Blu-Ice/DCS Administrator's Manual for Release 4.1

Scott McPhillips
Stanford Synchrotron Radiation Laboratory
scottm@slac.stanford.edu

July 6, 2011

Contents

1 Introduction

This document covers topics that will assist software developers and/or beam line administrators in installing and configuring the complete DCS framework. The DCS framework consists of a number of distributed software components designed for the purpose of controlling a protein crystallography beam line. This document describes the framework and implementation of SSRL beamlines. Additionally, instructions for installing and configuring a simple beamline simulation are included.

1.1 Software License Agreement

Disclaimer Notice

Work supported by the U.S. Department of Energy under contract DE-AC03-76SF00515; and the National Institutes of Health, National Center for Research Resources, grant 2P41RR01209.

Permission Notice

1.2 Definitions

- DCS (Distributed Control System): The system framework consisting of GUI/Clients, DCSS, and DHS programs.
- DCS Protocol: The plain text protocol spoken over TCP/IP sockets used between the programs in the DCS framework.
- GUI/client Protocol: Subset of the DCS protocol that is used by DCSS and the GUI/clients.
- DHS Protocol: Subset of the DCS protocol used between DCSS and the DHS programs.
- DCSS (Distributed Control System Server): The centralized server that opens listening ports for GUI/Clients and DHS programs. It acts a message router between the GUI/Clients and the DHS programs. It is also responsible for handling user permissions and storing state of the beam line.
- DHS (Distributed Hardware Server): Any program that speaks the DCS hardware protocol (or subset of it). A typical DHS accepts DCS messages and controls a piece of hardware directly. A DHS may also be used to translate and forward DCS messages to an alternate control system.
- Blu-Ice: SSRL's standard Graphical User Interface used for alligning a beam line and defining/monitoring Protein Crystallography experiments. This program speaks the "GUI/Client" protocol and connects to DCSS.
- Diffraction Image Server: A program that loads and caches diffraction images as requested, returning views of the diffraction image in compressed JPEG format. This is used by Blu-Ice for viewing diffraction images.

1.3 What's new for Release 4.1

1.3.1 Improved makefiles and batchbuild project

Projects now have a makefile in their top level directory. These makefiles have been tested on several different platforms using the gnu make program. To build a project simply type "gmake" in the project's root directory.

A CVS project by the name of `batchbuild` has been added which will assist in checking out other necessary projects out of CVS. The batchbuild project contains a gnu makefile which will support the following commands:

- "`gmake co`" Checks out all of the CVS projects.
- "`gmake clean`" Deletes the object and binary files out of all projects.
- "`gmake` " Builds all projects.
- "`gmake basic`" Builds a subset of the DCS projects for the purpose of a lightweight installation.

Also provided is a `restart_dcs` scripts which can be modified to start all of the DCS programs needed for a simple installation.

1.3.2 Simplified file based configuration

One of the challenges of creating a Distributed Control System is providing a way for each software program within the system to obtain its configuration easily. For several years DCS development involved a centralized mysql database which was used to maintain the complete configuration for all beam lines. However, experience has shown that maintaining a mysql database for this purpose is far more difficult than managing simple text files for configuration.

Release 4.0 introduces the dcsconfig project to help handle distributed configurations. This project provides a simple API for obtaining configuration data from a text file.

The following programs now use the dcsconfig project:

1. Enhanced and original Blu-Ice.
2. Distributed Control System Server (dcss)
3. Diffraction Image Server (imgsrv)
4. Simulated DHS (simdhs)
5. MyAuthClient
6. The "Legacy" Distributed Hardware Server (dhs project), controlling the following devices:
   (a) Galil DMC2180

(b) MAR 345 detector
(c) MAR family of CCD detectors
(d) Quantum 4 CCD detector
(e) Quantum 315 CCD detector
(f) Image analysis of sample jpeg for automated crystal centering

1.3.3 Authentication

SSRL's installation of the DCS project allows each user to collect data into their own accounts without requiring staff or users to restarting any of the distributed software components. This feature requires that the distributed components are capable of authenticating a particular user.

Additionally, web-related projects are under development and also require authentication. For this, and other reasons, we have developed a http based protocol for authentication that relies heavily on our web server infrastructure. Dcss has migrated to this style of authentication and so relies on an external program to authenticate users.

In the interest of keeping things simple for external collaborators, we have developed a simple program called MyAuthServer, which adheres to the same protocol and will authenticate users for all DCS applications without any code changes to these applications. Configuration of this program is included in this documentation.

Currently, the MyAuthServer does not except a secure SSL connection, so this implimentation should be used on a private network or local to a single machine for security reasons.

1.3.4 Impersonation Server

1.3.5 Blu-Ice: Enhanced vs. Original

Large portions of the Blu-Ice GUI have been completely rewritten. Because the internal changes are so significant, we have checked it in as a separate CVS project and have left the "original" blu-ice project in place, only upgrading it enough to communicate with DCS 4.0.

The "enhanced" Blu-Ice version is easier to configure and is designed with the intention of being able to rapidly develop new features. This version has the following advantages:

1. Object Oriented architecture
2. Relies on incr widgets for GUI.

3. Uses the dcsconfig project which allows easier configuration.

4. The application can be started with a desktop type interface that allows a developer to only open widgets of interest.

5. Scans are defined in Blu-Ice and are run in DCSS. This allows other Blu-Ice clients to monitor scans. This functionality is available for motor scans as well as fluorescence and excitation scans.

6. Improved visualitation of 2-d scans.

However, this version currently has the following disadvantages:

1. Incr widgets are slower, so a computer upgrade may be desired.

2. The command prompt is missing, which is a nice feature in the old blu-ice. Hopefully it will make it back with the next release.

3. The fonts and general shape of things are slightly different.

1.3.6 C++ library for developing new DHS programs

The dcsmsg project is a C++ library for handling the DCS message protocol for new DHS development. This project allows developers to build support for new hardware rapidly. There are several example DHS projects that use this new library:

1. Adac5500DHS (A/D card support for the adac5500 card).

2. Dsa2000DHS (Support for the DSA2000 fluorescence detector.)

3. epicsdhs (Reads PV's and can wait for PV's to become a certain state.)

4. impdhs (Allows access of an impersonation server via a DHS operation command.)

1.3.7 C++ Logging library

A new C++ library has been added which provides an API for logging. This is now being used by some of the DHS programs, the diffraction image server, and by DCSS.

1.3.8 Optimized C code for handling DCS protocol

Blu-ice and Dcss have previously used TCL code to handle the parsing of DCS messages. This TCL code had a drawback: the handling of messages slowed as the backlog of unhandled messages increased.

In order to speed up the handling of the protocol at connection time and during times of many motors moving simultaneously, a C library was created to handle the parsing of the messages.

At start-up, the Blu-Ice code will search for the C library and will use the optimized code if available. If the library is not available, the TCL-only code will be used automatically.

1.3.9 Simulated Detector DHS

A new program has been written that functions as a detector DHS, but will simply copy diffraction image files from one directory to another each time that DCSS requests an image. This program is useful for software developers that do not have access to a real area detector.

2 Installation

2.1 Using CVS for source code control

Currently the DCS/Blu-Ice software is maintained and managed using CVS. The web offers excellent CVS Documentation[1] to help a developer get started. Specifics about how the Blu-Ice/DCS software uses CVS is described more thoroughly in the following sub-sections.

To access the CVS repository you will need a username and password provided by SSRL. CVS does not provide secure encryption, so the CVS account will not be related to an actual SSRL computer account.

If you do not have a CVS account send a request to scottm@slac.stanford.edu with a brief statement of interest and the synchrotron and beam line(s) of which you are affiliated.

2.2 Obtaining and Building the Software

NOTE: This documentation was tested against gcc version 3.2.3 on Linux Red Hat Enterprise.

[1] http://cvsbook.red-bean.com/cvsbook.html

1. Create a new directory to install the software in. This documentation will refer to this directory as the 'DCS root' directory and the examples will use a 'DCS root' directory of `~/release-4_1/`.

   ```
   mkdir release-4_1
   ```

2. `cd release-4_1`

3. Log in to CVS replacing *yourusernameName* in the following line with your account name.

   ```
   cvs -d :pserver:yourusername@smb.slac.stanford.edu:/home/code/repository log
   Logging in to :pserver:yourusername@smb.slac.stanford.edu:2401/home/code/rep
   CVS password:
   ```

4. Checkout the batchbuild project, replacing *yourusernameName* in the following line with your account name.

   ```
   cvs -d :pserver:yourusername@smb.slac.stanford.edu:/home/code/repository
   checkout -r release-4_1 batchbuild
   ```

5. Change into the batchbuild project directory.

   ```
   cd batchbuild
   ```

6. Edit the makefile in the `batchbuild` directory, find the `CVSCOMMAND` definition, and change it as follows in order to have the makefile access the CVS repository remotely:

   ```
   CVSCOMMAND=cvs -d :pserver:yourusername@smb.slac.stanford.edu:/home/code/rep
   #CVSCOMMAND=cvs
   ```

7. From within the batchbuild directory, type the following to download the complete software:

   ```
   gmake co
   ```

8. From within the batchbuild directory, type the following to build the software:

   ```
   gmake basic
   ```

9. Correct any build errors in your environment until the gmake command completes successfully.

2.3 Testing the Installation

The following steps can be used to evaluate your installation.

1. Copy the BL_simple1.config and BL_simple11.dat files from the dcsconfig/examples directory to the dcsconfig/data directory.

   ```
   cd ~/release-4_1/dcsconfig/examples
   cp BL_simple1.config ../data/
   cp BL_simple1.dat ../data/
   ```

2. Configure the MyAuthServer for authentication:

   The users.txt file in the MyAuthServers/examples directory has one entry for a tigerw account with password birdie. Refer to Section 3.2 to add an account for yourself.

3. Start the MyAuthServer, referencing the users.txt file and the config file from the previous step.

   ```
   cd ~/release-4_1/MyAuthServer/linux/
   ./MyAuthServer ../../dcsconfig/data/BL_simple1.config ../examples/users.txt
   ```

4. From another shell, convert the device definition file to a memory map file for DCSS:

   ```
   cd ~/release-4_1/dcss/linux
   cp ~/release-4_1/dcsconfig/examples/BL_simple1.dat .
   ./dcss -r BL_simple1.dat
   ```

   You should see a final message:

   A total of 73 devices were read in from the dump file.

5. Set the TCLLIBPATH for DCSS.

   DCSS needs the TCLLIBPATH environment variable set to the DcsWidgets directory:

   Example:

   setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/scottm/release-4_1/Dc

6. Start DCSS.

   ```
   ./dcss -s
   ```

7. From another shell, start the simulated DHS for motors and ion chambers.

   The simdhs project needs the `TCLLIBPATH` environment variable set.

   Example:

   ```
   setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/scottm/relea
   ```

   Now start the application.

   ```
   cd ~/release-4_1/simdhs/scripts
   ./simdhs.tcl BL_simple1
   ```

8. Edit the `BL_simple1.config` configuration file in the `dcsconfig/data` directory.

   ```
   vi ~/release-4_1/dcsconfig/data/BL_simple1.config
   ```

9. Replace the parameter defined by `simdetector.imageDir` to reference a directory with some image files to be used by the simulated detector.

   ```
   simdetector.imageDir=/data/scottm
   ```

10. Save the config file.

11. From another shell, start the simulated detector DHS.

    ```
    cd ~/release-4_1/simdetector/linux
    ./simdetector BL_simple1
    ```

12. From another shell, start the diffraction image viewer.

    ```
    cd ~/release-4_1/imgsrv/linux
    ./imgsrv ../../dcsconfig/data/BL_simple1.config
    ```

13. From another shell, start Blu-Ice.

    The `TCLLIBATH` environment variable should point to the `BluIceWidgets` and `DcsWidgets` directories and the directory path for the BWidget package. On linux this may be in /usr/local/lib.

    Example:

    ```
    setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/scottm/release-
    ```

    Note: If you set the TCLLIBPATH variable to nothing (e.g. `setenv TCLLIBPATH ""`), blu-ice will make a suggestion to how the variable should be set.

    ```
    cd ~/release-4_1/BluIceWidgets/
    ./bluice.tcl BL_simple1
    ```

    Note: If the GUI does not open, try it again in developer mode to see if it can run as a stripped down application:

    ```
    ./bluice.tcl BL_simple1 developer
    ```

14. Enter the username (e.g `tigerw`) and password (e.g. `birdie`).

3 Configuration

3.1 Configuring the dcsconfig files

The `dcsconfig` project reduces the need to have separately defined configuration files for each of the programs in the DCS framework. It replaces the mysql database configuration used in previous releases.

The `dcsconfig/data/default.config` file should be used to specify system configurations which are the same for all of your beam lines. For example, this file is used to define the hostname and port for the program responsible for authentication, which can be a single instance running for several beam lines.

Beam line specific files should be generated for each beam line and placed within the `dcsconfig/data` directory as well. The names of these beam line specific file should be based on the beamline name. (e.g *BL9-2*`.config`). Any parameter defined in the beamline specific file will override any identical parameter defined in the `default.config` file.

As an example, this architecture allows both a server and client program to read the same configuration file to obtain the listening port of a server. The server will read the file to know which port to listen on, while the client will read the same file get the host and port to connect to. If the client and server programs reside on different machines, it is important that the local copy of the config files are the same. At SSRL, CVS is used to make sure that the distributed computers at a beam line have the latest versions of the configuration files.

3.2 Configuring MyAuthServer

The authentication server at SSRL is integrated with the web server applications. We have provided `MyAuthServer`, a simple stand-alone replacement application which you may find easier to configure and install than a full web server based application.

First, edit the `dcsconfig/data/default.config` or `dcsconfig/data/beamline.config` file and modify the following tags to indicate the listening port and host for the authentication server:

```
auth.host=localhost
auth.port=17000
auth.secureHost=localhost
auth.securePort=17001
```

Currently, MyAuthServer does not open a secure listening channel, secureHost and securePort will not affect MyAuthServer.

In the `MyAuthServer/src` directory, edit the users.txt file and add a new line for each user that should be able to access your beam line. A line with starting with a "#" is treated as a comment line.

An entry for user should be in the following generic format:
`login,name,phone,title,beamlines,staff,roaming,enabled,sessionID,password`
where

- The *`login`* is the unix login account name for the user.
- The *`name`* is the user's full name that will be shown in the Blu-Ice user's tab when the user connects to dcss.

- The *phone* is not really used anymore.
- The *beamlines* parameter indicates which beam line the user is allowed to connect to. If a beamline of `ALL` is used in the field, the user will be able to connect to all beamlines.
- The *staff* parameter should be either `TRUE` or `FALSE`. If `TRUE`, this parameter will enable access to the Setup Tab and provide access to motors which are configured as "staff only".
- The *roaming* parameter should be either `TRUE` or `FALSE`. If `TRUE`, this parameter will enable the user to become "active" while running Blu-Ice from a remote console.
- The *sessionID* parameter is an alpha-numeric string value that is returned to the client after a successful login. Programs that wish to validate that the user will check with the MyAuthClient to make sure that the sessionID and the username match. Note: The SessionID's must be unique for each user. Please change the sessionID from the example given in the users.txt file.
- The *password* parameter is a base64 encoding of the "username:password". It would be a good idea to keep this password different from the user's unix login password.

  Here is an example of how to generate a base64 encoded password using the TCL shell:

  ```
  >tclsh
  % package require base64
  2.2.1
  % ::base64::encode tigerw:birdie
  dGlnZXJ3OmJpcmRpZQ==
  ```

3.3 Configuring the Diffraction Image Server

Edit the `dcsconfig/data/default.config` and change the following flags as desired:

```
# image server
imgsrv.host=foo.slac.stanford.edu
```

```
imgsrv.guiPort=14005
imgsrv.webPort=14006
imgsrv.httpPort=14007
imgsrv.tmpDir=/home/webstaff/jpegscratch
imgsrv.maxIdleTime=60
imgsrv.logStdout=true
imgsrv.logUdpHost=
imgsrv.logUdpPort=
imgsrv.logFilePattern=./imgsrv_log_%d.log
imgsrv.logFileSize=31457280
imgsrv.logFileMax=1
imgsrv.logLevel=ALL
imgsrv.logLibs=
```

imgsrv.guiPort is the port number which Blu-Ice will connect to request JPEG images when using the C library for faster displaying.

imgsrv.httpPort is the port number which Blu-Ice will connect to request JPEG images when the C library is unavailable.

webPort is the port for the web based software to request images

tempImageDirectory is the directory in which the web based software can write temporary JPEG images.

The `imgsrv.host` tag should match the hostname for the machine that the image server will run on. Blu-Ice will use this tag to know the location of the image server.

3.4 Configuring DCSS

DCSS is the core of the DCS system. The 'dcss/src' directory contains 'C' code which acts mostly as a message router. This code also maintains a memory mapped file (i.e. the `database.dat` file), where copies of the motor positions and general configuration of the beam line devices are stored.

The `dcss/scripts/engine` directory is where the TCL 'scripting engine' code is located. The `dcss/src` directory contains the C code which handles message routing, authentication, and other core features of the system. The `dcss/scripts/engine` director is where the code for the scripting engine is located. The `dcss/scripts/devices` directory is the directory where a beamline scientist will write tcl code for handling 'scripted devices'. The 'dcss/scripts/operations' directory is where the scripts for automation and complex sequencing are placed.

The following subsections describe the steps for configuring a simple beam line.

3.4.1 Configuring DCSS's listening ports.

Edit a beamline specific config file in the `dcsconfig/data` directory and configure the following flags as desired:

```
dcss.host=foo.slac.stanford.edu
dcss.scriptPort=14244
dcss.hardwarePort=14242
dcss.guiPort=14243
```

The `dcss.host` tag should match the hostname for the machine that DCSS will run on. This will allow other programs, such as Blu-Ice to know what host to connect to.

`dcss.scripPort` is the port number which the scripting engine connects to. The *`dcss.hardwarePort`* for all of the hardware clients (DHS's) to connect to. The *`dcss.guiPort`* is the port number for which all of the gui clients connect to.

3.4.2 Configuring DCSS's Authentication.

Set the authentication protocol to 2. This tag is read by the Blu-Ice clients to allow them to know how they should authenticate with dcss. Authentication protocol 2 is refering to the new sessionID based authentication.

```
dcss.authProtocol=2
```

The `dcss.validationRate` tag tells dcss how often, in milli-seconds, it should query the authentication server to determine if a user's permissions have changed.

```
dcss.validationRate=30000
```

3.4.3 Restricting Remote Access

DCSS can use the Blu-Ice's host DISPLAY variable to restrict a user's capabilities based on the user's location and the state of the hutch door. Create a new row with the following format for each display that should be recognized by dcss.

```
dcss.display=category hostname display [#Description]
```

where

- The *category* can be either 'hutch', 'local', or 'remote'.
- The *hostname* is the X-windows HOST envionment variable for the machine that Blu-Ice is displaying on. This field is read but not used if the *display* variable starts with something other than a ':'.
- The *display* is the X-windows DISPLAY variable for the console that Blu-Ice is displaying on. If the display starts with a ':', then the *hostname* is used to determine the Blu-Ice client's location.
- The line may optionally add a comment describing the location of the terminal. This comment is broadcast to all of the Blu-Ice clients, further clarifying the location of the GUI.

The `dcss.forcedDoor` tag is used by dcss to force the hutch door to a particular state. This is useful for beam lines that do not have a dhs that can report the current state of the hutch door. Leaving this tag empty will require a DHS to report the state of the door.

```
dcss.forcedDoor=
```

It is recommended to set this value to "closed" when a DHS is unavailable to report the state. However, this will allow people to move motors remotely regardless of the real state of the hutch door.

```
dcss.forcedDoor=closed
```

For testing, it is also possible to force this state to "open".

```
dcss.forcedDoor=open
```

3.4.4 Configuring DCSS's logging style

The method, location, and style of the logging by dcss can be specified with the following tags:

```
dcss.logStdout=true
dcss.logUdpHost=
dcss.logUdpPort=
dcss.logFilePattern=./bl92_dcss_log_%g_%u.log
dcss.logFileSize=31457280
dcss.logFileMax=20
dcss.logLevel=ALL
dcss.logLibs=auth_client|http_cpp
```

3.4.5 Quick introduction to the database.dat file

DCSS requires a memory mapped file containing the list of beamline devices and current state of these devices. This file is currently always named `database.dat`. Refer to Section 5 for a complete reference on how to create and use this file. If you are just getting started and want to play with a simulation, an example database is available in text format. To use this file, follow the these steps:

1. Copy an example dump file (i.e ending with .dat) from the `dcsconfig/data` directory.

   ```
   cp ./dcsconfig/BL9-1.dat ./dcss/linux
   ```

2. Using a text editor, edit the DHS entries to match the hostnames of the computers running DHS's on the new beam line. For example, the third line of a DHS entry may need to change to reflect the name of a DHS running on a different computer. DCSS will reject a DHS connection from a hardware host that is coming from a host different from what is specified in this file.

   Example:

   ```
   simDhs
   3
   localhost 2
   ```

3. Generate the memory map file (i.e. `database.dat`) using the ./dcss -r command.

   Note: This step will completely overwrite an existing database.dat file, which usually contains current motor positions.

```
cd dcss/linux
./dcss -r example_bl111sim_dump.txt

....<OUTPUT REMOVED>.........

A total of 171 devices were read in from the dump file.
```

3.5 Configuring your impersonation server

The impersonation server will take a sessionId and compare it against the authentication server. A valid sessionID will result in the impersonation switching the process to be owned by the user and allow specific commands be executed as the user. Installation instructions for the impersonation server are located in the `imperson_cpp/doc/installation.txt` file.

The impersonation server is used by the "fluorescence scans", "motor scans", and "image server".

3.6 Configuring your crystal screening server

Coming soon...

3.7 Configuring "enhanced" Blu-Ice

Edit your beamline specific file in the `/dcsconfig/data` directory and modify the following tag:

`bluice.deviceDefinitionFilename`

This tag should reference a text file with the same format as a dump of the memory mapped file produced by DCSS. This text file will be used as a default configuration for Blu-Ice at initialization. The device permissions found in this file will be used by Blu-Ice and every device in the file will appear in the drop down menu on the "Setup Tab".

Additionally, the following tags can be modified to point at different incr widgets for displaying an alternate view of your optics.

```
bluice.mirrorView=DownwardMirrorView
bluice.monoView=SingleCrystalHorizFocusMonoView
```

If you want to add a different graphical view for the optics at your beam line, you can add a incr widget class to the DcsWidgets code and change these tags to use your new class' name.

Modify the `bluice.defaultDataDir` tag to set the directory that Blu-Ice will use to by default for data collection. The username will be appended to the defined value. In the following example, Blu-Ice will collect data by default into the /data/*username* directory.

```
bluice.defaultDataDir=/data
```

```
bluice.beamlineView=detectorPosition goniometer table frontEndSlits frontEndApertu
bluice.beamlineView=hutchOverview
```

The existence and order of tabs in Blu-Ice can be defined with the `tabOrder` option as defined in the `default.config` file:

```
bluice.tabOrder=Hutch Collect Screening Scan Users Setup
```

For example, to remove the screening tab from blu-ice, add the following line to the beamline specific config file in order to over ride the default file:

```
bluice.tabOrder=Hutch Collect Scan Users Setup
```

Note: As a minimum, blu-ice needs the Hutch, Users, and Setup tab.

If you want to add a different graphical view for the hutch view of your beam line, you can add a incr widget class to the DcsWidgets code and change these `hutchView` tag to use your new class' name. The default view defined in the `default.config` file is shown here:

```
bluice.hutchView=DCS::HutchOverview
```

To override this view, redefine it in the beamline specific file like this:

```
bluice.hutchView=MyNewHutchView
```

where `MyNewHutchView` is the name of your incr widget class containing the hutch view.

3.8 Configuring "original" Blu-Ice

The original project is in the `blu-ice` CVS project. This project represents a fork in the Blu-Ice development. New features will probably be added to the new Blu-Ice project as it is easier to maintain and configure. There are various reasons why you may want to use this version, so the configuration instructions are provided.

(a) Modify `blu-ice/scripts/ice.tcl` to reflect your installation.

- Change the `DCS_DIR` variable to point to your DCS root directory. Production level software at SSRL runs out of the `/usr/local/dcs/` directory.
  For example:
  ```
  < set DCS_DIR "/usr/local/dcs/"
  ---
  > set DCS_DIR "/home/scottm/release-3_2/"
  ```
- Modify the location of the `tcl_clibs.so` file to point at the appropriate build directory. By default, it is pointing at the irix build directory.
  ```
  < load $DCS_DIR/tcl_clibs/irix/tcl_clibs.so dcs_c_library
  ---
  > load $DCS_DIR/tcl_clibs/linux/tcl_clibs.so dcs_c_library
  ```
- Modify ice.tcl to connect to your DCSS server.
  The Blu-Ice program is started with 1 argument – the name of the beam line that it needs to connect to. This argument is used within a large `switch` statement to initialize the Blu-Ice configuration.

You will need to modify this `switch` statement either by changing an existing beam line configuration or by adding a new configuration. Some important parameters within this initialization are:

- `gBeamline(serverName)` hostname location of the DCSS server.
- `gBeamline(serverPort)` port on which Blu-Ice should attempt to connect to DCSS. This should correspond to the third port number entry in the serverPorts.txt file in DCSS's configuration.
- `gBeamline(title)` sets the text of the window's frame at the top of the Blu-Ice.
- `gBeamline(beamlineId)` used in various switch statements throught blu-ice.
- `gBeamline(simulation)` changes the color of Blu-Ice only.
- `gBeamline(detector)` lets Blu-Ice display appropiate detector modes and pictures
- `gBeamline(videoServerUrl)` various web address of Axis camera servers.
- `gBeamline(hutchVideoPath)`
- `gBeamline(sampleVideoPath)`
- `gBeamline(ptzPath)` web location of pan-tilt-zoom preset information

(b) In `blu-ice/data/` copy one of the existing files to the same directory using a new name:

`periodic_`*`beamlineID`*`.dat`

where *`beamlineID`*`.dat` is defined by the `gBeamline(beamlineId)` variable. Modify the new file to describe your current beam line's capabilities.

Example:

```
smblx5:~/release-3_2/blu-ice/data > cp periodic_bl92.dat periodic_bl92sim.dat
```

(c) Modify the `blu-ice/scripts/dice_tabs.tcl` file and change the following line to reflect a connection to your own Diffraction Image Server set up in Section 4.5.

```
blctl92:~/release-3_2/blu-ice/scripts > cvs diff
cvs diff: Diffing .
```

```
Index: dice_tabs.tcl
===================================================================
RCS file: /home/code/repository/blu-ice/scripts/dice_tabs.tcl,v
retrieving revision 1.39
diff -r1.39 dice_tabs.tcl
136c136
<  Diffimage lastImage $gWindows(collect,diffimage,frame) 134.79.31.20 1
---
>  Diffimage lastImage $gWindows(collect,diffimage,frame) localhost 1400
```

The 14001 in the above example should correspond to *guiPort* which is specified on the command line when starting the Diffraction Image Server.

3.9 Configuring the detector simulator

The following three parameters should be included in the beamline specific config file.

```
simdetector.name=detector
simdetector.imageDir=/data/scottm
simdetector.imageFilter=*.img
```

The above example configuration will allow the DHS to connect to dcss and announce itself as the 'detector' DHS program. It will use the `*.img` filter to select the files to copy from the `/data/joeuser` directory into the directory specified during data collection.

3.10 Configuring SSRL's DHS

SSRL uses a DHS program which supports the following hardware:

- DMC2180 motion controller.
- ADSC Quantum 4 CCD detector.
- ADSC Quantum Q315 CCD detector.
- MAR345 detector without a base.
- MARCCD family of detectors.

- Image analysis of JPEG's obtained by AXIS 2400 servers. (for automatic crystal centering)

(a) Check out the Q4 image transformation files found in the xform project. These files contain some algorithms and code which Area Detector Systems Corporation[2] does not wish to have distributed without prior permission.

```
blctlxx:~/release-4_1 > cvs checkout -r release-4_1 xform
cvs checkout: Updating xform
U xform/xform.c
U xform/xform.h
blctlxx:~/release-4_1 >
```

If you do not have access to this project, the `xformstub.c` and `xformstub.h` (provided by SSRL) can be used to build the DHS. Create the xform directory in your 'DCS root' directory before copying and renaming the follow the files as shown in this example:

```
blctlxx:~/release-4_1 > mkdir xform
blctlxx:~/release-4_1 > cd xform
blctlxx:~/release-4_1/xform > cp ~/release-4_1/dhs/src/xformstub.c xform.c
blctlxx:~/release-4_1/xform > cp ~/release-4_1/dhs/src/xformstub.h xform.h
```

(b) DCSS must be explicitly configured to expect the DHS' connection.

This is done by changing DCSS' `database.dat` file to expect the connection from the new DHS by making a DHS entry as discussed in Section 5.3.3.

3.10.1 Configuring the DHS to Control Dmc2180 Motion Controllers

```
#dhs.instance=instanceName hardwareType logFilePattern memoryMapName autoflush watchc
dhs.instance=galil1 dmc2180 /usr/local/dcs/BL11-1/galil1_%g_%u.log /usr/local/dcs/BL1

dmc2180.control=galil1 bl11ga1 /usr/local/dcs/dhs/galil_scripts/script5.txt blctl111p
```

[2] http://www.adsc-xray.com/

```
#           =motor_name channel P I D
galil1.servo=sample_x a 3291 221 32
galil1.servo=sample_y b 3291 221 32
galil1.servo=sample_z c 3291 221 32
galil1.servo=camera_zoom d 3291 221 32
galil1.stepper=gonio_kappa e
galil1.stepper=gonio_phi f
galil1.stepper=gonio_omega g
galil1.stepper=gonio_z h

#             =shutter_name channel voltage_low_state
galil1.shutter=shutter 1 closed
galil1.hutchDoorBitChannel=3
```

3.10.2 Configuring the DHS to Control a Quantum 4 Area Detector

```
dhs.instance=detector quantum4 /usr/local/dcs/BL11-1/detector_%g_%u.log /usr
quantum4.hostname=bl15ccd
quantum4.dataPort=8042
quantum4.commandPort=8041
quantum4.beamCenter=94.0 94.0
quantum4.nonUniformitySlowFile=/usr/local/dcs/ccd_411/NONUNF_slow
quantum4.nonUniformityFastFile=/usr/local/dcs/ccd_411/NONUNF_fast
quantum4.nonUniformitySlowBinFile=/usr/local/dcs/ccd_411/NONUNF_2x2
quantum4.nonUniformityFastBinFile=/usr/local/dcs/ccd_411/NONUNF_2x2
quantum4.distortionSlowFile=/usr/local/dcs/ccd_411/CALFIL
quantum4.distortionFastFile=/usr/local/dcs/ccd_411/CALFIL
quantum4.distortionSlowBinFile=/usr/local/dcs/ccd_411/CALFIL_2x2
quantum4.distortionFastBinFile=/usr/local/dcs/ccd_411/CALFIL_2x2
quantum4.postNonUniformitySlowFile=/usr/local/dcs/ccd_411/POSTNUF_slow
quantum4.postNonUniformityFastFile=/usr/local/dcs/ccd_411/POSTNUF_fast
quantum4.postNonUniformitySlowBinFile=/usr/local/dcs/ccd_411/POSTNUF_2x2
quantum4.postNonUniformityFastBinFile=/usr/local/dcs/ccd_411/POSTNUF_2x2
quantum4.darkDirectory=/usr/local/dcs/darkimages
#quantum4.darkRefreshTime=7200
#quantum4.darkExposureTolerance=0.10
```

```
#quantum4.writeRawImages=n
```

3.10.3 Configuring the DHS to Control a Quantum 315 Area Detector

```
dhs.instance=detector quantum315 /usr/local/dcs/BL11-1/detector_%g_%u.log /usr/local/
quantum315.commandHostname=blctl111
quantum315.commandPort=8041
quantum315.dataHostnameList=q315-902-01 q315-902-02 q315-902-03 q315-902-04 q315-902-
quantum315.dataPortList=9042 9042 9042 9042 9042 9042 9042 9042 9042
quantum315.beamCenter=157.5 157.5
quantum315.darkRefreshTime=7200
quantum315.darkExposureTolerance=0.10
quantum315.writeRawImages=n
```

3.10.4 Configuring the DHS to Control a MAR345 Area Detector

```
dhs.instance=detector mar345 /usr/local/dcs/BL9-3/detector_%g_%u.log /usr/local/dcs/B
mar345.commandDirectory=/logs/root
```

3.10.5 Configuring the DHS to Control a MAR CCD Area Detector

```
dhs.instance=detector marccd /usr/local/log/detector_%g_%u.log ./detector.dat 500 100
marccd.beamCenter=94.0 94.0
marccd.serialNumber=1
marccd.hostname=bl42marpc
marccd.commandPort=3000
marccd.darkRefreshTime=7200
marccd.darkExposureTolerance=.10
marccd.writeRawImages=F
```

3.10.6 Configuring the DHS to Perform Loop Analysis of Axis 2400 Camera View

```
dhs.instance=camera axis2400 /usr/local/dcs/BL9-2/camera_%g_%u.log /usr/local/dcs/BL9
axis2400.hostname=bl92aaxis
axis2400.port=8000
```

```
axis2400.passwordFile=/usr/local/dcs/BL9-2/axis2400Password.txt
axis2400.url_path=/axis-cgi/jpg/image.cgi?camera=2&clock=0&date=0&text=0 HTT
```

4 Starting and Stopping programs

Within the `batchbuild` project, there is a script named `restart_dcs`. This script is useful for starting all of the dcs programs on one machine. Individual programs can be started and stopped individually as shown in the following subsections.

4.1 Starting and Stopping DCSS

To start this program, it is necessary to first set your TCLLIBPATH to the BluIceWidgets and DcsWidgets directory.

For example: `setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/s`

A fully configured dcss should be started in the dcss build directory.

`~/release-4_1/dcss/linux > ./dcss -s`

There are no special procedures for stopping DCSS. Control-c at the terminal or the unix 'kill -9' command are the recommended ways of stopping dcss. Of course it is advisable to first make sure that there are no moving motors before shutting down DCSS.

4.2 Starting the hardware simulator

The hardware simulator is a TCL script which will connect to DCSS and announce itself as the 'simdhs' hardware server. All entries in the DCCS `database.dat` file that are controlled by the 'simdhs' hardware server will be simulated.

The script will simulate motors, shutters, and ion chambers.

To start this program, it is necessary to set your TCLLIBPATH to the BluIceWidgets and DcsWidgets directory.

For example:

`setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/scottm/release`

Start the program with the name of the beam line to simulate. This will automatically load the configuration file and connect to the appropriate DCSS.

`~/release-4_1/simdhs/scripts > ./simdhs.tcl BL9-1`

4.3 Starting "enhanced" Blu-Ice

To start the Blu-Ice GUI, first set your TCLLIBPATH to the BluIceWidgets and DcsWidgets directory.

For example:

`setenv TCLLIBPATH "/home/scottm/release-4_1/BluIceWidgets /home/scottm/release-4_1/DcsW`

Start the program by executing the following script with arguments as shown:

`BluIceWidgets/bluice.tcl` *`beamline [style]`*

where `beamline` is the name of the beamline that must match a file defined in the dcsconfig directory. The `style` is optional and defaults to classic mode. Other options for `style` include `developer` and `videoOnly`. The `developer` mode starts the GUI with only the "Setup Tab" for easy development of new widgets.

Some examples:

```
~/release-4_1/simdhs/scripts > ./simdhs.tcl BL9-1
~/release-4_1/simdhs/scripts > ./simdhs.tcl BL9-1 developer
```

4.4 Starting "original" Blu-Ice

To start the original Blu-Ice code, configure your TCLLIBPATH environment variable to grab the widget libraries.

For example:

`setenv TCLLIBPATH "/home/scottm/release-4_1/widgets"`

Start Blu-Ice with the name of the beam line that you wish to connect to.

`~/release-4_1/blu-ice/scripts/ice.tcl` *`beamlineName`*

where the *`beamlineName`* will be used in the `switch` statement within the `ice.tcl` file to configure the `Blu-Ice`.

4.5 Starting the Diffraction Image Server

Start the Diffraction Image Server in the build directory. The first argument is the configuration file where the image server tags are defined. For example

`./imgsrv ../../dcsconfig/data/default.config`

4.6 Starting the MyAuthServer for authentication

To start the MyAuthServer reference the `users.txt` file and the config file.

```
cd ~/release-4_1/MyAuthServer/linux/
```

```
./MyAuthServer ../../dcsconfig/data/default.config ../examples/users.txt
```

5 Maintaining the `database.dat` file

A properly configured DCSS requires a `database.dat` file.

The `database.dat` file contains the following information:

- List of all DHS programs allowed to connect to DCSS.
- List of all devices controlled by DCSS.
- Each device's last known position and/or state.
- Name of the DHS controlling each device.
- All other dynamic information regarding the device (e.g. software limits, scale factors, etc)
- Run definitions.

The `database.dat` file is memory mapped, allowing the operating system to keep the file as up-to-date as possible. Because the file is memory mapped, it is not possible to edit the file directly with a text editor. Instead, DCSS has the ability to transform the `database.dat` file into a text file. The text file can be edited and DCSS can create a new `database.dat` file from the modified file.

5.1 Dumping the `database.dat` file

To convert the contents of the `database.dat` file to text:

1. Stop DCSS.
2. `cd ~/release-3_2/dcss/linux`
3. `./dcss -d` *`fileName`*
4. The file *filename* should now contain the contents of the `database.dat` in text format.

5.2 Recreating the `database.dat` file

Performing this function is an easy way to *LOSE ALL OF YOUR MOTOR POSITIONS!*

To convert the contents of properly formatted text file into a `database.dat` file:

1. Stop DCSS and/or make sure that no DCSS is running.
2. `ps -ef | grep dcss`

   Verify that DCSS is not running.
3. `cd ~/release-3_2/dcss/linux`
4. `./dcss -r` *`fileName`*

   where *fileName* is the name of a properly formatted configuration file.
5. The `database.dat` file has now been created with the new configuration.

5.3 Format of the `database.dat` file

The text file accepted by the `dcss -r` command is a list of entries. Each entry is separated by a blank line (carriage return).

Each entry consists 3 or 4 lines of data describing the entry.

1. The first line of an entry is the *`device name`* as known by DCSS.

   The *`device name`* may consist of regular alpha/numerical characters and the underscore character.
2. The second line is an integer representing the *`device type`* as shown in Table 1.
3. The contents of the remaining lines depend upon the entry for the *`device type`* as discussed in the following subsections.

5.3.1 The Real Motor Entry

A real motor is a motor controlled by a hardware DHS.

An entry for a Real Motor definition should be in the following generic format:

`Line 1:`*`motorName`*
`Line 2:1`

Device type	description
1	Real Motor
2	Combo Motor
3	DHS description
4	Ion chamber
5	*Obsolete*
6	Shutter/Filter
7	*Obsolete*
8	Run Definition
9	Runs Definition
10	*Obsolete*
11	Operation
12	Encoder
13	String

Table 1: Device Types in `database.dat`

```
Line 3:responsibleDHS externalMotorName
Line 4:position upperLimit lowerLimit scaleFactor speed acceleration
backlash lowerLimitOn upperLimitOn motorLockOn backlashOn reverseOn
circleMode units
Line 5:0
Line 6:passiveOk remoteOk localOk inHutchOk closedHutchOk
Line 7:passiveOk remoteOk localOk inHutchOk closedHutchOk
```

- The *responsibleDHS* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.
- The *externalMotorName* parameter can be the same name as the *motorName*, but is provided here for mapping the motor name to a DHS(s) that names the motor differently. This is common with the motors controlled by the ICS system.
- The *position* parameter is the current position of the motor.
- The *upperLimit* and *lowerLimit* parameters are the software limits for the motor. The limits are only used if the corresponding *upperLimitOn* and *lowerLimitOn* flags are enabled.
- The *scaleFactor* converts motor steps to the default units of the motor.

- The *speed* parameter is in units of steps/second.
- The *acceleration* parameter is in units of milli-seconds.
- The *backlash* parameter is in steps.
- The *lowerLimitOn* parameter can be 0 (for no software limit) or 1 (to enable the *lowerLimit* parameter).
- The *upperLimitOn* parameter can be 0 (for no software limit) or 1 (to enable the *upperLimit* parameter).
- The *motorLockOn* parameter can be 0 (which allows the motor to move) or 1 (prevents the motor from moving.)
- The *backlashOn* parameter can be 0 (no backlash) or 1 (uses the *backlash* parameter for each move.)
- The *reverseOn* parameter can be 0 (no internal swapping of direction) or 1 (internally swaps the direction of the motor move.)
- The *circleMode* parameter can be 0 (linear motion) or 1 (automatically converts 360 degrees to 0, and automatically picks the shortest path of motion.)
- The two permissions lines (lines 6 and 7) are discussed in Section 5.3.11.
- *units* is

  ```
  mm|deg|eV|counts
  ```

Here is an example for `table_vert_1` motor entry. It is is motor controlled by a DHS called `gi`. The DHS `gi` knows this motor by a different name `tablev1`.

```
table_vert_1
1
gi tablev1
23.099118 49.999984 0.000000 3145.921000 500 125 1573 0 0 0 1 0 0 mm
0
0 1 1 1 1
0 1 1 1 1
```

5.3.2 The Pseudo Motor Entry

A pseudo motor is device that can be moved like a regular motor, but does not necessarily control a single real motor. The pseudo motor may actually move several motors in combination, or perform additional functions and checks while moving the motor. A pseudo motor controlled by the scripting engine is called a scripted device.

An entry for a Pseudo Motor definition should be in the following generic format:

`Line 1:`*`motorName`*
`Line 2:2`
`Line 3:`*`responsibleDHS externalMotorName`*
`Line 4:`*`position upperLimit lowerLimit lowerLimitOn upperLimitOn motorLockOn circleMode units`*
`Line 5:0`
`Line 6:0`
`Line 7:`*`passiveOk remoteOk localOk inHutchOk closedHutchOk`*
`Line 8:`*`passiveOk remoteOk localOk inHutchOk closedHutchOk`*

- The *`responsibleDHS`* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.

- The *`externalMotorName`* parameter can be the same name as the *`motorName`*, but is provided here for mapping the motor name to a DHS(s) that names the motor differently. This is common with the motors controlled by the ICS system.

  This parameter can also be used by the scripting engine to locate a specialized script for the motor. For example, using `standardVirtualMotor` would indicate to the `self` dhs that it should use the specialized `standardVirtualMotor` template when working with this motor.

- The *`position`* parameter is the current position of the motor.

- The *`upperLimit`* and *`lowerLimit`* parameters are the software limits for the motor. The limits are only used if the corresponding *`upperLimitOn`* and *`lowerLimitOn`* flags are enabled.

- The *`lowerLimitOn`* parameter can be 0 (for no software limit) or 1 (to enable the *`lowerLimit`* parameter).

- The *`upperLimitOn`* parameter can be 0 (for no software limit) or 1 (to enable the *`upperLimit`* parameter).

- The *motorLockOn* parameter can be 0 (which allows the motor to move) or 1 (prevents the motor from moving.)
- The *circleMode* should be always be 0. See bug 39[3].
- The two permissions lines (lines 7 and 8) are discussed in Section 5.3.11.
- *units* is

  `mm|deg|eV|counts`

5.3.3 The DHS definition

An entry for a complete DHS definition should be in the following generic format:

```
Line 1:name
Line 2:3
Line 3:hostName protocolLevel
```

- The *hostName* parameter is the host name of the computer that DHS will be running on. DCSS will reject a connection from a DHS announcing itself with the *name* if the connection is not coming from the computer/hostname *hostName*.
- The *protocolLevel* can be a 1 or a 2. A protocol of 1 is used by DHS(s) that are still using the 200 byte DCS protocol. A protocol of 2 is used by DHS(S) that have been upgraded to the dynamic length DCS protocol.

Here is an example for a `camera` DHS entry:

```
camera
3
biotest.slac.stanford.edu 2
```

Note: Every `database.dat` file should have an entry for the scripting engine DHS as shown here:

[3]https://smb.slac.stanford.edu/bugzilla/long_list.cgi?buglist=39

```
self
3
localhost 2
```

This will define a `self` DHS which `scripted operations` and `scripted devices` should use.

5.3.4 The Ion Chamber Entry

This entry maps very closely to the ICS control system's way of defining a ion chamber.

`Line 1:`*`ionchamberName`*
`Line 2:4`
`Line 3:`*`responsibleDHS counter counterChannel timer timerType`*

- The *`responsibleDHS`* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.
- The *`counter`* is the name of a counter in ICS.
- The *`counterChannel`* is the counter's channel in ICS.
- The *`timer`* is ?
- The *`timerType`* is ?

Here is an example ion chamber entry:

```
i_sample
4
simDhs hex1 3 rtc1 clock
```

5.3.5 The Shutter Entry

This entry can be used for devices that have binary state (e.g. shutters and filters)

`Line 1:`*`shutterName`*
`Line 2:6`
`Line 3:`*`responsibleDHS shutterState`*

- The *`responsibleDHS`* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.

- The *shutterState* is either a 0 (open) or a 1 (closed).

Here is an example shutter entry:

```
Al_4
6
galil2 0
```

5.3.6 The Run Definition Entry

There are 17 run definitions allocated in the `database.dat` file for the 17 possible runs that can be defined in the Collect tab in BLU-ICE.

An entry for a Run definition should be in the following generic format:

`Line 1:`*runName*
`Line 2:8`
`Line 3:`*runStatus nextFrame runLabel fileroot directory startFrameLabel axisMotor startAngle endAngle delta wedgeSize exposureTime distance numEnergy energy1 energy2 energy3 energy4 energy5 modeIndex inverseOn*

- The *runName* is usually `run0` through `run16`.
- The *runStatus* can be `collecting`, `paused` or `inactive`. If DCSS is stopped, a run with a status of `collecting` should be manually changed to `inactive` or `paused`.
- The *nextFrame* parameter is a positive integer that indicates how much of the run has been collected.
- The *runLabel* parameter is the text that appears on the tab. It also is used to generate the filename.
- The *fileroot* parameter is the text that is used as the root for the filename.
- The *directory* parameter is the directory that the user wishes to write the files into.
- The *startFrameLabel* parameter is a numerical value that is used as a starting point to generate a unique file name per frame.
- The *axisMotor* parameter is the motor name that will used in the oscillation code. Usually this will be either `gonio_phi` or `gonio_omega`.

- The `startAngle` parameter is the starting angle of the `axisMotor` for the run definition.
- The `endAngle` parameter is the final ending angle for the `axisMotor`.
- The `delta` parameter is the distance that the `axisMotor` will travel per frame.
- The `wedgeSize` parameter is the angular distance that the `axisMotor` must travel before rotating 180 degrees and collecting the frames in the inverse wedge.
- The `exposureTime` parameter is the amount of time each frame will be exposed. This value is the requested time, and is not the dose corrected time.
- The `distance` parameter is the position for `detector_z` for each frame in the run.
- The `numEnergy` parameter is number of energy values that will be used during the collection of the run. This parameter should never exceed 5, as there are only 5 energy values that may be defined in a single run. A zero in this position would also be bad.
- The `energy1...energy5` parameters are the values for energy that will be used.
- The `modeIndex` parameter is the detector mode in an integer format. This parameter can have different meanings depending on the detector type being used. For example, detector mode 0 for a MAR345 is different than detector mode 0 for a CCD.
- The `inverseOn` parameter is a boolean value (0=FALSE or 1=TRUE) indicating whether or not the inverse wedge should be collected or not.

5.3.7 The Runs Definition entry

The Runs Definition entry is used to store information regarding data collection that will apply to all defined runs. There can only be one one Runs Definition in a `database.dat` file.

An entry for a Runs definition should be in the following generic format:

```
Line 1:runs
Line 2:9
```

`Line 3:`*`runCount currentRun isActive doseMode`*

- The *`runCount`* parameter indicates how many of the run definitions are valid. For example, a value of 4 would indicate that run0, run1, run2, and run3 device entries are valid. All other run definitions are not valid.
- The *`currentRun`* parameter is no longer used.
- The *`isActive`* parameter is no longer used.
- The *`doseMode`* is a Boolean value indicating whether or not to use a dose corrected exposure time per frame during data collection (0=no dose mode, 1=dose mode).

Here is an example runs entry indicating that two runs are defined and dose mode is disabled:

```
runs
9
2 2 0 0
```

5.3.8 The Operation Entry

An operation is a generic function. It does not have state stored in the database.dat file.

`Line 1:`*`operationName`*
`Line 2:11`
`Line 3:`*`responsibleDHS externalName activeClientOnly`*
`Line 4:`*`passiveOk remoteOk localOk inHutchOk closedHutchOk`*
`Line 5:`*`passiveOk remoteOk localOk inHutchOk closedHutchOk`*

- The *`responsibleDHS`* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.
- The *`externalName`* parameter can be used by the scripting engine to locate a specialized script for the operation. In the case of the scripting engine, it appends a `.tcl` to this parameter to locate the script for the operation.
- The *`activeClientOnly`* parameter can be a 0 to allow any client to request the operation or 1 to allow only the active client to request the operation.

- The two permissions lines (lines 4 and 5) are discussed in Section 5.3.11.

Here is an example operation entry:

```
getLoopTip
11
camera getLoopTip 1
0 0 0 0 0
0 0 0 0 0
```

5.3.9 The Encoder Entry

An encoder entry allows DCSS to know which DHS has access to the encoder. It does not map an encoder to a real motor. The description of a motor's behavior in relationship to its encoder should be done using a scripted device.

`Line 1:`*`encoderName`*
`Line 2:12`
`Line 3:`*`responsibleDHS externalName`*

- The *`responsibleDHS`* must be the name of a DHS defined in separate entry as discussed in Section 5.3.3.
- The *`externalName`* parameter should probably be the same as the *`encoderName`*.

Here is an example of an encoder entry:

```
detector_z_encoder
12
simDhs detector_z_encoder
```

5.3.10 The String Entry

5.3.11 The Device Permissions Bits

The real motor, pseudo motor, and operation definition entries in the database.dat file each have two lines for restricting the use of the device. The two lines each have the same bits, but apply to different types of groups of users. The first line applies to the Blu-Ice user that have 'Staff' permissions. The second line applies to the Blu-Ice clients that do not have 'Staff' permissions.

There are two lines permission entry for each device will look like this:

```
 Line x:  passiveOk remoteOk localOk inHutchOk closedHutchOk
Line x+1:passiveOk remoteOk localOk inHutchOk closedHutchOk
```

1. The *passiveOk* parameter can be 0 or 1. A '0' indicates that the Blu-Ice client requesting the move must be 'Active' before DCSS forwards the request to the reponsible DHS. A '1' in this field would allow DCSS to forward the motor move request to the reponsible DHS even if the requesting Blu-Ice is not 'Active'.

2. The *remoteOk* parameter can be 0 or 1. A '0' indicates that with the hutch door open, the Blu-Ice client must be either running on a 'LOCAL' or 'HUTCH' console before DCSS will forward the move request to the responsible DHS. A '1' in this field would allow the move request to be forwarded to the DHS regardless of where the Blu-Ice client is running, even with the hutch door open.

3. The *localOk* parameter can be 0 or 1. A '0' indicates that with the hutch door open, the Blu-Ice client must be running on a 'HUTCH' console before DCSS will forward the move request to the responsible DHS. A '1' in this field would allow the move request to be forwarded to the DHS if the requesting Blu-Ice client is running on a 'LOCAL' machine, even with the hutch door open.

4. The *closedHutchOk* parameter can be 0 or 1. A '0' indicates that the hutch door must be open before the request to move the motor is forwarded to the responsible DHS. A '1' in this field indicates that the motor move may be moved with the hutch door closed.

Figure 1 shows the possible combinations of permission bits and the expected behaviour of the device depending on whether or not the client is 'REMOTE', 'LOCAL', or in the 'HUTCH', and whether or not the hutch door is closed or open.

6 Writing Scripted Devices and Operations

BLU-ICE and the scripting engine within DCSS are both written in the TCL scripting language. One of the features of a scripting language is that code does not need to be pre-compiled to be executed. This greatly simplifies the writing and testing of scripts. BLU-ICE itself allows the user to

Device Restrictions					door closed		door open		
passiveOk	remoteOk	localOk	inHutchOk	closedHutchOk	REMOTE	LOCAL	REMOTE	LOCAL	HUTCH
x	0	0	0	0	n2	n2	n2	n2	n2
x	0	0	0	1	y	y	n3	n4	n5
x	0	0	1	0	n7	n7	n3	n4	y
x	0	0	1	1	y	y	n3	n4	y
x	0	1	0	0	n7	n7	n3	y	n5
x	0	1	0	1	y	y	n3	y	n5
x	0	1	1	0	n7	n7	n3	y	y
x	0	1	1	1	y	y	n3	y	y
x	1	0	0	0	n7	n7	y	n4	n5
x	1	0	0	1	y	y	y	n4	n5
x	1	0	1	0	n7	n7	y	n4	y
x	1	0	1	1	y	y	y	n4	y
x	1	1	0	0	n7	n7	y	y	n5
x	1	1	0	1	y	y	y	y	n5
x	1	1	1	0	n7	n7	y	y	y
x	1	1	1	1	y	y	y	y	y

1 NOT_ACTIVE_CLIENT
2 NO_PERMISSIONS
3 HUTCH_OPEN_REMOTE
4 HUTCH_OPEN_LOCAL
5 In_HUTCH_RESTRICTED
6 IN_HUTCH_AND_DOOR_CLOSED
7 HUTCH_DOOR_CLOSED

Invalid
Valid

Inconsistent
Consistent

Figure 1: Device Permissions

open a command prompt, which gives the user access to the TCL interpreter running BLU-ICE. New scripts can be loaded from this command prompt and executed, allowing complicated or repetitive tasks to be somewhat automated.

There are several drawbacks to running scripts through the command prompt of BLU-ICE:

- The BLU-ICE must remain open during the duration of the script.
- The BLU-ICE executing the script must remain the active client during the duration of the script.
- Other open BLU-ICE's will not know what the intentions of the script is, and therefore would not be able to interpret intermediate results.
- The BLU-ICE running the script must have full permissions to perform all actions within the script.
- The performance of the script depends on the local machine that is running the instance of BLU-ICE.

An alternative to running a script in BLU-ICE would be to run the script as an operation within the DCSS scripting engine. This feature overcomes all of the restrictions of running a script through BLU-ICE, but it does have several drawbacks of its own:

- The name of the script must be added manually to the `database.dat` file.
- Changes to the script requires DCSS to be shut down and restarted.
- A user's permissions must be checked within the scripting engine before each command is issued, because the scripting engine has unlimited privileges.

In many cases, development of new scripts can be done first through BLU-ICE and ported later to DCSS when the script has been tested sufficiently.

6.1 General DCS Scripting Commands

The scripting engine as well as the BLU-ICE command prompt offer the full command set of the TCL language. In addition to the standard TCL commands, BLU-ICE and the scripting engine add commands for controlling a beam line. These commands hide the details and complexity of the network protocol.

6.1.1 Writing to the log window in BLU-ICE

There are three commands for writing to the log window. All of the commands are written with a time stamp.

- `log_note` *`comments`*

  Outputs a green comment to the log window.

- `log_warning` *`warning`*

  Outputs a brown warning to the log window.

- `log_error` *`error`*

  Outputs a red error message to the log window

6.1.2 Querying a motor position

Motor positions are available by adding a $ in front of the name of the motor.

From the BLU-ICE command prompt, the user may type the following to obtain a motor position:

```
log_note $table_vert
```

Within a script it is necessary to include the following statement at the top of each procedure for each motor of interest: `variable` *`motorName`*.

Example: Writing the current motor position to the log window

```
proc print_some_motor_positions {} {
variable table_vert
variable gonio_phi

log_note $table_vert
log_note $gonio_phi
}
```

Example Output:

```
15 Feb 2002 16:15:06 NOTE: 30.774772
15 Feb 2002 16:15:16 NOTE: 21.000000
```

6.1.3 Moving a motor

The intuitively named `move` command is used to move a motor. The motor may be a scripted device or a real motor.

For scripts running within the scripting engine, the *units* field may be omitted and the scaled units will be used by default. The scripting engine does not currently[4] know the units of the motor, and moves can only be made in the units described by the scale factor for the motor.

The general format of the command is:
`move` ***`motorName`*** `by|to` ***`value units`***
The *units* parameter can be

- *scaled* for movement using the scale factor for the motor.
- *unscaled* for movement using motor steps
- *mm,um,A,eV,V,deg* if the command is being issued from BLU-ICE.

Examples:
Issuing move commands from BLU-ICE:

- `move gonio_phi to 30 deg`

  Moves the motor `gonio_phi` to a position of 30 degrees.

- `move gonio_phi by 30 deg`

  Moves the motor `gonio_phi` by 30 degrees from its current positions.

- `move gonio_phi to 30 unscaled`

  Moves the motor `gonio_phi` by 30 steps.

- `move energy to 13000.00 eV`
- `move energy to 1.00 A`

Issuing move commands from the Scripting Engine:

- `move gonio_phi by 30 scaled`

  Moves the motor to 30 degrees, if the scale factor for `gonio_phi` converts steps to degrees.

- `move energy to 13000.00`

  The move will be in eV, if the script for `energy` reports its value in eV.

[4]https://smb.slac.stanford.edu/bugzilla/show_bug.cgi?id=23

6.1.4 Inserting/Removing shutters and filters

The DCS command for removing a filter is:

`open_shutter` *shutterName*

The DCS command for inserting a filter is:

`close_shutter` *shutterName*

Examples:

- `open_shutter shutter`
- `close_shutter Al_8`
- `open_shutter Se`

6.1.5 Waiting for hardware

An important aspect of writing scripts that interact with hardware is being able to guarantee that a hardware related function has completed before issuing the next command. The `wait_for_devices` command allows the script writer to halt the script at the appropriate time and wait for a hardware component to complete its current task. This command does not freeze the BLU-ICE GUI or the scripting engine while the script is waiting for the event that will trigger the `wait_for_devices` command to return.

The DCS command in its general form is:

```
wait_for_devices deviceName1 [deviceName2 [deviceName3 [...]]]
```

Each *deviceName* argument may be the name of a motor or ion chamber. The list of devices may be in any order.

An additional command is provided to wait for shutters/filters to be inserted and removed:

```
wait_for_shutters shutterName1 [shutterName2 [shutterName3 [...]]]
```

It should be understood that the `wait_for_shutters` command waits for the DHS responsible for the shutter to say that the shutter is open or closed. However, because there is often no shutter state feedback to a motion controller, the shutter may still be in the mechanical process of opening or closing.

At some point the functionality of `wait_for_shutters` should be added[5] to the `wait_for_devices` command, and the `wait_for_shutters` command should be phased out.

[5] https://smb.slac.stanford.edu/bugzilla/show_bug.cgi?id=25

Examples:

- **Waiting for a motor to stop moving**

```
move gonio_phi by 30 deg
wait_for_devices gonio_phi
```

- **Waiting for multiple motors to stop moving**

```
#start two motors moving
move energy to 11000.00 eV
move table_vert to 22.3 mm

#wait for both motors to stop moving
wait_for_devices energy table_vert

#Both motors have stopped moving now.
```

- **Waiting for shutters/filters**

```
open_shutter shutter
open_shutter Al_1
wait_for_shutters shutter Al_1
```

6.1.6 Using ion chambers

The command for reading an ion chamber or chambers is

`read_ion_chambers time detectorName1 [detectorName2 [...]]`

All of the detectors must be associated with the same timer at the hardware level. After the command is issued, the `wait_for_devices` command can be issued to wait for the detectors to count over the requested time. After the `wait_for_devices` is issued, each of the readings for the ion chambers can be acquired individually using the `get_ion_chamber_counts` command in the following format:

`get_ion_chamber_counts detectorName`

Examples:

- **Reading two ion chambers for 2.5 seconds**

```
read_ion_chambers 2.5 i2 i5
wait_for_devices i2

#print the results to the log window

log_note [get_ion_chamber_counts i2]
log_note [get_ion_chamber_counts i5]
```

6.1.7 Using encoders

The command for reading an encoder position is:

`get_encoder` *encoderName*

The command for setting an encoder position is

`set_encoder` *encoderName*

After issuing the `get_encoder` command, it it necessary to issued a command to wait for the results to come back from the DHS responsible for this encoder. The command that does this is:

`wait_for_encoder` *encoderName*

This function should return the value of the encoder when the message is completed, but currently does not. This bug has a work around as described in the bug repository[6].

6.1.8 Using Operations

An operation can be a script executing by the scripting engine, or it can be hardware function executed by a hardware server. For example, reading out a detector may be a specific hardware function that can be accessed via an operation. The data collection script within DCSS is an example of a scripted operation.

6.1.9 Starting an operation

There are two forms of the command used to start operations:

```
start_operation          operationName [arg1 [arg2 [arg3 [...]]]]
start_waitable_operation operationName [arg1 [arg2 [arg3 [...]]]]
```

[6] https://smb.slac.stanford.edu/bugzilla/long_list.cgi?buglist=30

Both of these commands start the operation, but the `start_waitable_operation` will return a ***operation handle***. This handle should be stored in a TCL variable and can be used later to obtain the results of the operation as discussed in section 6.1.10.

The number of needed arguments depends upon the operation that is being called.

Examples:

- `start_operation collectRuns 4`

  This starts an operation `collectRuns` with a single argument of 4. There is no way to for the caller to obtain results from the operation after initiating the operation in this way.

- `set opHandle [start_waitable_operation optimize table_vert i2 20 0.05 0.1]`

  This starts an operation called `optimize` with several arguments. The TCL variable `opHandle` will contain a unique value which can be used later to obtain the results of the operation.

6.1.10 Obtaining operation results

The `start_waitable_operation` command returns a unique *operation handle*, which can be passed to the following command to obtain the results of the operation:

```
wait_for_operation operationHandle
```

This function will return the results of the operation in the following formatted string:

```
status returnValue1 [returnValue2 [returnValue3 [...]]]
```

The `status` field will contain one of the following:

- `normal`

  This is returned if the operation has completed without any errors, and the return arguments are valid.

- `update`

  This indicates that an update from the operation has been received. There may be arguments that can be used as intermediate results (e.g.

for graphing individual points as the operation proceeds). The `update` message may also be used as a trigger to perform another operation.

A status of `update` indicates that the operation is not yet completed. It is necessary to issued the `wait_for_operation` command again until the status changes to `normal`.

- An error message

  Any string other than `normal` or `update` in the `status` field indicates that an error has occurred in the operation.

  The `wait_for_operation` command will return a TCL level error which must be caught (using the TCL `catch` command) if the script writer intends to do any clean up. If the error is not caught, the currently running operation or scripted device will return an error condition.

  See section 6.3 for details on handling errors.

It is not necessary to worry about the exact timing of the `wait_for_operation` command. If the `wait_for_operation` command is issued after the operation has already completed, the results of the operation will be returned by the scripting engine immediately. If the command is issued before the operation is completed, the script will hang until the operation is completed or the operation sends an `update` message.

Example: Waiting for the results of the optimize operation

```
set opHandle [start_waitable_operation optimize table_vert i2 20 0.05 0.1]
set result [wait_for_operation $opHandle]
log_note $result

output:  normal 30.774772
```

6.2 Scripted Device Family Relationships

A scripted device must describe its relationship to other devices directly within its script. This is achieved using three commands: Device relationships are explicitly written within the scripts for the devices by using the following commands:

- `set_children`
- `set_siblings`

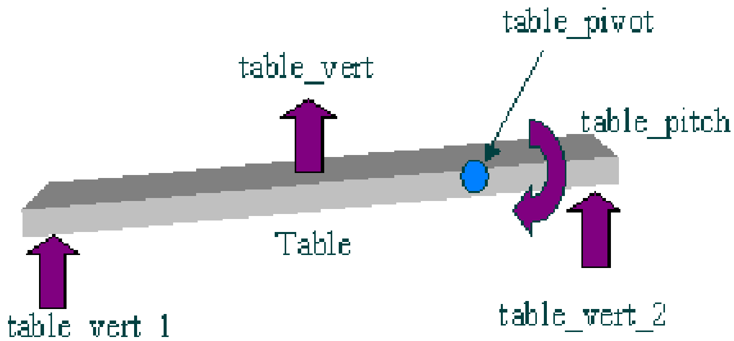

Figure 2: Children and Parent Motor Device

- `set_observers`

These commands should be issued from within the *deviceName*`_initialize` procedure as described in Section 7.1.

Figure 2 shows a typical example of several real motors being described by several pseudo motors. In this example, `table_vert` and `table_pitch` are parents of `table_vert_1` and `table_vert_2`. The parent motors `table_vert` and `table_pitch` are siblings (a.k.a joint custody parents).

6.2.1 Children and Parents

A scripted device is often used to control several real motors, or a hierarchy of scripted devices and real motors. If the relationship of the devices is such that a scripted device (`Device1`) should reevaluate its current position with each movement of (`Device2`), it is likely that `Device2` is a child of `Device1`, the parent device.

Within the script of the parent device, it is possible to declare this relationship with the following command:

`set_children` *childDevice1* `[[`*childDevice2*`] [...]]`

After this command is issued, the scripting engine will automatically call the *deviceName*`_update` procedure each time a child motor moves. This procedure must be written by the script writer and should re-calculate and set the parent motor's new position. With the setting of the new position the scripting engine will then call all of the *deviceName*`_update` procedures for all scripted devices that are parents of this parent motor. Therefore, a single updated position from a child motor will cause all of the parents in the hierarchy to re-calculate their own new positions.

6.2.2 Siblings

Very often several scripted devices can be written to describe different relationships between the same real motors. If the user expects to be able to move each scripted device without affecting the position of the other scripted device, then the `set_siblings` command can be used to declare this desired relationship between two parent devices.

For example, the average position of `table_vert_1` and `table_vert_2` can be written into a scripted device called `table_vert`. Moving this `table_vert` "motor" would move `table_vert_1` and `table_vert_2` by an equal amount. However, `table_vert_1` and `table_vert_2` can also be described in terms of an angle, `table_pitch`. With the `table_vert` and `table_pitch` motors now defined, a user may expect to be able to move `table_vert` without changing `table_pitch`. On the other hand, a user may also expect to be able to move `table_pitch` without changing `table_vert`.

A problem arises with this scenario when one considers that the `table_vert` motor issues two independent move commands to two real motors. These two motors may travel at slightly different speeds, thus altering the `table_pitch` motor during the move. At the end of the move, the `table_pitch` should once again be back to its original position before the move was started. This by itself is probably acceptable, as the limits of `table_pitch` are continuously checked and the move would be aborted if the limits were exceeded.

However, if a move of `table_vert` is aborted, either by a user or by a scripted device that has had its limits violated, the position of `table_pitch` will have been altered at the end of the move. Further moves of `table_vert` would maintain the new altered position of `table_pitch`. This is unacceptable because the `table_pitch` motor was altered simply because of an interrupted move of `table_vert`.

The `set_siblings` command can help the scripting engine overcome this problem by doing the following:

- With each move a motor that completes normally, the current position of all of the motor's siblings are stored as the last good position.
- With each move a motor that completes with an error (aborted or limit violation), no value will be stored.
- Every time a motor move is started, the status of all of the sibling motors are checked, and any of these motors that are not at the last good position will be told to move to their last good position before the initial request move is performed.

With `table_pitch` defined as a sibling of `table_vert`, aborting `table_vert` during a move will not update `table_pitch`'s last good position. When `table_vert` is moved again, the `table_pitch` motor will first be moved to its last good position before `table_vert` is allowed to move.

6.2.3 Observers

6.3 Exception Handling

As most developers of complex systems know, it is often a huge challenge to build in the ability to handle unexpected events. The designer should always be asking what the software should do when events like the following occur:

- A motor hits a hardware limit during a complex operation.
- The user hits the `abort` button during a complex operation.
- The computer running a DHS is powered off accidentally.
- The user issues a command with strange parameters that triggers a division by zero.
- A motion controller's power supply blows out.
- A vital network switch is re-booted.

The scripting engine within DCSS attempts to handle unexpected errors in a consistent way without requiring the script writer adding a single line of code. The writer of the script only needs to write the script as if everything is working normally to obtain the default error handling behavior. However, the default handling of exceptions may be insufficient in some situations, and the script writer can write additional code to handle these special cases.

All un-handled exceptions within a script will do one of two things, depending on the script type:

- For scripted operations, the script will terminate and will automatically append the TCL error to the operation's terminating DCS message.

  This will allow the caller to be able to determine the reasons for scripted operation's failure.

- For scripted devices, the script will terminate and will automatically return a `aborted` status to the caller.

  This is a system deficiency that will be overcome in later versions of DCS. The error message should be generic in the same was as the scripted operations.

Currently the scripting engine design is very picky about errors on motors and may seem heavy-handed. However, the purpose is to prevent possible hardware damage instead of allowing a script to grind onward by repeating a move again and again. The script writer should be careful about added exceptions handlers for the code that could override a default system behavior that was designed with safety in mind. Often it is best to allow the script to exit and force the user to resolve the problem.

6.3.1 The legalities of problems with motors

When issuing a `move` command from a script in the scripting engine, the following situations will cause an exception to be thrown:

- The global abort was been issued after this script was started.
- The motor is already moving.
- The move would exceed the motor's software limits.
- The motor has a child motor that is currently moving.
- The motor is locked.

The following errors will cause the scripting engine to issue a global abort.

- A sibling motor needs to be moved back to its last good position, but the `move` command threw an exception for one of the reasons listed above.
- The scripting engine receives a message indicating that a motor has completed its move abnormally,

After successfully issuing a `move` command, the script will continue until the `wait_for_devices` function is called. The `wait_for_devices` command will return and allow the script to continue only if the device completed its move normally. If the device has not completed normally, an exception will be thrown with the reason that it failed.

6.3.2 Handling the global abort

Currently, the `Abort` feature within DCS is a global function, and abort messages will be sent to all active operations and moving motors. It is not possible to issue an abort to a particular motor. Future developments may make the aborting of an individual motor or operation a requirement, but this is not the current design philosophy.

After the scripting engine receives an `abort` message it will not allow a currently running script to start a motor moving or start another operation. The `move`, `start_operation`, and `start_waitable_operation` commands will throw an exception if they are called after the system has been aborted. The script is allowed to continue executing up to the point where it tries to issue one of these commands.

6.3.3 Handling DHS Crashes

The scripting engine will automatically generate terminating DCS messages for operations that are active and motors that are moving if the responsible DHS loses its socket connection. The terminated DCS message for each moving motor and active operation contains an error code, and an exception will be thrown for all of the outstanding `wait_for_devices` and `wait_for_operations` calls associated with the crashed DHS.

6.3.4 Throwing your own exceptions

The scripting engine will catch all un-handled errors and append these errors to the script's terminating DCS message. Therefore, the script writer may also throw exceptions that will automatically be transmitted out on the operation's completed message. For example, the following code within a script would terminate the script with a completed message and an error of `negativeNumber`. BLU-ICE clients could interpret this message and handle it however they wished.

```
if { $value < 0 } {
return -code error negativeNumber
}
```

6.3.5 Writing specialized exception handlers

All exceptions generated within the scripting engine obey the rules of TCL. Therefore, the script writer should consult the TCL documentation to fully understand the syntax of exception handling.

If the script writer wishes to start an operation after a global abort has been issued, there are two commands for doing this:

- `start_recovery_operation operationName [arg1 [arg2 [arg3 [...]]]]`
- `start_waitable_recovery_operation operationName [arg1 [arg2 [arg3 [...]]]]`

These two functions behave the same as the `start_operation` and the `start_waitable_operation` commands, except that they bypass the check for the global abort flag.

These functions can be used to start operations that are needed in order to recover from a bad state.

For example, the collectRun operation interfaces with a Q4 CCD detector. The Q4 CCD detector should not flush its buffer after every image, because this would eliminate the double buffer capability of the detector. However, if an exception happens within the script somewhere, the detector must flush its buffer before the script is completed. Therefore the collectRun operation catches all exceptions, and starts the recovery operation `detector_stop` before throwing the original exception again.

7 Adding Scripts to the Scripting Engine

Adding new scripts to the DCSS scripting engine involves several steps as outlined in the following sub-sections.

7.1 Adding New Scripted Devices

This section describes the procedure for creating a new scripted device.

1. Create an entry in the `database.dat` file for the new device as described in Section 5.3.2.

    - Set the *`responsibleDHS`* to 'self'.
    - Set the *`externalName`* parameter to the name of your new device file without the `.tcl` extension.

      Alternatively you can set the *`externalName`* to `standardVirtualDevice`. This would make the device a motor that simply holds a position. If you select the `standardVirtualDevice`, then this is the only step that you need to perform, and you can restart DCSS.

2. Create the new device file in the `dcss` devices directory: `dcss/scripts/devices`

   The file name should be the name of the new device (as listed in the database.dat file) with a `.tcl` extension. For example, the `table_vert` scripted device should have a file `table_vert.tcl` within the devices directory.

3. Within the new scripted device file define 5 TCL procedures, where *deviceName* is replaced by the name of the new scripted device.

   - `proc` *deviceName*`_initialize {}`

     This procedure is executed when DCSS starts up. This procedure may be empty, or it may be used to initialize variables associated with the scripted device.

     This is also the correct place to call the `set_children`, `set_siblings`, and `set_observers` functions as discussed in Section 6.2.

   - `proc` *deviceName*`_set { newPosition}`

     This is the procedure that is called when a motor configuration is initiated by the `set` command. The value being applied to the scripted device motor is passed in the `newPosition`. This procedure does not necessarily have to accept the new position, but can use the value to set other motors.

   - `proc` *deviceName*`_move {newPosition}`

     This is the procedure that is executed when an attempt is made to move the scripted device. The script is free to do whatever it wants, including moving other motors or calling scripted operations. If the script moves other motors, it is likely that these motors should be listed as children using the `set_children` command. However, this is not a requirement to moving other motors.

   - *deviceName*`_update`

     This procedure is called whenever the scripting engine has reason to believe that the scripted device's current position is out of date. This happens under the following conditions:

     – The scripting engine receives an update an a child motor of the scripted device.

     – The scripting engine receives an move complete on a child motor of the scripted device.

  - The scripting engine receives a configuration on a child motor of the scripted device.
- `proc` *deviceName*`_calculate {[child1 [child2 [child3]]]}`

  This procedure is called by the scripting engine when determining if a child motor is allowed to move to a certain position without violating the parent's software limits. The arguments of this function are listed in the order that this function listed its children in the `set_children` command.

  This function must accept theoretical children positions and recalculate a new position. It is common for the *deviceName*`_update` command to use this procedure to update its current position using current children motor positions.

4. Restart DCSS.

7.2 Adding New Scripted Operations

This section describes the procedure for creating a new scripted operation.

1. Create an entry in the `database.dat` file for the operation as described in Section 5.3.8.

   Set the *responsibleDHS* to 'self'.

   Set the *externalName* parameter to the name of your new operation file without the `.tcl` extension.

2. Create the new operation file in the `dcss` operation directory, `dcss/scripts/operations`

3. Within the new operation file define two procedures, where the *operationName* is replaced by the name of the new operation.

   - `proc` *operationName*`_initialize {}`

     This procedure is executed when DCSS starts up. This procedure may be empty, or it may be used to initialize variables associated with the scripted operation.

   - `proc` *operationName*`_start { [arg1 [arg2 [arg3 [args]]]]}`

     The *operationName*`_start` procedure is executed when a `start_operation` message is received by the scripting engine. This procedure should contain the actual functionality if the operation.

     The arguments passed to this procedure are the same arguments passed to the initiating command:

```
start_operation operationName [arg1 [arg2 [arg3 [...]]]].
```

4. Restart DCSS.

8 Example scripts

The following subsections should help give the reader ideas for writing their own scripted devices and operations.

8.1 Testing the diffractometer

It is possible to imagine a damaged diffractometer in which it is possible for the motorized sample motors to lose their encoder feedback lines in certain orientations of phi. The script in Figure 3 shows a diagnostics test that can be run from BLU-ICE to look for this type of situation. To run this script from BLU-ICE, open the command prompt and type

`source` *`filename`*

where the *`filename`* is the name of the file that contains the script.

This script will look for orientations of phi that disconnect the encoder lines. When the encoder lines are broken the sample motors would move until they hit a hardware limit, and the script will halt at the bad phi position.

If this script were found to be extremely useful and used often, the DCS administrator could convert this script into a scripted operation as shown in Figure 4. The operation would need to be added to the database.dat file as described in Section 7.2. The user of BLU-ICE would then be able to type in the following from the BLU-ICE command prompt to execute the operation:

`start_operation diffractometerTest`

8.2 The `energy` device on different beam lines

The differences between beam lines at SSRL can can be easily addressed by writing specific scripts for each unique hardware architecture.

For example, beam line 9-2 and beam line 1-5 at SSRL each have their own energy script: `energy_bl92.tcl` and `energy_bl15.tcl` respectively. The `database.dat` file for each beamline has an `energy` device entry which points at the energy script for that beam line.

The entry for the `energy` scripted device may look like this in the `database.dat` file on beam line 9-2:

```
#store the current positions of the sample motors
set startx $sample_x
set starty $sample_y
set startz $sample_z

#initialize our phi variable
set phi 0
while {$phi < 360} {

    log $phi

    #test the next position of phi
    move gonio_phi to $phi deg
    wait_for_devices gonio_phi

    #move the sample motors a little bit
    move sample_x by 0.01 mm
    move sample_y by 0.01 mm
    move sample_z by 0.01 mm

    #wait for the motors to finish the move
    wait_for_devices sample_x sample_y sample_z

    #move the sample motors back to start
    move sample_x to $startx mm
    move sample_y to $starty mm
    move sample_z to $startz mm

    #wait for the motors to finish moving back
    wait_for_devices sample_x sample_y sample_z

    #move our phi variable
    set phi [expr $phi + 0.1]
}
```

Figure 3: Script for testing Diffractometer: Text Listing.

```
proc diffractometerTest_initialize {} {
}

proc diffractometerTest_start { } {
   #Bring motor positions into this namespace
   variable sample_x
   variable sample_y
   variable sample_z

   #store the current positions of the sample motors
   set startx $sample_x
   set starty $sample_y
   set startz $sample_z

   #initialize our phi variable
   set phi 0
   while {$phi < 360} {

      log $phi

      #test the next position of phi
      move gonio_phi to $phi
      wait_for_devices gonio_phi

      #move the sample motors a little bit
      move sample_x by 0.01
      move sample_y by 0.01
      move sample_z by 0.01

      #wait for the motors to finish the move
      wait_for_devices sample_x sample_y sample_z

      #move the sample motors back to start
      move sample_x to $startx
      move sample_y to $starty
      move sample_z to $startz

      #wait for the motors to finish moving back
      wait_for_devices sample_x sample_y sample_z

      #move our phi variable
      set phi [expr $phi + 0.1]
   }
```

Figure 4: Script for testing Diffractometer: Text Listing.

```
energy
2
self energy_bl92
14500.036128 15000.000000 5900.000000 0 0 0 0
0
0
```

whereas on beam line 1-5 the entry for `energy` may look like this:

```
energy
2
self energy_bl15
12999.846769 16000.000000 5900.000000 0 0 0 0
0
0
```

Beam line 9-2 moves the monochromator in order to change energy for the beam line. The monochromator is equipped with a encoder which allows the software to verify that the position is changing correctly. A `mono_theta_corr` scripted device (not shown here) was written which is responsible for moving the real `mono_theta` motor and verifies (using the encoder) that the motor achieved the desired position. The script for `energy` for beam line 9-2 shown in Figure 5 simply moves the `mono_theta_corr` device.

Beam line 1-5, on the other hand, requires moving a monochromator (without an encoder this time) as well as changing the voltage on a piezo. A scripted device was written to assist in controlling the voltage of the piezo according to a curve that was experimentally determined. In addition to this, the beam moves with the change in energy, and the table is adjusted slightly to accommodate this change. The resulting `energy` script is significantly different for beam line 1-5 as shown in Figure 6.

These scripts completely encapsulate the process of changing energy, hiding it as if the software were simply moving a motor. This greatly simplifies higher level software and allows the Blu-Ice Gui to provide a simple drop down menu for energy selection.

9 The DCS Protocol

Three components comprise the DCS architecture: hardware servers (DHS), GUI clients, and the DCS server, DCSS. GUI clients and hardware servers

```
# energy.tcl

proc energy_initialize {} {

   # specify children devices
   set_children mono_theta_corr d_spacing
}

proc energy_move { new_energy } {

   # global variables
   variable d_spacing

   # calculate destination for mono_theta_corr
   set new_mono_theta_corr \
       [energy_calculate_mono_theta_corr $new_energy $d_spacing]

   # move mono_theta to destination
   move mono_theta_corr to $new_mono_theta_corr

   # wait for the move to complete
   wait_for_devices mono_theta_corr
}


proc energy_calculate { mtc ds } {

   # calculate energy from d_spacing and mono_theta_corr
   return [expr 12398.4244 / (2.0 * $ds * sin([rad $mtc]) ) ]
}


proc energy_calculate_mono_theta_corr { e ds } {

   # calculate mono_theta from energy and d_spacing
   return [deg [expr asin(  12398.4244 / ( 2.0 * $ds * $e ) ) ]]
}
```

Figure 5: Energy on Beam Line 9-2

```
# energy.tcl

proc energy_initialize {} {

   # specify children devices
   set_children mono_theta d_spacing mono_piezo table_pitch \
                table_yaw table_vert table_horz
}

proc energy_move { new_energy } {

   # global variables
   variable energy
   variable d_spacing
   variable mono_piezo_offset
   variable table_horz_1_offset
   variable table_horz_2_offset
   variable table_vert_1_offset
   variable table_vert_2_offset

   # calculate destination for mono_theta
   set new_mono_theta [energy_calculate_mono_theta $new_energy $d_spacing]

   # move mono_theta
   move mono_theta to $new_mono_theta

   # move mono_piezo
   move mono_piezo to [expr \
      [energy_calculate_mono_piezo $new_mono_theta] + $mono_piezo_offset]

   # move table_horz_1
   move table_horz_1 to [expr \
      [energy_calculate_table_horz_1 $new_mono_theta] + $table_horz_1_offset]

   # wait for the moves to complete
   wait_for_devices mono_theta mono_piezo table_horz_1
}

proc energy_calculate { mt ds pv tp ty tv th } {

   # calculate energy from d_spacing and mono_theta
   return [expr 12398.4244 / (2.0 * $ds * sin([rad $mt]) ) ]
}

proc energy_calculate_mono_theta { e ds } {

   # calculate mono_theta from energy and d_spacing
   return [deg [expr asin(  12398.4244 / ( 2.0 * $ds * $e ) ) ]]
}

proc energy_calculate_mono_piezo { mt } {

   expr 0.368 + 11.44397 * $mt - 0.208731 * $mt*$mt
}

proc energy_calculate_table_horz_1 { mt } {

   expr -11.3647 + 0.0695967 * $mt - 0.0007411889 * $mt*$mt
}
```


Figure 6: Energy on Beam Line 1-5

communicate solely with DCSS, and DCSS operates the only listening sockets (i.e., network server) in the DCS system. Consequently, hardware servers are in a sense network clients of DCSS and are sometimes referred to as clients of DCSS in the DCS message protocol documentation.

DCS hardware servers encapsulate low-level control of physical devices such as motors, shutters, detectors, and so on. Hardware servers simply connect to DCSS and do whatever DCSS tells them to do. DCS GUI clients such as BLU-ICE are the ultimate source of these instructions to the hardware servers. DCSS passes the requests from GUI clients down to the appropriate hardware servers, and broadcasts all replies from hardware servers back to all of the GUI clients, thus keeping all GUI clients in complete synchronization.

By convention, the first four letters of the command in a DCS message specify the source and destination of the message in the DCS architecture. The five possibilities are the following:

- `stog`: server to gui client
- `stoh`: server to hardware client
- `gtos`: gui to server
- `htos`: hardware client to server
- `stoc`: server to client (hardware or GUI)

9.1 The DCS Message Structure

All communication between the components of the DCS architecture uses a message passing protocol which runs on top of TCP/IP. The protocol is completely asynchronous (i.e., not strict client/server).

The network message is split into three distinct sections: the header, the text section, and the binary section.

9.1.1 The DCS message header

Each DCS message must start with 26 bytes of text message. These 26 bytes must contain a 2 numbers is ASCII format. The two numbers indicate the size of the text section (in bytes) and binary section (in bytes). The text message must contain a terminating 0 at the end of the two numbers. This allows a library to simply use the `scanf` function to obtain the two numbers.

An example header may look like this:

											1	2	3	5							2	3	2	5	0x00

The first number in this example indicates that there are 1235 bytes of text following the header. The second number in this string indicates that there are 2325 bytes of binary data following the 1235 bytes of text.

The 0x00 terminates the header.

DCSS does not currently support the binary portion of the DCS message completely. In other words, the binary message should always be "0" for messages passed from DHS to DCSS.

The binary portion of the message is currently only used for sending the authentication key to BLU-ICE during the authentication stage made during the initial connection of BLU-ICE to DCSS.

9.1.2 The text section

The text portion of each DCS message consists of a null terminated ASCII text string composed of white-space separated tokens. The first token in each message is the DCS command; all remaining tokens are the arguments to the command. For example, the following is a command from a GUI client to DCSS requesting that the motor `table_vert_1` be moved to the absolute scaled position of 10.0:

```
gtos_start_motor_move table_vert_1 10.0
```

9.1.3 The binary section

This is currently used only during the authentication stage between DCSS and BLU-ICE.

9.2 Connection Protocol

The connection protocol has been simplified from the first two releases of DCS. The GUI and hardware clients connection to DCSS on different ports.

For hardware servers (DHS):

1. The new DHS opens a socket connection to the hardware listening port on DCSS.

2. DHS begins reading and handling messages sent from DCSS.

3. DCSS will send a `stoc_send_client_type` to the DHS.

4. The DHS has 1 second to respond with the following message:

   `htos_client_is_hardware` *`dhsName`* as discussed in Section 9.4.1.

   At this point DCSS will disconnect a DHS for the following reasons:

   - The DHS does not respond within 1 second.
   - The DHS does not respond with the name of a DHS listed within the `database.dat` file.
   - There is already a DHS connected with the same name as the DHS that is currently trying to connect.

For GUI clients (BLU-ICE):

1. The new BLU-ICE opens a socket connection to the GUI client listening port on DCSS.

2. DCSS will send a `stoc_send_client_type` to the BLU-ICE.

3. The BLU-ICE has 1 second to respond with the following message:

   `htos_client_is_gui` *`userName hostname display`*

   Where *`userName`* is the unix account name of a the user that initiated the BLU-ICE.

   At this point DCSS has the option to disconnect the BLU-ICE for the following reasons:

   - The BLU-ICE does not send the response within 1 second.
   - The BLU-ICE responds with a *`userName`* that is not listed within the mysql database `Users` table.

4. After DCSS confirms that the userName is listed in the mysql database, DCSS will send the following message to BLU-ICE with 200 bytes of trailing binary data:

   `stog_respond_to_challenge`

5. BLU-ICE must read the 200 bytes of binary data and respond to DCSS appropriately.

   At this point DCSS will disconnect the BLU-ICE for the following reasons:

   - The BLU-ICE client does not respond within 1 second.
   - The BLU-ICE does not send the appropriate response.

6. DCSS sends an message `stog_login_complete`

7. DCSS sends another message

   `stog_set_permission_level` *`permissionLevel`*

   where *`permissionsLevel`* is a number obtained from the mysql's Users table that can be used by the GUI to disable or enable certain buttons and tabs. DCSS will also use this value to prevent the user from doing what they are not allowed to do.

8. DCSS will then proceed to update the BLU-ICE client with the complete state of the beam line. This will include every motor position.

9. DCSS will then begin processing message from BLU-ICE as they come in, and will send messages to the BLU-ICE client to keep the BLU-ICE up-to-date with the latest status of motor positions or operation activity.

10. DCSS will disconnect the BLU-ICE under the following circumstances:

    - The BLU-ICE is too slow in processing the message in its TCP/IP input buffer and DCSS's TCP/IP output buffer becomes full.
    - The BLU-ICE socket has an error.

9.3 Gui to Server Messages (gtos)

All commands sent from a GUI client to DCSS start with a `gtos` (pronounced g-2-s).

9.3.1 gtos_abort_all

The format of the message is

`gtos_abort_all` *[hard | soft]*

This command requests that all operations either cease immediately or halt as soon as possible. All motors are stopped and data collection begins shutting down and stopping detector activity. A single argument specifies how motors should be aborted. A value of hard indicates that motors should stop without decelerating. A value of soft indicates that motors should decelerate properly before stopping.

9.3.2 gtos_become_master

The format of the message is

`gtos_become_master` *[force | noforce]*

This command requests that the sender be made the `Active` GUI client. A mode of `force` indicates that the client wants to become the `Active` client even if another client is currently the `active` client. A mode of `noforce` indicates that it only wants to be `Active` if there currently is no other `Active` client.

DCSS responds with `stog_become_master` if the GUI succeeds in becoming the `Active` client. If `noforce` was specified by the GUI client and another GUI is currently `Active`, DCSS will reply with `stog_other_master`.

9.3.3 gtos_become_slave

The format of the message is

`gtos_become_slave`

This command requests that the sender be made a non-Active GUI client. DCSS responds with `stog_become_slave`.

9.3.4 gtos_configure_device

The format of the message can take one of two forms:

1. `gtos_configure_device` *motorName position upperLimit lowerLimit lowerLimitOn upperLimitOn motorLockOn units*

    - *motorName* is the name of the motor to configure.
    - *position* is the scaled position of the motor.
    - *upperLimit* is the upper limit for the motor in scaled units.

- *lowerLimit* is the lower limit for the motor in scaled units.
- *lowerLimitOn* is a boolean (0 or 1) indicating if the lower limit is enabled.
- *upperLimitOn* is a boolean (0 or 1) indicating if the upper limit is enabled.
- *motorLockOn* is a boolean (0 or 1) indicating if the motor is software locked.

2. gtos_configure_device *motoName position upperLimit lowerLimit scaleFactor speed acceleration backlash lowerLimitOn upperLimitOn motorLockOn backlashOn reverseOn*

   where

   - *motor* is the name of the motor to configure
   - *position* is the scaled position of the motor
   - *upperLimit* is the upper limit for the motor in scaled units
   - *lowerLimit* is the lower limit for the motor in scaled units
   - *scaleFactor* is the scale factor relating scaled units to steps for the motor
   - *speed* is the slew rate for the motor in steps/sec
   - *acceleration* is the acceleration time for the motor in seconds
   - *backlash* is the backlash amount for the motor in steps
   - *lowerLimitOn* is a boolean (0 or 1) indicating if the lower limit is enabled
   - *upperLimitOn* is a boolean (0 or 1) indicating if the upper limit is enabled
   - *motorLockOn* is a boolean (0 or 1) indicating if the motor is software locked
   - *backlashOn* is a boolean (0 or 1) indicating if backlash correction is enabled
   - *reverseOn* is a boolean (0 or 1) indicating if the motor direction is reversed

This command requests that the configuration of a real motor be changed. DCSS updates the device configuration in its internal database (database.dat) and forwards the message to the appropriate hardware server.

Note: This message should probably be two separate messages `gtos_configure_real_motor` and `gtos_configure_pseudo_motor`.

9.3.5 gtos_read_ion_chambers

The format of the message is

`gtos_read_ion_chambers` *time repeat ch1 [ch2 [ch3 [...]]]*

where

- *time* is the time in seconds over which to integrate counts.
- *repeat* is a boolean (0 or 1) indicating if chamber should be read iteratively.
- *ch1..ch[n]* is the list of names of the ion chambers to read.

This command requests that one or more ion chambers be read, using the same real time clock. All of the specified ion chambers must be controlled by the same hardware server (DHS).

9.3.6 gtos_set_motor_position

The format of the message is

`gtos_set_motor_position` *motorName position*

where

- *motorName* is the name of the motor to configure.
- *position* is the new scaled position of the motor.

This message requests that the position of the specified motor be set to specified scaled value. It is similar to the `gtos_configure_device` but changes the position only. DCSS forwards this message to the appropriate hardware server(DHS).

9.3.7 gtos_set_shutter_state

The format of the message is

`gtos_set_shutter_state` *shutterName state*

where

- *shutterName* is the name of the shutter/filter to open or close.
- *state* is `open` or `closed`.

This message requests that the state of the specified shutter or filter be changed to a particular state.

9.3.8 `gtos_start_motor_move`

The format of the message is

`gtos_start_motor_move` *`motorName destination`*
where

- *`motorName`* is the name of the motor to move
- *`destination`* is the scaled destination of the motor.

The message requests that the specified motor be moved to the specified scaled position.

9.3.9 `gtos_start_oscillation`

The format of the message is

`gtos_start_oscillation` *`motorName shutter deltaMotor deltaTime`*
where

- *`motorName`* is the name of the motor to oscillate during the exposure.
- *`shutter`* is the name of the shutter used to expose the sample.
- *`deltaMotor`* is the range of motion that the exposure should occur.
- *`deltaTime`* is the time over which the exposure should occur.

The message requests that a precision exposure be performed using the specified motor and shutter, and over the motor range and time interval specified. The speed of the motor over the deltaMotor range should be uniform.

DCSS forwards this message to the appropriate hardware server as a `stoh_start_oscillation` message.

Note: This message is primarily used for testing oscillation code in hardware servers from BLU-ICE. It is not currently used to perform data collection sequencing from the GUI client level.

Note: This message will be converted to an operation message and will be obsolete in later versions of DCS.

9.3.10 gtos_start_operation

The format of the message is

`gtos_start_operation` *`operationName operationHandle [arg1 [arg2 [arg3 [...]]]]`*

where

- *`operationName`* is the name of the operation to be started.
- *`operationHandle`* is a unique handle currently constructed by calling the `create_operation_handle` procedure in BLU-ICE. This currently creates a handle in the following format:

  `clientNumber.operationCounter`

  where `clientNumber` is the number provided to the BLU-ICE by DCSS via the `stog_login_complete` message. DCSS will reject an operation message if the `clientNumber` does not match the client. The `operationCounter` is a number that the client should increment with each new operation that is started.
- *`arg1 [arg2 [arg3 [...]]]`* is the list of arguments that should be passed to the operation. It is recommended that the list of arguments continue to follow the general format of the DCS message structure (space separated tokens). However, this requirement can only be enforced by the writer of the operation handlers.

The message requests DCSS to forward the request to the appropriate hardware server.

9.4 Hardware to Server Messages (htos)

All commands sent from a hardware server (DHS) to DCSS start with a `htos` (pronounced h-2-s).

9.4.1 htos_client_is_hardware

This should be sent by all hardware servers in response to `stoc_send_client_type` message from DCSS.

The format of the message is

`htos_client_is_hardware` *`dhsName`*

where

Where *`dhsName`* is the name of a hardware server listed within the `database.dat` file as described in Section 5.3.3.

Note: This message is not forwarded to the GUI clients.

9.4.2 `htos_configure_device`

This is a message from a hardware server to DCSS updating the configuration of a device. DCSS updates its internal database and forwards the message to all GUI clients. Arguments are the same as for `gtos_configure_device` as discussed in Section 9.3.4.

9.4.3 `htos_motor_move_completed`

Indicates that the move on the specified motor is now complete. DCSS forwards this message to all GUI clients as a `stog_motor_move_completed` message.

The format of the message is
`htos_motor_move_completed` *`motorName position completionStatus`*

- *`motorName`* is the name of the motor that finished the move.
- *`position`* is the final position of the motor.
- *`completionStatus`* is the status of the motor with values as shown below:
  - `normal` indicates that the motor finished its commanded move successfully.
  - `aborted` indicates that the motor move was aborted.
  - `moving` indicates that the motor was already moving.
  - `cw_hw_limit` indicates that the motor hit the clockwise hardware limit.
  - `ccw_hw_limit` indicates that the motor hit the counter-clockwise hardware limit.
  - `both_hw_limits` indicates that the motor cable may be disconnected.
  - `unknown` indicates that the motor completed abnormally, but the DHS software or the hardware controller does not know why.

9.4.4 `htos_motor_move_started`

This message indicates that the requested move of a motor has begun. This message is forwarded by DCSS to all GUI clients as a `stog_motor_move_started` message.

The format of the message is

`htos_motor_move_started` *`motorName position`*

where

- *`motorName`* is the name of the motor.
- *`position`* is the destination move the motor.

9.4.5 `htos_update_motor_position`

This message updates the position of a motor during a motor move. DCSS forwards this message to all GUI clients as a `gtos_update_motor_position` message.

The format of the message is

`htos_update_motor_position` *`motorName position status`*

where

- `motorName` is the name of the motor
- `position` is the new position of the motor
- `status` is the status of the motors See Section 9.4.3.

9.4.6 `htos_report_shutter_state`

This message reports a change in the state of a shutter. This may occur as a result of handling the `stoh_set_shutter_state command` or during a timed exposure with automated shutter handling. DCSS forwards this message to all GUI clients as a `stog_report_shutter_state` message.

The format of the message is

`htos_report_shutter_state` *`shutterName state`*

where

- *`shutterName`* is the name of the shutter.
- *`state`* is the new state (`open | closed`.)

9.4.7 `htos_report_ion_chambers`

This message reports the results of counting on one or more ion chambers in response to the `stog_read_ion_chamber` message. The first three arguments are mandatory. Additional ion chambers are reported by adding additional arguments. DCSS forwards this message to all GUI clients as `stog_report_ion_chambers` message.

The format of the message is

`htos_report_ion_chambers` *`time ch1 counts1 [ch2 counts2 [ch3 counts3 [chN countsN]]]`*

where

- *`time`* is the time in seconds over which counts were integrated.
- *`ch1`* is the name of the first ion chamber read.
- *`cnts1`* is the counts from the first ion chamber.

9.4.8 `htos_send_configuration`

This message requests that the configuration of the specified device (as remembered by DCSS) be returned to this DHS. DCSS will respond with a `stoh_configure_device` message for the device. This message is not forwarded to the GUI clients.

The format of the message is

`htos_send_configuration` *`deviceName`*

where *`deviceName`* is the name of the device for which the configuration information is needed.

9.4.9 `htos_simulating_device`

This message indicates that the specified device is being simulated by the hardware server. DCSS forwards this message to all GUI clients as the `gtos_simulating_device` message.

The format of the message is

`htos_simulating_device` *`deviceName`*

where *`deviceName`* is the name of the device which is being simulated.

Note: If this message is not sent, the BLU-ICE clients will not know what motors are being simulated and what motors are real.

9.4.10 `htos_operation_update`

This message can be used to send small pieces of data to the GUI clients as progress is made on the operation. It can also be used to indicate to a calling GUI client that the operation cannot continue until the caller performs another task.

The message format is as follows:

`htos_operation_update` *`operationName operationHandle arguments`*

- *`operationName`* is the name of the operation that completed.
- *`operationHandle`* is the unique value that indicates which instance of the operation completed.
- *`arguments`* This is a list of return values. It is recommended that list of return arguments adhere to the overall DCS protocol (space separated tokens), but this can only be enforced by the writer of the operation handle.

9.4.11 `htos_operation_completed`

The message is used to indicate that an operation has been completed by this hardware server.

The general format of the message is

`htos_operation_completed` *`operationName operationHandle status arguments`*

where

- *`operationName`* is the name of the operation that completed.
- *`operationHandle`* is the unique value that indicates which instance of the operation completed.
- *`status`* Anything other than a `normal` in this field will indicate to DCSS and BLU-ICE that the operation failed, and this token will become the reason of failure.
- *`arguments`* This is a list of return values. It is recommended that list of return arguments adhere to the overall DCS protocol (space separated tokens), but this can only be enforced by the writer of the operation handle.

9.5 Server To GUI Client Messages (stog)

All commands sent from DCSS to a GUI client start with a `stog` (pronounced s-2-g).

9.5.1 `stog_become_master`

A command telling the GUI client that is now the `ACTIVE` GUI client. No arguments.

9.5.2 `stog_become_slave`

A command telling the GUI client that is not the `ACTIVE` GUI client. No arguments.

9.5.3 `stog_configure_real_motor`

A command to reconfigure a real motor. Arguments are the same as `stoh_configure_real_motor` see Section 9.6.4.

9.5.4 `stog_configure_pseudo_motor`

A command to reconfigure a pseudo motor. Arguments are the same as `stoh_configure_pseudo_motor`, see Section 9.6.5.

9.5.5 `stog_motor_move_completed`

Indicates that the move on the specified motor is now complete. Arguments are the same as for `htos_motor_move_completed`, see Section 9.4.3.

9.5.6 `stog_motor_move_started`

Indicates that the requested move of a motor has begun. Arguments are the same as for `htos_motor_move_started`, see Section 9.4.4.

9.5.7 `stog_no_hardware_host`

Indicates the hardware server for the specified device is not connected to DCSS. This message is sent by DCSS when a command from a GUI client refers to a device that cannot be accessed because the associated hardware server is not connected.

The format of the message is

`stog_no_hardware_host` *`deviceName`*

where *deviceName* is the name of the device for which no hardware server is connected

9.5.8 stog_other_master

Indicates that another client is currently the ACTIVE client. This message is sent to a particular GUI client if it sends a gtos_become_master noforce message while another GUI client is master. No arguments.

9.5.9 stog_report_ion_chambers

Reports the results of counting on one or more ion chambers in response to the stog_read_ion_chamber message. Arguments are the same as for htos_report_ion_chambers, see Section 9.4.7.

9.5.10 stog_report_shutter_state

Reports a change in the state of a shutter. Arguments are the same as for htos_report_shutter_state, see Section 9.4.6.

9.5.11 stog_simulating_device

Indicates that the specified device is being simulated by the a hardware server. Arguments are the same as for htos_simulating_device, see Section 9.4.9.

9.5.12 stog_unrecognized_command

Sent to a specific GUI client if it sends a DCS message that DCSS does not recognize. No arguments. Useful for debugging.

Note: Sometimes this command is sent when the user is requesting a command that requires the user to be the ACTIVE client.

9.5.13 stog_update_motor_position

Updates the position of a motor during a motor move. Arguments are the same as for htos_update_motor_position, see Section 9.4.5.

9.5.14 stog_operation_completed

Arguments are same as stog_operation_completed, see Section 9.4.11.

9.5.15 `stog_operation_update`

Arguments are same as `stog_operation_update`, see Section 9.4.10.

9.6 Server To Hardware Messages (stoh)

All commands sent from DCSS to a hardware server (DHS) start with a `stoh` (pronounced s-2-h).

9.6.1 `stoh_abort_all`

A command to stop all motors and other operations. Arguments are the same as for `gtos_abort_all`, see Section 9.3.1.

9.6.2 `stoh_correct_motor_position`

Requests that the position of the specified motor be adjusted by the specified correction. This is used to support the circle parameter for motors (i.e., modulo 360 behavior for a phi axis).

The format of the message is
`stoh_correct_motor_position` *`motorName correction`*
where

- *`motorName`* is the name of the motor.
- *`correction`* is the correction to be applied to the motor position

9.6.3 `stoh_start_motor_move`

This is a command to start a motor move. Arguments are the same as for `gtos_start_motor_move`, see Section 9.3.8.

9.6.4 `stoh_configure_real_motor`

The format of the message is
`stoh_configure_real_motor` *`motoName position upperLimit lowerLimit scaleFactor speed acceleration backlash lowerLimitOn upperLimitOn motorLockOn backlashOn reverseOn`*
where

- *`motor`* is the name of the motor to configure
- *`position`* is the scaled position of the motor

- *upperLimit* is the upper limit for the motor in scaled units
- *lowerLimit* is the lower limit for the motor in scaled units
- *scaleFactor* is the scale factor relating scaled units to steps for the motor
- *speed* is the slew rate for the motor in steps/sec
- *acceleration* is the acceleration time for the motor in seconds
- *backlash* is the backlash amount for the motor in steps
- *lowerLimitOn* is a boolean (0 or 1) indicating if the lower limit is enabled
- *upperLimitOn* is a boolean (0 or 1) indicating if the upper limit is enabled
- *motorLockOn* is a boolean (0 or 1) indicating if the motor is software locked
- *backlashOn* is a boolean (0 or 1) indicating if backlash correction is enabled
- *reverseOn* is a boolean (0 or 1) indicating if the motor direction is reversed

This command requests that the hardware server change the configuration of a real motor.

9.6.5 `stoh_configure_pseudo_motor`

The format of the message is
`stoh_configure_pseudo_motor` *motorName position upperLimit lowerLimit upperLimitOn motorLockOn*
where

- *motorName* is the name of the motor to configure.
- *position* is the scaled position of the motor.
- *upperLimit* is the upper limit for the motor in scaled units.
- *lowerLimit* is the lower limit for the motor in scaled units.

- *lowerLimitOn* is a boolean (0 or 1) indicating if the lower limit is enabled.

- *upperLimitOn* is a boolean (0 or 1) indicating if the upper limit is enabled.

- *motorLockOn* is a boolean (0 or 1) indicating if the motor is software locked.

A command to reconfigure a pseudo motor at the DHS level.

9.6.6 `stoh_read_ion_chambers`

Requests that one or more ion chambers be read. The first three arguments are mandatory. Additional arguments specify additional ion chambers to read simultaneously. Arguments are the same as for `gtos_read_ion_chambers`, see Section 9.3.5.

9.6.7 `stoh_set_motor_position`

Requests that the position of the specified motor be set to specified scaled value. Essentially an optimized `gtos_configure_device` for setting position only. Arguments are the same as for `gtos_set_motor_position`, see Section 9.3.6.

9.6.8 `stoh_set_shutter_state`

Requests that the state of the specified shutter be set to the specified state. Arguments are the same as for `gtos_set_shutter_state`, see Section 9.3.7.

9.6.9 `stoh_start_oscillation`

Requests that an oscillation be performed using the specified motor and shutter, and over the motor range and time interval specified. Arguments the same as `gtos_start_oscillation`, see Section 9.3.9.

9.6.10 `stoh_start_operation`

Requests the hardware server to do an operation. Arguments the same as `gtos_start_operation`, see Section 9.3.10.

10 Example code listings in ASCII

10.0.1 Test Diffractometer BLU-ICE Script

10.0.2 Test Diffractometer Operation Script

11 Adding new hardware support

Currently, the Blu-Ice/DCS project is rather open-ended regarding the specifics of adding new hardware support. Basically, adding hardware support involves writing a program that translates the Server to Hardware messages described in Section 9.6 into commands issued through the hardware controller's API. Alternatively, the Server To Hardware messages could be translated into another control system's command language, enabling a gateway for DCS to talk to another control system.

The simulated DHS project, (i.e. simdhs) is a great place to start when deciding how to implement a new DHS. It handles many of the possible hardware commands in several pages of code. Its main limitation is that it does not have handles for detector data collection related operations.

After studying the protocol, a developer may wish to look at the 'standard' DHS used at SSRL (found in the dhs project) which is used for controlling the Galil DMC 2180, ADSC Quantum 4, ADSC Quantum 315, and the MAR345. On the other hand, this code may be too heavy for a starting point, depending on your application. Specifically, the most difficult aspect of this code is that it uses the mysql database to acquire its configuration parameters. Developers are often lead to believe that a requirement for a program to be a DHS is that it somehow uses a mysql database. This is simple not true, and the 'simdhs' project is a counter example.

11.0.1 Example: Adding MAR CCD support

The following example lists the basic steps that were used to add support for a MAR CCD using the 'standard' DHS.

1. The MAR CCD protocol was studied and it was determined to be most similar to the existing Quantum 315 protocol. Therefore, it was decided to use the 'standard' DHS as a baseline.

2. The 'standard' DHS uses the mysql database to acquire its configuration. Thus, the following table was created to store a MAR CCD's configuration. At the mysql prompt:

```
CREATE TABLE MARCCD (
  ControllerID int(11) NOT NULL,
  serialNumber varchar(30) NOT NULL default '0',
  HostName varchar(30) default NULL,
  CommandPort int(11) default NULL,
  DarkRefreshTime int(11) default NULL,
  BeamCenterX double(16,4) default NULL,
  BeamCenterY double(16,4) default NULL,
  darkExposureTolerance float(10,2) NOT NULL default '0.10',
  writeRawImages char(1) default NULL,
  PRIMARY KEY  (ControllerID)
) TYPE=MyISAM;
```

3. After the table was created, a sample configuration was inserted into the table:

```
INSERT INTO MARCCD VALUES (1,'0','marpc',3000,180,94.0000,94.0000,0.10,'Y');
```

4. The `dhs_config.cc` file was modified to look at the MAR CCD table when determining its configuration.

```
diff -c -r1.8 dhs_config.cc
*** dhs_config.cc 8 Nov 2002 19:52:38 -0000 1.8
--- dhs_config.cc 13 Dec 2002 18:01:53 -0000
***************
*** 70,75 ****
--- 70,76 ----
  XOS_THREAD_ROUTINE Quantum315Thread (void * parameter );
  XOS_THREAD_ROUTINE MAR345 (void * parameter );
  XOS_THREAD_ROUTINE ASYNC2100 (void * parameter );
+ XOS_THREAD_ROUTINE MARCCDThread(void * paramater);
  #ifdef WITH_CAMERA_SUPPORT
  XOS_THREAD_ROUTINE DHS_Camera (void * parameter );
  #endif
***************
*** 213,218 ****
--- 214,220 ----
   deviceTables["MAR345"] = (XOS_THREAD_ROUTINE_PTR)MAR345;
   deviceTables["CCD"] = (XOS_THREAD_ROUTINE_PTR)Quantum4Thread; //i.e. Quan
```

```
   deviceTables["Quantum315"] = (XOS_THREAD_ROUTINE_PTR)Quantum315Thread;
+  deviceTables["MARCCD"] = (XOS_THREAD_ROUTINE_PTR)MARCCDThread;
  #endif
```

5. The `dhs_Quantum315.cc` file was copied to `dhs_MARCCD.cc` and the Quantum315Thread procedure was modified to MARCCDThread.

6. The final step involved coding handles for each operation command and appropriately controlled the MAR CCD.

12 CVS software projects

The Distributed Control System is composed of various software components or 'projects' within the CVS repository. The projects related to DCS are listed here.

- xos: library of system calls.
- auth: Some simple authentication procedures.
- tcl_clibs: C functions loadable by TCL.
- dcs_tcl_packages: tcl functions used by scripting engine and BLU-ICE.
- dcss: The distributed control system server.
- dhs: SSRL's Distributed hardware server.
- simdhs: A TCL program acting as a DHS, simulating motors and other common hardware.
- widgets: Graphical TCL widgets used by BLU-ICE.
- blu-ice: The GUI client interface to DCSS.
- jpegsoc: Uses jpeg-6b library to send and receive JPEGS over a socket connection.
- diffimage: Loads diffraction images.
- imgsrv: The diffraction image server.

13 Packages needed by BLU-ICE

- Verify that you have a recent version of TCL installed on your computer.

```
tclsh
% info tclversion
8.3
% exit
```

- Verify that the third party library Itcl[7] is installed correctly on your computer.

```
 bl92a:~ > wish
% package require Itcl
3.2
% exit
```

- Verify that the third party TCL library BWidget[8] is installed correctly on your computer.

```
bl92a:~ > wish
% package require BWidget
1.2.1
% exit
bl92a:~ >
```

- Verify that the third party TCL library BLT[9] is installed correctly on your computer.

```
bl92a:~ > wish
% package require BLT
2.4
% exit
bl92a:~ >
```

[7] http://sourceforge.net/projects/incrtcl/
[8] http://sourceforge.net/projects/tcllib/
[9] http://sourceforge.net/projects/blt/

- Verify that the third party TCL library Img[10] is installed correctly on your computer.

```
bl92a:~ > wish
% package require Img
1.2.4
% exit
bl92a:~ >
```

[10] http://www.xs4all.nl/ nijtmans/img.html

14 Document Version Information

```
Working revision: 1.17 Mon May  2 18:25:38 2005
```