Scripting reference

CTIO 60 inches Echelle

ECH60S-3.1

La Serena, December 2009

Contents

Introduction

The following document is a reference to the CTIO 60 inches ECHELLE scripting. It provides a way of understanding the tools that allow to make scrips to handle the complete ECHELLE application.

Scripting is an alternative way to the GUI for handling the Echelle software. It has the advantage, or the potential, for building much more complex tasks than the GUI. It provides also a way of pretty much "automate" a good part of the work.

When using a script, if the GUI is opened (as it will usually be) it will show what the software is doing, so the GUI is still useful to have a feedback on what is going on.

The scripts will talk directly to the application, not to the GUI; the GUI runs "in parallel" to the scripts, in the sense that they both talk to the main application, and the GUI knows and reflect the activities only because it receives -as the normal operation of the GUI- asynchronous messages.

All the application is ascii-commands driven, which means that everything can be done with scripts -taking images, handling lamps and temperatures, even moving the telescope-

The philosophy is simple: **the user can use any script he likes**. **There are no constrains in the language or the scripting type**. This is possible because the software receives the commands in raw tcp/ip sockets, so:
a) any language than can handle a socket would do (scripts like python, pearl, tcl, and any compiled programming langiage)
b) The software **provides wrappers that can be called as any system call**, which make possible for scripting languages that cannot handle sockets as easily to also handle the system (csh, bash, cl, etc)

Since these wrappers are provided, a language like python can also use them instead of handling the socket directly (just by making a simple OS system call to these wrappers). This makes possible, of course, to send a command manually using any terminal.

Note: the term "app_root" that will appear repeatedly in the next pages is, for this application, /home/observer. Also, in the commands syntax description there will appear some symbols:

< >: encloses a mandatory field
[ ]: encloses an optional field, or a field that may or may not be present

| : separates valid options for a command

Chapter 1: The wrappers

1.1 Brief internal description

We will explain a bit how the wrappers work, without this is not needed for the maker of scripts. We will explain it just for the ones that like to know the implementation details. For those who are not interested in these details, please skip this and go directly to ***1.2***

In the bin directory specific to the echelle application (***/app_root//apps/ECHELLE/bin***) there is a binary called "**sendsockcmd**". This is a small binary that, when called, opens a socket, sends the given argument to that socket, wait for the response and when received, print the response out, in the standard output -the screen. Finally it closes the socket channel.

The Echelle software has a device whish only job is to listen for tcp/ip messages, send those messages into the SML-world (internall protocol of the application) and then transmit the response back into the tcp/ip socket. This device is called SYNCDEV (see ***ECH60S-2.0***, software architecture).
Every time the application is started ("start_ECHELLE) it creates, on the fly, a csh script called ECHELLE, on ***app_root/apps/bin***

This is the main and primary wrapper, because it allows to talk to any device, or part of the application. It is a very simple script, and looks similar to this:

```
#! /bin/csh -f
#This script is automatically generated
#Any edition to this file will be lost!
set command =  ""
while ($#argv  > 0)
        set command = " $command $argv[1]"
        shift argv
end
/home/observer/apps/bin/sendsockcmd -h localhost -p 1325  "$command"  -t 40000
```

The service port "1325" here, is read directly from the SYNCDEV configuration file. So as it is easy to see, it just passes to "**sendsockcmd**" any argument given. All the other wrappers are based on this one, and they are, actually, not really necessary.

1.2 ECHELLE

This is the first and primary wrapper. It allows to send any command to any available device. The syntax for the commands is:

ECHELLE <device> <command> <args ....>

where

device: any available device in the application:

- PAN: detector and temperature commands (PANDEV)
- LAMP: lamps commands (ECHLAMP)
- TCS: Telescope Control System (TCS) commands (TCSCT60DEV)
- IODCELL: Iodine Cell motor commands (IODCELLDEV)

command: any command for the specified device

The response syntax is:

<DONE | ERROR> [message]

or the requested information

For examples, please see chapter 2. For the detailed commands for each device, see Appendix A or ***ECH60S-2.0*** document, with the detailed description of each device.

1.3 dhe or DHE

This is a wrapper for all the PAN (or DHE: detector head electronics) commands. The syntax is:

dhe <command>[args ...]

where

command: any available PAN (and panview) command (see Appendix B for panview commands)

This is the same as

ECHELLE DHE <command> [args ...]or

ECHELLE PAN <command>[args ...]

1.4 lamp or LAMP

This is a wrapper for all the LAMP commands. The sytax is

lamp <command>[args ...]

where

command: any available LAMP command

This is he same as

ECHELLE LAMP <command>[args ...]

1.5 tcs or TCS

This is the warpper for all the TCS commands. The syntax is

tcs <command>[args ...]

where

command: anay available TCS command

This is the same as:

ECHELLE TCS <command>[args ...]

1.6 pan

This wrapper talks directly to panview, actually bypassing the application itself. This is not bad just because panview sends async. Messages to the application, so the application will know anyway. Using "pan" is the same as using.

The syntax is

pan <command> [args ...]

where

<command>: any available panview (PAN) command. See Appendix A for a summary of the most useful panview commands.

Using this wrapper produces the same effect as using:

ECHELLE PAN <command> [args ,,,]

1.7 panview macros

Panview has the capability of handling simple macros. A "macro" is basically a set of valid commands, one per line, that are executed in order, just as if they were sent through the command line. Panview can then read the macro file, execute all the commands in it, and then return with a final, unique response. The macro files are identified by the extension ".mc". The file can be specified using an absolute path or a relative one. If using a relative path, it will assume as the "basepath" a default macros directory defined in its configuration file. If the macro in the default directory, then just giving the name is enough. To invoke the command execution:

pan appmacro <macro_name> or
ECHELLE PAN appmacro <macro_name>

For example

pan appmacro geometry_iodine

will look for the macro file called geomtry_iodine.mc on the default macro directory.
The default macro directory in this application is $HOME/panview/fpas/_echelle/config/DETECTOR.

The command

pan appmacro /home/observer/bin/mymacro

will actually execute the macro mymacro.mc on /home/observer/bin

The macros can be called recursively (I.e, inside a macro there can be another macro call). Using macros is an easy way of saving time. Of course an alternative is writing a script instead of a macro. Another option, usually better, is a combination of both (scripting and macro calling). See the C***hapter 3*** for examples on this.
Also, the echelle software implements important functions through macros. See ***ECH60S-6.0*** on panview setup

Chapter 2: the sockets and the protocol behind the scenes.

This chapter may be useful only to those who would like to skip the wrappers and handle the sockets directly. This allows for a more efficient handling (it is possible to have more than one action at a time, while with the wrappers this is it possible.

For those who are not interested in the internals of the protocol and are not planning to handle the sockets directly, please skip tis chapter and go directly to ***Chapter 3: Examples***

2.1 The protocol

The protocol that we will describe here is the general protocol of the commands and the messages between the devices; handling the socket directly requires having a basic knowledge of this protocol.

We are not talking here about SML, but just the command / response syntax.

First, some terminology.

- Command: a requested action
- immediate response: the immediate response sent in return of a command. This response should not last more than about 5 secs to be received
- Callback: this is a later, final response for commands that take long time in finishing (like exposing an image or offset the telescope). This are then messages associated to an original command
- asynchronous messages: these are messages not associated to any specific command, but spontaneously generated, usually to indicate “abnormal” conditions, like an alarm, etc

The protocol has two communication channels:

a) command / response: this is a bidirectional channel. Here the client send the command and receives the immediate response

b) asynchronous: this is a unidirectional channel, from server to client. Here are returned the callbacks and also the asynchronous messages.

There are two types of commands:

a) short commands: the requested action is finished when the response is returned

b) long commands: the requested action will take some time, so it is first returned an immediate response, and then a callback when the action is actually done. See point ***2.1.2*** (responses)

The system implements two tcp servers, in two different ports (command/response and asynchronous). The client will use the asynchronous channel to receive only -any message going from the client to the server in the asynchronous channel will be ignored. Both servers support multiple clients.

2.1.1 Commands

The command syntax is

<device> <command> [args ...]

where

device: device to which the command is directed

command: command to send

args: optional command arguments

The available devices for this application are: PAN, TCS and LAMP. In Appendix A and B are the lists of the available commands for each device. This is also on document ***ECH60S-2.0*** (software architecture). In ***Chapter 3*** there are some examples

2.1.2 Responses

The response syntax is

short commands

on success: ***DONE | <requested information>***

on error: ***ERROR <error_message>***

long commands

on success:

- immediate response: ***OK <estimated_action_time_in_msecs>***
- callback: DONE [message] (<device:<command>>)

on error

- immediate response: ***ERROR <error_message>***
- callback: ***ERROR [message] (<device:<command>>)***

For the long commands, then, it is first returned an “OK” with an estimation of the time it will take to finish the action. This number could be used by the client toimplement their own timeout mechanisms, although this is not recommended since the system has already timeouts included, and it should, normally return the timeout error. The final callback is received throuhg the asynchronous channel with the device and command reference at the end. Example

DONE image_Done (DHE:EXPOSE)

Note that the PAN commands may return the ID as “DHE” (Detector Head Electronics).

2.1.3 Asynchronous messages

The syntax for the callbacks was explained below,so here we will explain the truly asynchronous messages (not associated to a command):

ASYNC <message> [<< ID]

The message will always start with the ASYNC key (remember that in this channel also the callbacks are received,and those will never start with this key). Then it will come the body of the message, and at the end it may appear the ID of that who generated the message. The only one that, in general, will add its ID is panview (so, messages coming from PANDEV)

2.2 Handling it

So, now that we know the basics, the client that want to handle the system directly (without the provided wrappers) will need to open two sockets, one to the command/response server port and one to the asynchronous server port. It will need to send the command and parse the immediate responses in the first,and handle the async. Messages and callbacks in the second. The actual service ports in use are in

app_root/apps/ECHELLE/config/DEV_SYNC.cfg

(this is also explained in ***ECH60S-2.0***). Inside this file, the keywords are

[COMMS]
port=<command/response port>
asyncport=<asynchronous port>
blockport=<blocking port>

The "blocking port" is an special port that "blocks" the long commands. It basically hides all the command/response/async. Protocol; when the client sends a command -even if it islong- the server will block until the action is done. This allows a much easier handling, but of course the truly asynchronous messages are ignored -lost-. **This is indeed the port that the wrappers use, so no need of multiple sockets handling if using this port**. The disadvantage is that it is of course less efficient, since the client needs to wait for the action to be finished before requesting a new one (so things like moving the telescope while the data is being read cannot be done using this port or, for the same reason, using the wrappers).

Note that any of this ports can be set in the config file as environmental variables, allowing more flexibility in the handling of the system.

Chapter 3: Examples

This chapter provides short examples on scripting.

3.1 csh

Csh is a very common scripting language. It is easy to use when easy operations are requiered. A very simple example is the following script, build to set the system in a very specific operation mode. The script is called “iodine”

!#/bin/csh -f

echo “setting roi, binning and speed mode”

pan appmacro geom_iodine

echo “setting basename”

pan set image.basename ga12.

echo “updating GUI”

pan dhe async send guiupdate basename

This script is used to set the software in “iodine” mode. Note that the script calls a panview macro called geom_iodine. This is a macro that can be called from the GUI to setup the iodine mode. This script, besides setting up the iodine mode also setup the basename for the images. The geom_iodine macro looks like this

appmacro set_roi 1 400 2048 1200

dhe set binning 1 1

appmacro speed_fast

dbs set geom_mode iodine STR

dbs async send update geometry

The first line calls another macro that setup the ROI. Then the binning is set to 1 1, and the speed is set to “fast” calling another macro. Then the internal database variable “geom_mode” is set to “iodine”. This

value is used for a fits header key. The last line is just so the GUI gets updated (updates itself when it receives that command)

It get obvious by now that any scripting language can be used by calling the wrappers. We will still see another example, in CL

3.2 CL

CL is the scripting language from iraf. Using it has the advantage of acquiring and reducing (using iraf tasks) without the need of calling outside languages (acquiere and reduce in iraf). Of course this can also now be done using pyraf, which allows to use pythin language and yet being able to call iraf tasks (and python tasks!). The following example is just cl iraf scripting. It uses an iraf task called “socksend” to send the command to the application (through a socket). The following script can takes biases, darks and flat (calibration) as specified

 # echcal.cl

procedure echcal()

bool bias {"yes", prompt="Take bias images?"}

int biasnum {"5", prompt="Number of bias images"}

bool flat {"yes", prompt="Take flats?"}

int flatnum {"5", prompt="Number of flats"}

real flatet {"50.0", prompt="camera flat exposure time"}

string lamp {"QUARTZ", enum="QUARTZ\|TH-AR", prompt="Lamp to use

"}

bool dark {"no", prompt="Take dark images?"}

int darknum {"5", prompt="Number of darks"}

real darket {"300.0", prompt="camera dark exposure time"}

begin

int timeout = 200000000

```
printf ("\n")
printf ("Please make sure that the dome is dark\n")
printf ("<ENTER> to continue\n")
soksend.host = get_arcinfo.ssvrhost
soksend.port = get_arcinfo.ssvrport
if (bias) {
printf("Setting title to Bias\n")
soksend(command="DHE set image.title bias")
printf("Setting type to Bias\n")
soksend(command="DHE set obs.obstype bias")
printf("Setting basename to Bias\n")
soksend(command="DHE set image.basename bias")
printf("Setting exposure time to 0.0\n")
soksend(command="DHE set obs.exptime 0.0")
printf("Setting number of bias images to %d\n", biasnum)
soksend(command="DHE set obs.nimages " // biasnum)
soksend(command="DHE expose", timeout=timeout)
}
if (flat) {
printf("Setting title to domeflat\n")
soksend(command="DHE set image.title domeflat")
printf("Setting type to Calibration\n")
soksend(command="DHE set obs.obstype Calibration")
printf("Setting basename to flat\n")
soksend(command="DHE set image.basename flat")
printf("Setting exposure time to %5.1f\n", flatet)
soksend(command="DHE set obs.exptime " // flatet*1000)
printf("Setting number of domeflats to %d\n", flatnum)
soksend(command="DHE set obs.nimages " // flatnum)
printf("Turning %s on\n", lamp)
soksend(command="LAMP set " // lamp // " on")
```

```
soksend(command="DHE expose", timeout=timeout)
printf("Turning %s off\n", lamp)
soksend(command="LAMP set " // lamp // " off")
}
if (dark) {
printf("Setting title to dark\n")
soksend(command="DHE set image.title dark")
printf("Setting type to dark\n")
soksend(command="DHE set obs.obstype dark")
printf("Setting basename to dark\n")
soksend(command="DHE set image.basename dark")
printf("Setting exposure time to %5.1f\n", darket)
soksend(command="DHE set obs.exptime " // darket*1000)
printf("Setting number of darks to %d\n", darknum)
soksend(command="DHE set obs.nimages " // darknum)
soksend(command="DHE expose", timeout=timeout)
}
soksend(command="DHE set obs.obstype object")
soksend(command="DHE ASYNC SEND UPDATE")

end
```

The task “soksend” just sends, through a socket, the specified command to the software

Appendix A: available commands

The description of this commands can be found on the ECH60S2.0 document, in a device-by-device description. Here we will just present a quick reference. In the examples, the “>” symbol represent just the line prompt of the terminal (as if typed manually)

A.1 TCS

info: gets the current information

example:

> ECHELLE TCS info

A.2 LAMP

list

gets the list of available names (lamps, manual switches, motor)

example:

> lamp list

> name= QUARTZ

name = TH-AR

name = SW_TH-AR

name = SW_QUARTZ

name = MOTOR

set <TH-AR | QUARTZ > <on | off>

example:

> lamp set QUARTZ on

> DONE

get <TH-AR | QUARTZ | SW_QUARTZ | SW_TH-AR | MOTOR>

example:

> lamp get QUARTZ

> ON

status

shows the status of the device and of all the available "names" in it, as "name <on | off> status"

example:

> lamp status

> IDLE

TH-AR OFF OK

QUARTZ ON OK

SW_QUARTZ OFF OK

SW_TH-AR OFF OK

MOTOR ON OK

A.3 IODCELL

list

gets the list of available names (iodine cell, manual switch)

example:

> ECHELLE IODCELL list

> name= IODINE

name = SW_IODINE

set IODINE <in | out>

example:

> ECHELLE IODCELL set IODINE in

> DONE

get <IODINE | SW_IODINE>

example:

> ECHELLE IODCELL get IODINE

> ON

status

shows the status of the device and of all the available "names" in it, as "name <on | off> status"

(on=IN, off=OUT)

example:

> ECHELLE IODCELL status

> IDLE

IODINE OFF OK

SW_IODINE OFF OK

A.4 PAN (panview)

This is a very long list. Following there is a quick summary. Any of this commands can be invoked using the "pan" or "ECHELLE PAN" wrappers:

example (very first listed command):

> pan set obs.exptime 5000

> DONE

The following list describes briefly the command and parameters available for the detector controller electronics. For a more detailed description on the parameters, please see the complete DHE command list

Here is presented only a reduced list to setup observations and observe. The low level and engineering commands are not listed. NIR-related commands are also not listed

The meaning of the used symbols is:

< > : holds a parameter which is necessary (mandatory)

[ ] : holds a parameter which is optional

| : separates possible values for an enumerated type.

A.3.1 Observation setup

SET / GET (dhe set / dhe get)

All the following parameters can be set or read, unless specifically stated.

obs parameters

complete command	alternates "short"	description	example
obs.exptime <*uint* | *float*> [[s]]	exposuretime, exptime	exposure time in msecs, or secs if specified ([s])	dhe set obs.exptime 22.0 [s]
obs.display <on | off>	displayimage, display_image	real time display on or off	dhe set displayimage off
obs.numreads <u*int*>	imagestoread	number of images to read in the next sequence	dhe set obs.numreads 2
obs.autowrit <on | off>	write_to_disk, autowrite	writes the images to disk or not	dhe set autowrite off
obs.observer <*string*>	-	observer field (for image headers)	dhe set obs.observer Burrus
obs.obstype <*string*>	type	obstype field (for image headers). Only "dark" has real meaning (no shutter)	dhe set obs.obstype dark
obs.roi <xstart><ystart><xlen><ylen>	-	Region Of Interest (subwindow). Coords. in unbinned units, from lower left corner	dhe set roi 513 513 1024 1024
progress	-	returns the progress of the image/exposure. (READ ONLY)	dhe get progress
obs.autoobsid	autoobsid	sets auto observation ID on/off	dhe autoobsid off
obs.obsidbase		sets the basename for the obs. ID	dhe set obs.obsid myObs
obs.obsidcounter		sets the initial value for the obsID counter	dhe set obs.obsidcounter 10

image parameters

complete command	alternates "short"	description	example
image.title <*string*>	imagetitle	image title, to be the "object" field on the headers	dhe set title M51
image.comment <*string*>	imagecomment	image comment, to appear on the headers	dhe set imagecomment this is a test
image.directory <abs path>	-	directory where image will be stored	dhe set image.directory /home/images/
image.basename <*string*>	-	basename for image name	dhe set image.basename junk
image.rootname <directory/basename>	rootname	complete rootname, which is directrory + basename	dhe set rootname /home/images/junk
image.prefix <*string*>	imprefix	prefix for basename	dhe set imprefix red_
image.sufix <*string*>	imsufix	sufix for basename	dhe set imsufix _red
image.number <*uint*>	imagenumber, number	number for next image, or initial image if a sequence	dhe set imagenumber 22
image.mext <on | off>	multipleextensions, extensions	image will be single or MEF fits file	dhe set extensions off
image.modifiers <key1 val1 ... keyN valN>	-	data sections of image, like prescan, overscan, etc	dhe set modifiers prescans 22 overscan 50
image.autoexpid <on | off>	autoexpid	sets auto exposure ID	dhe set autoexpid on
image.expidbase		sets the base of the expID (auto if undef)	dhe set image.expidbase myexpID
image.expidcounter		sets initial counter for the expID.	dhe set image.expidcounter 1

dhe parameters

complete command	alternate "short"	description	example
dhe.binning <xbin><ybin>	-	binning in x and y (OPTICAL ONLY)	dhe set binning 2 3
dhe.autoshutter <on | off>	-	opens shutter automatically when exposure starts or not (ex, "off" for darks)	dhe set autoshutter on
dhe.shutter <open | close>	-	opens or close the shutter manually	dhe set shutter open
dhe. readmode <*string*>	-	readmode. The readmode must be supported by the hardware	dhe set readmode lower_left
dhe.timing	timinginfo	information on frame and pixel time (READ ONLY)	dhe get timinginfo
dhe.geometry	-	information on geometry -topology- (x and y size, etc) (READ ONLY)	dhe get geometry
dhe.config	controllerconfig	information on controller configuration (READ ONLY)	dhe get dhe.config
status	-	information on current status of the controller (READ ONLY)	dhe get status

A.3.2 Observation control

command	description
EXPOSE	starts the exposure sequence
ABORT	aborts all activity (current exposure/readout/sequence). Current image get lost
STOP [sequence | exposure]	stops sequence but waits for current exposure/readout to finish (sequence) or stops the current integration and reads out (exposure)
PAUSE	pauses exposure, closing the shutter and stopping the exposure count. (OPTICAL ONLY)
RESUME	resumes a paused exposure (OPTICAL ONLY)
RESET <controller | pci>	resets controller or PCI card

A.3.3 Fits commands

The "key add" and "key delete" commands are permanent, because they affect the actual file template

command	description	example
fits key add <KEY> <datatype> <value> // <comment>	Adds (or edit if key already exists) the specified KEY to the specified value (and datatype) and comment	fits key add TEST FLOAT 2.0 // this is a test
Fits key delete <KEY>	Deletes the specified KEY from the template	Fits key delete TEST
Fits key get preview [ext]	Shows how the headers will look like for the next image (keywords, values, comments and datatypes). If MEF fits, the extension can be specified (starting from 0)	Fits get preview
Fits key get all	Shows the actual template (the KEYS and sources, not the current values as the preview)	Fits key get all
Fits get hdrfile [-full]	Shows the header file template. If "-full" is specified shows the complete (absolute) path to the template	Fits get hdrfile -full

References

- panview documentation