument data from Status Server web pages.
For instruments that do not use INDI for instrument control, there are simple options for making instrument data available to the observatory as INDI properties. The following diagram illustrates two options for instrument data. These two options will likely both be used by existing facility instruments.

One approach for instruments that are not using INDI internally would be to run an indiserver and indi driver on the instrument control machine. The indi driver would access all the state information of the instrument. For an instrument like MODS, this would be straightforward because most of its status is available in shared memory, available to external programs (like the isisCmd, command-line tool). The indi driver could periodically read the status and set INDI properties. The LBTO web host would chain its indiserver to the indiserver on the instrument host (see left side of figure above).
An instrument that does not have an external interface to its shared memory (like LBC, for example) could implement an “INDI translator” within the instrument application that would make instrument status data available to an indi driver.

Alternatively, if an instrument control host is also a web server, a very minimal INDI driver can be written to `wget` a file of status data from the instrument control host and parse it into INDI properties. The instrument control software would just require a thread that continually writes a file of instrument status data into the instrument’s web root (see right side of figure above). (This method of INDI service was implemented as a prototype to get TCS data from the “wda” application which provides input to the FACSUM and WEATHERSTATION GUIs.)

The INDI community is active. The LBTI INDI development team can provide both a C function library and C++ class library for implementing INDI drivers; a javascript library for implementing web GUI clients; and a python interface to INDI called pyindi for writing scripts. They also have java and C++ class libraries for writing java and Qt GUIs but LBTI has abandoned these in favor of using web pages for all GUIs. They will share any of this code, including their collection of drivers.

Appendix A provides a complete indi driver (a service driver, not a hardware interface) example using the LBTI C++ class library.
4. FACSUM

The facility summary GUI (FACSUM) displays an overall status of the LBTO systems. As such, it contains instrument information representing the status of the *authorized* instruments. LBC, LUCI, AO, and ARGOS all provide instrument state data for display on the FACSUM. For instance, LBC provides the current filter name, the sequence in the observing block, exposing state, guiding state; LUCI provides exposing state, overall status, mechanism states, camera name, etc. Instruments should be prepared to provide a robust set of state information and performance metrics available for reference.

In the past, instruments were able to set pre-defined TCS DD entries via IIF SetParameter/SetMultiParameter calls. “Pre-defined” means these definitions were built into the TCS subsystems. This use is now deprecated. Future instrument status information should be made available via INDI properties.
5. Alarms

There is currently a hardware instrument alarm system (IAS) that monitors up to 16 instrument relays and transmits alarm status to the programmable logic controller (PLC). Instruments must provide a set of contacts that are closed when no alarm is present and which open on loss of power or instrument alarm condition. The instruments actively connected to IAS in the upper-right treehouse are: LUCI 1, LUCI 2, and both AO systems. See the LBT Instrument Alarm System Specification

http://abell.as.arizona.edu/~hill/xlbt/lbts/601s101a.pdf

For software-driven alarms, LBTO has implemented an overall facility alarm GUI, using the Experimental Physics and Industrial Control System (EPICS) Alarm Handler

(http://www.aps.anl.gov/epics/extensions/alh/index.php,
http://wiki.lbto.org/bin/view/Software/AlarmHandlerMainPage)

The EPICS Alarm Handler reads EPICS channels and triggers alarms based on thresholds in a database. The channels are maintained via EPICS input/output controller processes. TCS subsystems and LBTO instruments can set EPICS channels in a variety of ways. LBTO has implemented an input/output controller (IOC) process that reads DD entries and writes EPICS channels. In that way, TCS or instrument subsystems can write DD entries to write channels. LBC and LUCI are writing TCS DD entries for temperatures and pressures.

EPICS channels can also be written directly from scripts. For example, there is a MODS script that periodically reads MODS temperatures and pressures and writes the corresponding EPICS channels. Computer status (disk, CPU, network) is also written this way by Information Technology scripts based on Zabbix monitoring.
Development of the LBTO Status Server (statserv.lbto.org) has provided an INDI driver that monitors groups of EPICS channels and rolls the status up into a web-based alarm viewer that mimics the EPICS Alarm Handler channel hierarchy. It is a display-only, quicklook type tool. See screenshot below.

Instrument INDI alarm properties should address signal names, priority (low, high), and threshold information (for auto priority detection). Instruments may provide alarm information as EPICS channels directly, or as INDI properties. In which case the observatory software infrastructure would provide translation services between INDI alarm properties to EPICS fields so that the alarm handler incorporates instrument alarms correctly.
Below is a sample screenshot from the EPICS Alarm Handler in use on-site at the LBTO. The EPICS implementation provides a rich set of important features for operators to organize and respond to alarms. The Status Server only provides a web-based, simple roll-up of the status for quicklook.
6. Telemetry

TCS telemetry data is written to Hierarchical Data Format (HDF5) archive files. See the Telemetry System Users Guide for details, http://abell.as.arizona.edu/~hill/xlbt/lbts/481s347

Any TCS telemetry item of the 6200 or so fields is available to graph (for at least a small time frame) from the visualization tools: http://telemetry.lbto.org/visualization/telem_vis.html

Currently, there is only a minor amount of instrument data recorded in the HDF telemetry – some of the status information that is displayed on the FACSUM is also routed to telemetry via the Data Dictionary Server (DDS) TCS subsystem.

Instrument software can take advantage of the telemetry collection library, provided by LBTO (http://telemetry.lbto.org/collection-doc/index.html) to write HDF5 telemetry files or write HDF5 files directly (given an interface definition that is not written yet). The LBTO telemetry collection library provides identical setup to the TCS telemetry for the HDF5 dataset, attributes, and the file characteristics (naming convention, rollover at 00:00UTC). The current LBTO collection library interface is not intuitive and a little unwieldy. But, it can be set up once and then just cut-and-pasted over and over with minimal changes for new telemetry streams. We would like to create a nicer interface in the future as well as a Python and Java interface.

See Appendix B for a simple, standalone C++ program that uses the telemetry collection library to write a telemetry stream using data from a file.
The advantage to writing telemetry data in the same format (dataset and file name) as the LBTO telemetry is that any instrument time series data can be graphed using the LBTO telemetry visualization tools. For example, the DIMM instrument uses the collection library to write seeing data to HDF5 telemetry streams and can be graphed, correlated with TCS data, with no effort using the telem_vis.html page:
7. **Instrument Data Requirements**

As discussed above, instruments should be prepared to provide a wide set of instrument monitoring and telemetry data. In the table below, “Monitoring” refers to data desired for the LBT StStus server and FACSUM; “Alarming” refers to data for the EPICS alarm handler; and “Telemetry” refers to data written to LBTO-compliant HDF5 archive files.

This information may be unique for each instrument, so the tools that use it must be flexible. It is the instrument team’s responsibility to create and coordinate a detailed list of data defining instrument states, metrics, alarms, and telemetry. As an illustration of possible data, see Appendix C which lists the data that LBC and LUCI are providing via the TCS data dictionary and the data that MODS has defined to be provided. A table providing general guidance is provided below.

<table>
<thead>
<tr>
<th>Data</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Monitoring</td>
</tr>
<tr>
<td>power state</td>
<td>✓</td>
</tr>
<tr>
<td>active state</td>
<td>✓</td>
</tr>
<tr>
<td>(idle, setup, flushing, integrating, readout...)</td>
<td></td>
</tr>
<tr>
<td>health indicators for each component</td>
<td>✓</td>
</tr>
<tr>
<td>setup state (mask, filter, cam, etc)</td>
<td>✓</td>
</tr>
<tr>
<td>detector state(s)</td>
<td>✓</td>
</tr>
<tr>
<td>shutter state(s)</td>
<td>✓</td>
</tr>
<tr>
<td>temperatures</td>
<td>✓</td>
</tr>
<tr>
<td>pressures</td>
<td>✓</td>
</tr>
<tr>
<td>quality statistics (FWHM, Strehl, etc)</td>
<td>✓</td>
</tr>
<tr>
<td>calibration state</td>
<td>✓</td>
</tr>
<tr>
<td>instrument unique parameters (status for masks, filters, gratings, probes, etc.)</td>
<td>✓</td>
</tr>
<tr>
<td>TBD (instrument specific)</td>
<td>✓</td>
</tr>
</tbody>
</table>
8. Observing Tool Interfaces

The LBTO Observing Tool (OT) is an observation planning tool available for facility instruments. Instrument configuration for different modes and situations can be specified. The tool allows for visualization of the science field, guide stars, guide probe, offsets, field rotation, etc. It calculates precise time allocations for each observation. It keeps a real time update of the observing log as it happens, an account of the data being generated, etc. Information on the LBT OT can be found at:

https://sites.google.com/a/lbto.org/observing-tool-manual

The LBTO does not have a standard for its instruments or scripts and as a result each instrument has its own script definition and format. This means that a script generated by the OT could be translated in as many ways as instruments the LBT supports. The instrument scripts differ in syntax and also morphology, for instance OT sequence steps might be grouped in items differently depending on the instrument type and even instrument configuration. Some use xml and others plain text commands, etc.

The OT generates observing block scripts for use by LBTO facility instruments. Non-facility instrument script generation is TBD.
9. Abbreviations and Acronyms

ALH  Alarm Handler
AO   Adaptive Optics
ARGOS Advanced Rayleigh-guided Ground layer adaptive Optics System
EPICS Experimental Physics and Industrial Control System
CAN  Category Assembly Number – LBTO document repository system
DD   Data Dictionary
DDS  Data Dictionary Server TCS subsystem
FACSUM Facility Summary GUI
FS   Facility Summary GUI
FWHM Full Width Half Max
GUI  Graphical User Interface
HDF5 Hierarchical Data Format
IAS  Instrument Alarm System
IF   Interferometry
IIF  Instrument Interface TCS subsystem
LBC  Large Binocular Camera
LBT  Large Binocular Telescope
LBTI Large Binocular Telescope Interferometer
LBTO Large Binocular Telescope Observatory
LINC/NIRVANA LBT Interferometric Camera/Near Infrared Visible Adaptive Interferometer for Astronomy
LUCI LBT Near-Infrared Spectroscopic Utility with Camera and Integral Field Unit for Extragalactic Research
MODS Multi-Object Double Spectrograph
OSU  Ohio State University
OT   Observing Tool
OVMS Optical Path Difference and Vibration Management System
PEPSIPFU Potsdam Echelle Polarimetric and Spectroscopic Instrument Permanent Fiber Unit
PEPSIPOL Potsdam Echelle Polarimetric and Spectroscopic Polarimeter
PLC  Programmable Logic Controller
TCS  Telescope Control System
UTC  Coordinated Universal Time
10. References / Links

[1] ICE Instrument Interface Control Document
   http://abell.as.arizona.edu/~hill/xlbt/cgi/ican.cgi?481s013


[3] Introduction to INDI


[5] indilib.org


[14] “Queue software reuse and implementation at the Large Binocular Telescope Observatory”,
    http://spie.org/Publications/Proceedings/Paper/10.1117/12.2232990
Appendix A – Sample INDI Service

The following is a code snippet from a C++ program that reads a file of values and publishes them as INDI properties for some observatory weather parameters. This example uses the framework classes provided by LBTI for IndiDriver, IndiElement, and IndiProperty. The framework provides easy access to logging, signal handling, XML manipulation, etc.

Only a little more sophistication is required in a real INDI driver – for example, error handling and checking for changed parameters before setting.

NOTE: If you are using the libcommon for these classes, make sure you have the updated IndiDriver.cpp file, or you could have bad behavior in the SendXml method.

class LBTO_DDS : public pcf::IndiDriver
{
    private:
        // declare our properties - separate for numbers, text, etc.
        pcf::IndiProperty weather;

        // declare a handy logger instance
        pcf::Logger log;

    public:
        // override constructor
        LBTO_DDS (const std::string &szName,
                   const std::string &szDriverVersion,
                   const std::string &szProtocolVersion);

        // override callbacks
        virtual void handleGetProperties( const pcf::IndiProperty &ipRecv );
        virtual void handleNewProperty( const pcf::IndiProperty &ipRecv );
        virtual void execute(void);
    };

LBTO_DDS::LBTO_DDS (const std::string &szName,
                     const std::string &szDriverVersion,
                     const std::string &szProtocolVersion )
    : IndiDriver( szName, szDriverVersion, szProtocolVersion )
{
    // configure for one log entry per line
    log.enableClearAfterLog( true );

    // set up the weather input property -- all numbers
    weather = IndiProperty (IndiProperty::Number);
    weather.setDevice(szName);
    weather.setName("weather");
    weather.setPerm(IndiProperty::ReadOnly);
    weather.setState(IndiProperty::Idle);
    weather.add(IndiElement ("Humidity", "Humidity, percent", "%4.1f"));
    weather.add(IndiElement ("Temperature", "Air temp, degs C", "%8.1f"));
    weather.add(IndiElement ("WindDir", "Wind direction, degs E of N", "%8.1f"));
}

void LBTO_DDS::handleGetProperties( const IndiProperty &ipRecv )
{
// ignore if not our device
if (ipRecv.hasValidDevice() && ipRecv.getDevice() != getName())
  return;

// publish definition if want all properties or a specific one
if (!ipRecv.hasValidName() || ipRecv.getName() == weather.getName())
  sendDefProperty (weather);
}

void LBTO_DDS::handleNewProperty( const pcf::IndiProperty &ipRecv )
{
  // for this example, the properties are read-only, so we do
  // not need to override this method
}

void LBTO_DDS::execute()
{
  // Get the parameters file written by wda for FACKSUM/WEATHERSTATION
  system("wget -nc -q http://ourwebsite.org/params.txt");
  std::vector<std::string> parameterStrings;
  std::ifstream readFile("/params.txt");  //writes to top-slash dir..
  while(getline(readFile,line))   {
    parameterStrings.push_back(line);
  }

  // Iterate over each line in the file and extract the parameter name and value.
  for(std::vector<std::string>::iterator it = parameterStrings.begin();
      it != parameterStrings.end(); ++it)
  {
    int spaceIndex = it->find_first_of(" ");
    std::string parameter = it->substr(0,spaceIndex);
    std::string value = it->substr(spaceIndex+1,(it->length()-spaceIndex));

    double dbl_wx;
    if (parameter == "env.weather.lbt.humidity")
    {
      dbl_wx   = atof(value.c_str());
      weather["Humidity"] = dbl_wx;
    }
    if (parameter == "env.weather.lbt.temperature")
    {
      dbl_wx   = atof(value.c_str());
      weather["Temperature"] = dbl_wx;
    }
    if (parameter == "env.weather.lbt.windDirection")
    {
      dbl_wx   = atof(value.c_str());
      weather["WindDir"] = dbl_wx;
    }
  }

  sendDefProperty (weather);
  sleep(1);
}
/ INDI driver Device, and versions for code and properties
#define INDI_DRIVER_DEVICE "LBTO"
#define INDI_DRIVER_VERSION "0.1"
#define INDI_PROTOCOL_VERSION "0.1"

int main( int argc, char **argv )
{
    // What path should we use for configuration & logging?
    std::string szIndiPath( ::getenv( "LBTDIHOME" ) );
    szIndiPath = ( szIndiPath.length() == 0 ) ? ( "." ) : ( szIndiPath );

    // Initialize the logging mechanism
    pcf::Logger::init( szIndiPath + "/logs/" + INDI_DRIVER_DEVICE,
                      INDI_DRIVER_DEVICE );

    // Initialize the configuration from a file.
    int iErr = pcf::Config::init( szIndiPath + "/config/", INDI_DRIVER_DEVICE
                                 ".cfg" );
    if ( iErr != pcf::ConfigFile::ErrNone )
    {
        pcf::Logger msgInfo;
        msgInfo << pcf::Logger::enumError << "Config for " << INDI_DRIVER_DEVICE
                 << " had an error: " << pcf::Config::getErrorMsg( iErr ) << std::endl;
    }

    // instantiate an INDI driver
    LBTO_DDS driver( INDI_DRIVER_DEVICE, INDI_DRIVER_VERSION, INDI_PROTOCOL_VERSION
                     );

    // start the execute loop()
    driver.activate();

    // handle all incoming messages, only returns when process is signaled
    driver.processIndiRequests();

    // kindly stop the execute loop()
    driver.deactivate();
    return (0);
}
The following is a sample web page that reads the above weather data INDI properties and prints them. This code uses the indi.fcgi gateway process as specified in the INDI introduction document. The LBTI-provided indi.js file (provided by the LBTI development team) provides the core of the processing. The only html/JavaScript required is something like:

```html
<!DOCTYPE html>
<h2>LBTO Prototype Status Server (using INDI)</h2>
<script src="/jquery/jquery.js"></script>
<script src="/jquery/jquery-ui.min.js"></script>
<script src="/js/utility.js"></script>
<script src="/js/indi.js"></script>
<p><b>LBTO weather</b></p>
<script>
setPropertyCallback ("LBTO.weather", function(map) {
    document.getElementById("indidemo").innerHTML =
        "&nbsp;&nbsp; LBTO.weather.Temperature = " + map['Temperature'] +
        "&nbsp;&nbsp; LBTO.weather.Humidity = " + map['Humidity'] +
        "&nbsp;&nbsp; LBTO.weather.WindDir = " + map['WindDir'];
});
</script>
</body></html>
```
Appendix B – Simple Telemetry Interface Program

An hdf5 source tar file with the LBTO modified source code has been added to SVN with a README file for the required configure options, etc. If the HDF5 library is not configured correctly, the telemetry collection library will complain.

https://svn.lbto.org/repos/tools/trunk/README-LBT-hdf5-1.10.txt
https://svn.lbto.org/repos/tools/trunk/LBT-hdf5-1.10.0.bz2

Anyone using LBT’s telemetry collection library (https://svn.lbto.org/repos/telemetry/trunk/collection) should use this version of the HDF library.

The following is a sample program using the LBTO telemetry collection library to write three values to a telemetry stream – a float, a 32-bit integer, and a boolean.

#include <telcollection.hxx>

/// simple standalone program to read a file of weather data
/// and create a telemetry stream of three values – int, float, bool
/// the input data file contains a timestamp that is from 2014, but
/// we’re ignoring that and writing each record with the current
/// timestamp, as you would in a typical telemetry-writing scenario.

std::tr1::shared_ptr <lbto::tel::collector> telCollector;

class callback : public lbto::tel::ambassador
{
public:
    callback() throw() {};
    ~callback() throw() {};

    /// what is this name? it’s not in the directory structure – what does it have to
    /// be unique to?
    lbto::tel::name get_name() throw(std::bad_alloc)  {return lbto::tel::name(“test”);}

    std::string get_hdf_path_name() throw(std::bad_alloc)  {return ”/data/telemetry”;}

    std::string get_leap_seconds_file_name() throw(std::bad_alloc)  {return ”/lbt/UT/leap-seconds.list”;}

    std::string get_disk_buffer_dir() throw(std::bad_alloc)  {return ”/tmp”;}

    uint64_t get_disk_buffer_size() throw()  {return 12000000;}

    /// Telemetry notice message – this is currently only sent for
    /// the leap-seconds file being expired
    void notice(const std::string msg) throw() {

```cpp
std::cout << "Telemetry notice: " << msg << std::endl;
}

/// Telemetry critical message
void handle_critical(const std::string msg) throw() {
    std::cout << "Telemetry critical: " << msg << std::endl;
}

/// Collection connected to Telemetry
void notify_connected(const lbto::tel::name &telemeterName, const lbto::tel::system &system) throw() {
    std::cout << std::string(telemeterName) << " collection connected to Telemetry" << std::endl;
}

/// Collection disconnected from Telemetry
void notify_disconnected(const lbto::tel::name &telemeterName, const lbto::tel::system &system) throw() {
    std::cout << std::string(telemeterName) << " collection disconnected from Telemetry" << std::endl;
}

void telInit();
void store(struct WEATHER_INFO);
void sigHandler(int);
bool quit = false;

/// a structure representing the current values of
/// all the fields in the telemetry stream, to be
/// "stored" in each of the "measures"
struct WEATHER_INFO {
    float temperature;
    int windDirection;
    int rain;
};

/// we need a "measure" variable for each of the fields in
/// the telemetry stream
lbto::tel::float_measure::buf_proxy temperature;
lbto::tel::int_32_measure::buf_proxy windDirection;
lbto::tel::bool_measure::buf_proxy rain;

int main(int narg, char* argc[]) {
    /// set up ^C signal handler
    struct sigaction action;
    action.sa_handler = sigHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGINT, &action, NULL);
```
/// initialize telemetry
std::tr1::shared_ptr<lbto::tel::ambassador> callBack(new callback);
try {
    lbto::tel::collection_manager::init(callBack);
}
catch(std::exception const & Exn) {
    std::cout << "main() telemetry exception: " << Exn.what() << std::endl;
} catch(...) {
    std::cout << "main() unknown telemetry exception" << std::endl;
}
telInit();

std::ifstream wxFile;
std::string wxFilename("20140921MergedData.txt");
std::cout << "opening filename " << wxFilename << std::endl;
wxFilename.open(wxFilename.c_str(), std::ios_base::in);
char readString[256];
char timeStr[15];
struct WEATHER_INFO wxInfo;
float fDummy = 0.0;
if (wxFile.is_open()) {
    wxFile.getline(readString,200);   // read the title line
    while (!wxFile.eof()) {
        wxFile.getline(readString,200);
        sscanf(readString, "%s %f %f %f %f %d %f %f %f %f",
               timeStr,
               &wxInfo.temperature,
               &fDummy,
               &fDummy,
               &fDummy,
               &fDummy,
               &wxInfo.windDirection,
               &fDummy,
               &wxInfo.rain,
               &fDummy,
               &fDummy,
               &fDummy);
        store(wxInfo);
    }
}
wxFile.close();

/// wait until buffer is empty
quit = false;
while(telCollector->count_buffered_samples() != 0)
{
    if (quit)
        break;
    sleep(1);
}
telCollector.reset();
return 0;

void telInit()
{
    std::cout << "Initializing telemetry" << std::endl;
    try
    {
        /// the system name will be the top-level directory structure,
        /// the other name will be the name of the stream - converted to lower-case and underscores added
        /// in this example, we will have a /data/telemetry/env directory with dated directories underneath
        /// in today's directory (/data/telemetry/env/2016/MM/DD) we will have a YYYYMMDDHHMM.env.lbt_weather.h5 file
        lbto::tel::telemeter_definer myDefiner = lbto::tel::collection_manager::instance().make_telemeter_definer
            (lbto::tel::system(lbto::tel::name("env")), lbto::tel::name("LBT weather"));

        /// the order the measures are added to the definer will determine the order of the fields in the stream
        /// the stream named lbt_weather_01 will have fields: temperature, wind_direction, rain
        /// along with the telemetry-added fields of time_stamp and tai_offset
        myDefiner.add_child(lbto::tel::float_measure(temperature, lbto::tel::unit::celsius(), lbto::tel::name("temperature")));
        myDefiner.add_child(lbto::tel::int_32_measure(windDirection, lbto::tel::unit::degree(), lbto::tel::name("wind direction")));
        myDefiner.add_child(lbto::tel::bool_measure(rain, lbto::tel::name("rain")));

        lbto::tel::telemeter_definition const myDefinition = myDefiner.make_definition();
telCollector.reset(new lbto::tel::collector(myDefinition, 10000000, false));
    }
    catch(std::exception const & Exn)
    {
        std::cout << "telInit() exception: " << Exn.what() << std::endl;
        exit(1);
    }
    catch(...)
    {
        std::cout << "telInit() unknown exception" << std::endl;
        exit(1);
    }
}
void store(struct WEATHER_INFO wxInfo)
{
    try
    {
        temperature.store(wxInfo.temperature);
        windDirection.store(wxInfo.windDirection);
        rain.store(wxInfo.rain);

        // write this sample with current timestamp, not the timestamp from our input file
        struct timeval current_time;
        gettimeofday(&current_time, NULL);
        telCollector->commit_sample(lbto::tel::date::from_posix_utc_s(current_time));
    }
    catch(lbto::tel::sample_dropped const & Exn)
    {
        std::cout << "sample dropped: " << Exn.what() << std::endl;
    }
    catch(std::exception const & Exn)
    {
        std::cout << "store() exception: " << Exn.what() << std::endl;
    }
    catch(...)
    {
        std::cout << "store() unknown exception" << std::endl;
    }
}

void sigHandler(int sig)
{
    quit = true;
}
Appendix C - Current TCS Data Dictionary Definition for Facility Instruments

The following parameters are currently in the TCS shared memory definition; if the name has an “S_” in the front, it is a “sided” variable which means there is a “L_” and a “R_” version. The ‘X’ in the Status field below means the item is currently populated by the LBC instrument.

<table>
<thead>
<tr>
<th>DD Item – Public Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBCOBProgress</td>
<td>string</td>
<td>---</td>
<td>OB status (ready/running)</td>
<td></td>
</tr>
<tr>
<td>S_LBCFilter</td>
<td>string</td>
<td>---</td>
<td>Filter name</td>
<td></td>
</tr>
<tr>
<td>S_LBCShutterState</td>
<td>string</td>
<td>---</td>
<td>N/A/enabled/disabled/not commanded/open/closed</td>
<td></td>
</tr>
<tr>
<td>S_LBCDetectorState</td>
<td>string</td>
<td>---</td>
<td>ready/setup/flushing/hard flushing/exposure/saving readout/passive/preset</td>
<td></td>
</tr>
<tr>
<td>S_LBCGuidingState</td>
<td>string</td>
<td>---</td>
<td>processing/failed/not required/delaying/starting/exposing/ready/..</td>
<td></td>
</tr>
<tr>
<td>S_LBCRotatorAngle</td>
<td>float</td>
<td>deg</td>
<td>Rotator angle</td>
<td></td>
</tr>
<tr>
<td>S_LBCCCDTemp</td>
<td>float</td>
<td>K</td>
<td>CCD detector temperature</td>
<td></td>
</tr>
<tr>
<td>S_LBCDewarPres</td>
<td>float</td>
<td>mbar</td>
<td>CCD dewar vacuum pressure</td>
<td></td>
</tr>
<tr>
<td>S_LBCDewarTemp</td>
<td>float</td>
<td>K</td>
<td>CCD dewar LN reservoir temperature</td>
<td></td>
</tr>
<tr>
<td>S_LBCFWHM</td>
<td>float</td>
<td>arcsec</td>
<td>Calculated full width at half max for the guide image</td>
<td></td>
</tr>
<tr>
<td>S_LBCPowerState</td>
<td>int</td>
<td>---</td>
<td>Power state</td>
<td></td>
</tr>
<tr>
<td>S_LBCSetupState</td>
<td>int</td>
<td>---</td>
<td>Configuration state</td>
<td></td>
</tr>
<tr>
<td>S_LBCHealthState</td>
<td>int</td>
<td>---</td>
<td>Health state</td>
<td></td>
</tr>
<tr>
<td>S_LBCHeartbeatTime</td>
<td>string</td>
<td>HMS</td>
<td>When the last data item was sent to DD from LBC</td>
<td></td>
</tr>
<tr>
<td>S_LBCExposureSeq</td>
<td>string</td>
<td>---</td>
<td>Exposure m of n in the sequence</td>
<td></td>
</tr>
<tr>
<td>S_LBCImageSeq</td>
<td>string</td>
<td>---</td>
<td>Image m of n in the sequence</td>
<td></td>
</tr>
<tr>
<td>S_LBCFilterSeq</td>
<td>string</td>
<td>---</td>
<td>Filter change m of n in the sequence string</td>
<td></td>
</tr>
</tbody>
</table>
LUCI is writing most of the following TCS DD parameters.

<table>
<thead>
<tr>
<th>DD Item – Public Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_LUCIDDDUpdateTime</td>
<td>double</td>
<td>days</td>
<td>MJD UT heartbeat time</td>
</tr>
<tr>
<td>S_LUCIFilterWheel1Position</td>
<td>int</td>
<td>---</td>
<td>Filter wheel 1 number</td>
</tr>
<tr>
<td>S_LUCIFilterWheel1Status</td>
<td>int</td>
<td>---</td>
<td>Filter wheel 1 status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCIFilterWheel2Position</td>
<td>int</td>
<td>---</td>
<td>Filter wheel 2 number</td>
</tr>
<tr>
<td>S_LUCIFilterWheel2Status</td>
<td>int</td>
<td>---</td>
<td>Filter wheel 2 status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCIGratingSelected</td>
<td>string</td>
<td>---</td>
<td>Grating name</td>
</tr>
<tr>
<td>S_LUCIGratingStatus</td>
<td>int</td>
<td>---</td>
<td>Grating status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCICameraName</td>
<td>string</td>
<td>---</td>
<td>Name of selected camera</td>
</tr>
<tr>
<td>S_LUCICameraUnitStatus</td>
<td>int</td>
<td>---</td>
<td>Camera status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCIPupilViewerPosition</td>
<td>int</td>
<td>---</td>
<td>Pupil viewer position (0 in, 1 out, -1 unknown)</td>
</tr>
<tr>
<td>S_LUCIPupilViewerStatus</td>
<td>int</td>
<td>---</td>
<td>Pupil viewer status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCICalibrationUnitPosition</td>
<td>int</td>
<td>---</td>
<td>Calibration unit position (0 out, 1 in, -1 unknown)</td>
</tr>
<tr>
<td>S_LUCICalibrationUnitStatus</td>
<td>int</td>
<td>---</td>
<td>Calibration unit status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCIReadoutStatus</td>
<td>int</td>
<td>---</td>
<td>Readout status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>S_LUCTemperatureAlarm</td>
<td>bool</td>
<td>---</td>
<td>True if temperature in alarm</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[0]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[1]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[2]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[3]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[4]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[5]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[6]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[7]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[8]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[9]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[10]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[12]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCTemperatureStatus[13]</td>
<td>float</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>S_LUCIPressures[0]</td>
<td>float</td>
<td>mbar</td>
<td>Measured pressure</td>
</tr>
<tr>
<td>S_LUCIPressures[1]</td>
<td>float</td>
<td>mbar</td>
<td>Measured pressure</td>
</tr>
<tr>
<td>S_LUCIPressureStatus[0]</td>
<td>int</td>
<td>---</td>
<td>Pressure status (0=OK, 1=underRange, 2=overRange, 3=sensorError, 4=sensorOff, 5=noSensor, 6=IDerror)</td>
</tr>
<tr>
<td>S_LUCIPressureStatus[1]</td>
<td>int</td>
<td>---</td>
<td>Pressure status (0=OK, 1=underRange, 2=overRange, 3=sensorError, 4=sensorOff, 5=noSensor, 6=IDerror)</td>
</tr>
<tr>
<td>S_LUCIExpTimeRemaining</td>
<td>float</td>
<td>sec</td>
<td>Exposure time remaining</td>
</tr>
<tr>
<td>S_LUCIExpTimeRemaining</td>
<td>float</td>
<td>sec</td>
<td>Exposure time remaining</td>
</tr>
<tr>
<td>S_LUCIExpTimeRemaining</td>
<td>float</td>
<td>sec</td>
<td>Exposure time remaining</td>
</tr>
<tr>
<td>S_LUCIExpTimeRemaining</td>
<td>float</td>
<td>sec</td>
<td>Exposure time remaining</td>
</tr>
<tr>
<td>S_LUCIExpTimeRemaining</td>
<td>float</td>
<td>sec</td>
<td>Exposure time remaining</td>
</tr>
<tr>
<td>S_LUCIDetectorState</td>
<td>string</td>
<td>---</td>
<td>Integrating, stopped, flushing, etc</td>
</tr>
<tr>
<td>S_LUCIDetectorStatus</td>
<td>int</td>
<td>---</td>
<td>Detector status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>S_LUCIMOSUnitStatus</td>
<td>int</td>
<td>MOS unit status (0=OK, 1=active, 2=idle, 3=engineering, 4=warning, 5=error, 6=panic)</td>
<td></td>
</tr>
<tr>
<td>S_LUCIMaskID</td>
<td>string</td>
<td>ID of mask in use</td>
<td></td>
</tr>
<tr>
<td>S_LUCIShutterState</td>
<td>string</td>
<td>open/closed</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPowerOn[0]</td>
<td>bool</td>
<td>Power for MCE, ROE, coldHead1, coldHead2</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPowerOn[1]</td>
<td>bool</td>
<td>Power for MCE, ROE, coldHead1, coldHead2</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPowerOn[2]</td>
<td>bool</td>
<td>Power for MCE, ROE, coldHead1, coldHead2</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPowerOn[3]</td>
<td>bool</td>
<td>Power for MCE, ROE, coldHead1, coldHead2</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPower[0]</td>
<td>float</td>
<td>volt</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPower[1]</td>
<td>float</td>
<td>volt</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPower[2]</td>
<td>float</td>
<td>volt</td>
<td></td>
</tr>
<tr>
<td>S_LUCIPower[3]</td>
<td>float</td>
<td>volt</td>
<td></td>
</tr>
<tr>
<td>S_LUCIExposureState</td>
<td>string</td>
<td>Running, ready</td>
<td></td>
</tr>
<tr>
<td>S_LUCISaveState</td>
<td>string</td>
<td>Saving, …</td>
<td></td>
</tr>
</tbody>
</table>
MODS has defined all the housekeeping data required, although nothing yet is being written to the TCS DD. See MODS DD Parameters OSU-MODS-2009-008 Version 1.1. A few more instrument status data items are suggested in this document that are not part of the MODS definitions.

<table>
<thead>
<tr>
<th>DD Item – Public Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
<th>MODS keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_MODSBlueCCDTemp</td>
<td>float</td>
<td>C</td>
<td>Blue camera CCD detector mount temperature</td>
<td>blueCCDTemp</td>
</tr>
<tr>
<td>S_MODSBlueDewPres</td>
<td>float</td>
<td>torr</td>
<td>Blue camera CCD dewar vacuum pressure</td>
<td>blueDewarPressure</td>
</tr>
<tr>
<td>S_MODSBlueDewTemp</td>
<td>float</td>
<td>C</td>
<td>Blue camera CCD dewar LN reser voir temperature</td>
<td>blueDewarTemp</td>
</tr>
<tr>
<td>S_MODSBlueHEBTemp</td>
<td>float</td>
<td>C</td>
<td>Blue camera CCD Head Electronics Box internal air temperature</td>
<td>blueHEBTemp</td>
</tr>
<tr>
<td>S_MODSBlueIEBTemp</td>
<td>float</td>
<td>C</td>
<td>Blue camera Instrument Electronics Box internal air temperature</td>
<td>blueIEBTemp</td>
</tr>
<tr>
<td>S_MODSBlueImageQuality</td>
<td>float</td>
<td>TBD</td>
<td>Blue image quality metric</td>
<td>TBD</td>
</tr>
<tr>
<td>S_MODSBlueFilter</td>
<td>string</td>
<td>---</td>
<td>Blue filter name</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueGrating</td>
<td>string</td>
<td>---</td>
<td>Blue grating name</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueShutterState</td>
<td>string</td>
<td>---</td>
<td>Blue CCD shutter state (open/closed)</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueDetectorState</td>
<td>string</td>
<td>---</td>
<td>Blue CCD exposure state (ready/setup/flushing/exposure/saving)</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueSetupState</td>
<td>int</td>
<td>---</td>
<td>Blue configuration state</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedCCDTemp</td>
<td>float</td>
<td>C</td>
<td>Red camera CCD detector mount temperature</td>
<td>redCCDTemp</td>
</tr>
<tr>
<td>S_MODSRedDewPres</td>
<td>float</td>
<td>torr</td>
<td>Red camera CCD dewar vacuum pressure</td>
<td>redDewarPressure</td>
</tr>
<tr>
<td>S_MODSRedDewTemp</td>
<td>float</td>
<td>C</td>
<td>Red camera CCD dewar LN reservoir temperature</td>
<td>redDewarTemp</td>
</tr>
<tr>
<td>S_MODSRedHEBTemp</td>
<td>float</td>
<td>C</td>
<td>Red camera CCD Head Electronics Box internal air temperature</td>
<td>redHEBTemp</td>
</tr>
<tr>
<td>S_MODSRedIEBTemp</td>
<td>float</td>
<td>C</td>
<td>Red camera Instrument Electronics Box internal air temperature</td>
<td>redIEBTemp</td>
</tr>
<tr>
<td>S_MODSTimeStamp</td>
<td>long</td>
<td>sec</td>
<td>Time of data measurement in seconds from epoch</td>
<td></td>
</tr>
<tr>
<td>S_MODSTimeStamp</td>
<td>string</td>
<td>HMS</td>
<td>Date/time string UTC ISO-8601 format</td>
<td></td>
</tr>
<tr>
<td>S_MODSUtilBoxTemp</td>
<td>float</td>
<td>C</td>
<td>Utility distribution box internal air temperature</td>
<td>utilBoxTemp</td>
</tr>
<tr>
<td>S_MODSUtilState</td>
<td>int</td>
<td>---</td>
<td>Calibration state</td>
<td></td>
</tr>
<tr>
<td>S_MODSMaskID</td>
<td>string</td>
<td>---</td>
<td>Mask identifier</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueImageQuality</td>
<td>float</td>
<td>TBD</td>
<td>Blue image quality metric</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueFilter</td>
<td>string</td>
<td>---</td>
<td>Blue filter name</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueGrating</td>
<td>string</td>
<td>---</td>
<td>Blue grating name</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueShutterState</td>
<td>string</td>
<td>---</td>
<td>Blue CCD shutter state (open/closed)</td>
<td></td>
</tr>
<tr>
<td>S_MODSBlueDetectorState</td>
<td>string</td>
<td>---</td>
<td>Blue CCD exposure state (ready/setup/flushing/exposure/saving)</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedSetupState</td>
<td>int</td>
<td>---</td>
<td>Red configuration state</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedImageQuality</td>
<td>float</td>
<td>TBD</td>
<td>Red image quality metric</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedFilter</td>
<td>string</td>
<td>---</td>
<td>Red filter name</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedGrating</td>
<td>string</td>
<td>---</td>
<td>Red grating name</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedShutterState</td>
<td>string</td>
<td>---</td>
<td>Red CCD shutter state (open/closed)</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedDetectorState</td>
<td>string</td>
<td>---</td>
<td>Red CCD exposure status (ready/setup/flushing/exposure/saving)</td>
<td></td>
</tr>
<tr>
<td>S_MODSRedSetupState</td>
<td>int</td>
<td>---</td>
<td>Red configuration state</td>
<td></td>
</tr>
</tbody>
</table>