Administrator's Manual

Software Architecture

CTIO 60 inches CHIRTEMP (CHIRON)

CHI60S-2.1-T

La Serena, December 2010

Contents

Introduction

The following document is a reference to the CTIO 60 inches CHIRTEMP Software structure. It provides a way of understanding its' internal structure and configuration

This document describes each of the devices present in this application. It also describes the directories structure and where to find binaries and configuration files.

Note that that several of the devices described are identical to those of the CHIRON application (manual CHI60S-XX)

Chapter 1: Software general architecture

Introduction

The Chirtemp 60 inches software is based on software modules, called “devices”, that talk to each other using a protocol called “SML”. Each device is in charge of an specific task. Each device is independent on one another, being the SML protocol the only way of “contact” between them. In this way, the software is totally modular. The “core” of the software (called “ENV”) only starts the devices at boot time.

The Chiron software is composed of two separate applications: CHIRTEMP and CHIRON. The first is exclusively in charge of the temperature handling/logging/alarming, etc, while the second is in charge of the operation of the instrument. It was done in this was just so the temperatures are always being run/recorded independent on the fact that the instrument software may be down due to problems or simply because the instrument is not in use. The present document describes CHIRTEMP only. For a description of the CHIRON application, please see document ***CHI60S-XX***

In ***Figure 1.1*** is shown a general diagram of the software. Each device is represented as a box. The **SML** protocol is represented as the upper redish line that connects all the boxes (devices) inside each application. Each device was designed for handling a very specific part of the hardware or functionality. The name of each box self-explains the purpose of each device. The “external” clients can talk to the software using raw tcp/ip, allowing easy access to scripts. The software also provides wrappers that encapsulates the tcp/ip, making even easier for the scripts or external clients to access all the functionality. Details on this wrappers is provided in ***Chapter 2***, and details on scripting is provided in document in ***CHI60S-3.0*** (scripting). In the following points we will briefly explain each device and application.

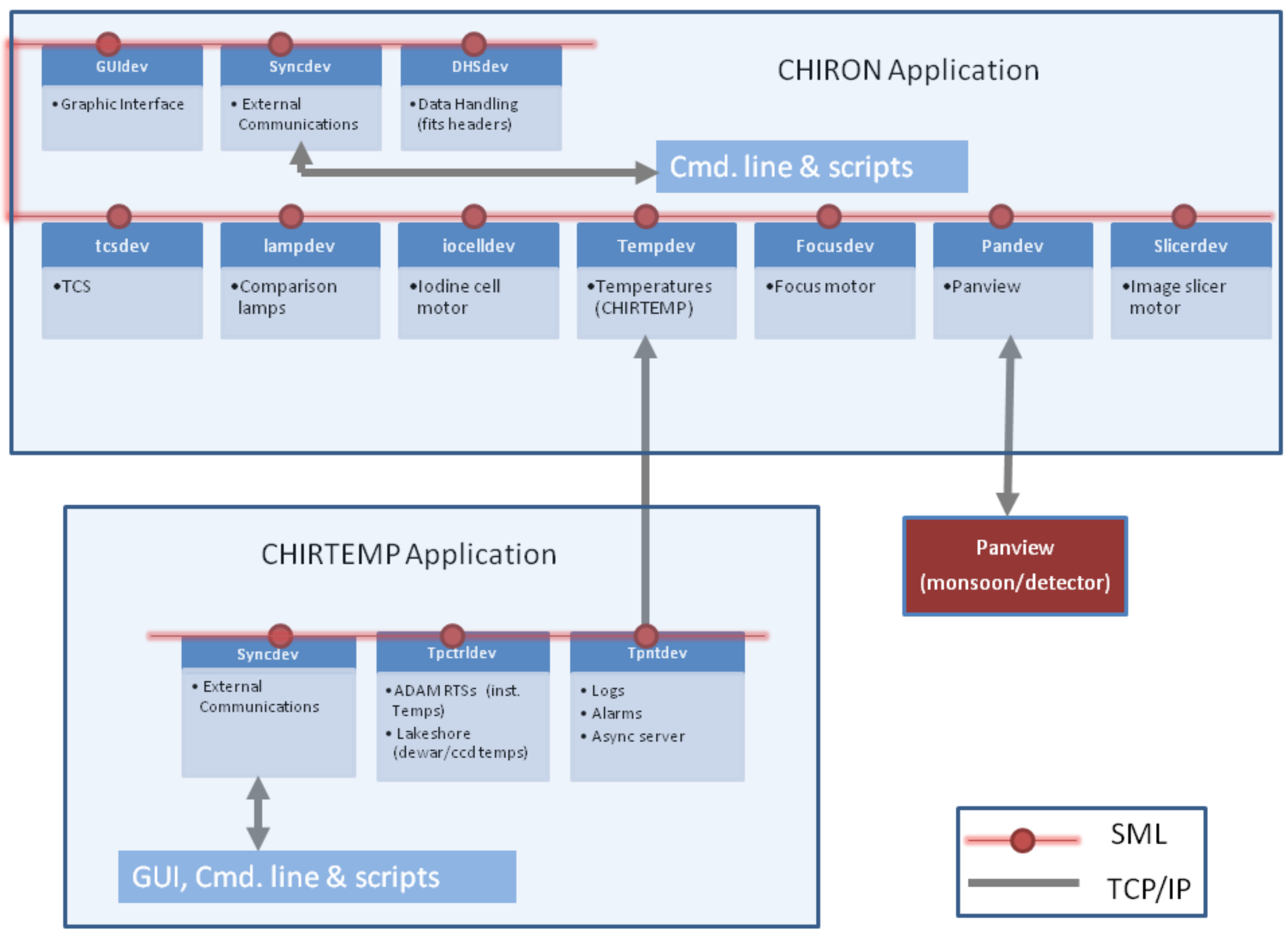

Figure 1.1: software diagram

1.1 COMMSDEV (Communications Device): SYNCDEV

1.1.1 Description

This device's only function is to provide an interface between the "external world" and the SML "bus" (so, the devices themselves). This device has a multiple clients tcp/ip server that allow to receive command and send responses, and also to send asynchronous messages. In general, the communications protocol is based on two channels: a command/response channel, and an asynchronous messages channel. This device, then, receives the command through the tcp/ip channel, and passes that command -using SML- to the appropriate device, passing then the response back from the device (SML) to the client (tcp/ip). It also passes an SML "async" message into a tcp/ip async. Channel.

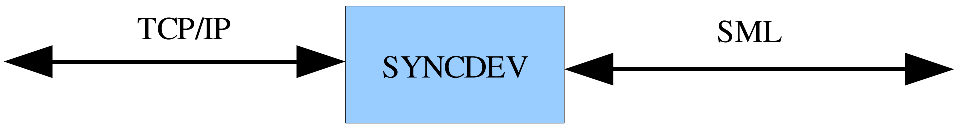

Figure 1.2: communications device: syncdev

The wrappers provided to easy scripting "hide" all this, giving to the user a single point to handle the software without the need to worry about the protocol details, other than the command and response syntax. See next chapter for examples.

1.1.2 Configuration file

The configuration file is called **DEV_SYNC.cfg** and it is located in the standard application's configuration directory (see next chapter)

```
[COMMS]
port=1920                 // tcp multi-client command/response service port
asyncport=1930            // tcp multi-client async. Server port
blockport=1940            // tcp single client blocked port
maxcmdsvr=2               // maximum amount of command/response clients allowed at a time
maxasyncsvr=2             // maximum amount of async. Client allowed at a time

[LOG]
log=yes                   // enable logging?
file=__LOGPATH/DEV_SYNC.log          // file log path. See next chpater for "__LOGPATH" definition
```

1.2 TPCTRL Device (TestPoint Control Device): TPCTRLDEV

1.2.1 Description

This device is in charge of handling (reading/setting) the “testpoints”. A “testpoint” is any point that can be monitored in any hardware that desires so. To interact with the testpoint it is usually required some external hardware, so this device have some “submodules” that allow to talk to the specific hardware. In the case of the current application, all the testpoints are temperatures. These temperatures includes two types:

a) instrument temperatures: RTDs distributed inside/outside the instrument itself. These are all read-only (no setpoints) and are read through an etehrnet ADAM 6015 module. This module is installed in an external box, described in document ECH60HF-7.2 (echelle dataio/rtds controlbox)

b) camera temperatures: Diodes inside the CCD camera, that read the temperature of the CCD and the NECK of the dewar. Besides that, it can set the CCD temperature (the heating element is a resistive wiring). All this is done through a Lakeshore 325 temperature controller. A detailed description of the Lakeshore settings and hardware involved can be found in document CHI60SHF-XX (Chiron temperature control). The Lakeshore controller has a serial interface (RS-232)

The TPCTRLDEV device provides the connectivity to the hardware, and a periodic loop requesting the current value of each defined testpoint. The logging itself is done through a different device (see 1.3). The user can request the value of any testpoint at any time (or change the CCD setpoint)

1.2.2 Configuration file

The configuration file is called DEV_TPCTRL.cfg, and it is located in the standard configurations directory (see next chapter on software tree). It is based on sections and key/value pairs (as most of the devices configurations.

```
[DEVINFO]

tpparams="monitor=on, updaterate=180, delay=30"

instparams="localupdate UPDATERATE 18; TP suscribe; TP add -tptype all"

cmds2exec=INIT
```

General settings. The “tpparams” are parameters that will be passed to the logger device (TPNT, see 1.3).

“instparams” are local parameters. The ones there are saying to make an update of every testpoint every 18 seconds (this is, go read the hardware),and then to suscribe to the logger module (TPNT) all the defined testpoints.

```
[LOG]
log=yes
file=_LOGPATH_/TPCTRL.log
```

Log in the standard loggin directory

```
[ALGORS]
file=ALGORS.list
```

Any required conversion algorithm will be defined in this file (ALGORS.list)

```
[MODS]
LKS325="init type serial,  brate 9600, port 1"
ADAM6000="init type udp, address 139.229.3.236, port 1024, timeout 5000"
```

This defines the actual hardware modules to be used. Here it specifies the LKS325, which is the hardware submodule that handles the Lakeshore 325 controller,and the ADAM6000 module which is the submodule that handles the ADAM 6000 models family.

The it passes to each of them the parameters used to connect (the parameters values will of course depend on the connection “type”, like in the example serial for the Lakeshore and udp for the ADAM module

Following is the definitoion of the actual testpoints. This testpoints are, on run time, “suscribed” to the TPNT device in order to be monitored, according to the given directives (see last line of each testpoint definitoon)

```
[TEMPAIR]               /*name of the testpoint*/
module=ADAM6000         /*what module -defined under MODS- to use*/
type=temp               /*testpoint type*/
outunit=c               /*unit in which the value is given to the user*/
sensorunit=sensor       /*unit in which the hardware is read*/
address=0x0             /*address of the point in the hardware*/
localupdate=true        /*update it in the local regular update -read it every time*/
range="0 12 30"         /*range allowed for the value. If out of range, it can generate an alarm*/
```

```
permissions=R                   /*read-only*/
filter=average                  /*do a moving average of the readings*/
samples=10                      /*using 10 samples*/
TPparams="monitor=on, alarms=on, log.file=_LOGPATH_/insttemp.log, log=on, limit.alarmtype=async error"
```

The Tpparams are parameters passed to the logger (TPNT) module:

monitor this testpoint, generate an alarm if it is out of range, log the values in the specified logfile, and when an alarm is generated, produce an asynchronous message

The rest of the points are similar to this. We will just show here 1 more, belonging to the lakeshore controller

```
[CCDSETP]
module=LKS325
type=setpoint
outunit=c
sensorunit=c
address=0x0
localupdate=true
range="-140 -100 -80"
permissions=RW
TPparams="monitor=on, alarms=on, log.file=_LOGPATH_/dettemp.log, log=on, limit.alarmtype=async error"
```

So it is very similar to the others, but this is RW since it can be written to (it is s setpoint)

1.2.3 Header Information

This device can send to the Data Handling System device (DHSDEV, see ***1.5***) the current status of each testpoint; however, in this application there is no direct generation of header information, since the values are passes asyhnchronously to the CHIRON application through the async. Server of the logger (TPNT) module. It is then CHIRON application the one that generates the header information.

1.2.4 Available commands

<> indicates an obligatory field
[] indicates an optional field
| separates argument options
Commands are case sensitive.

Prefix: TPCTRL

get <name> [-params]
description
gets the last value read from the specified testpoint. If “-params” is specified then it gets the settings (configs) of that testpoint (module, address, etc)
return value
<value> or ERROR <error message>

read <name>
description
reads from the hardware the reading of the specified testpont (forces a hardware read)
return value
<value> or ERROR <error message>

get -list
description
gets the list of all the deined testpoints
return value
DONE or ERROR <error message>

1.3 TPNT Device (Testpoint Monitoring Device): TPNTDEV

1.3.1 Description

This device has as the only function to monitor other devices that suscribes to it. It provides then a loop for reading the suscribed points, logging and alarms. In this application, the only device that suscribes to TPNT is the testpoint control device, TPCTRL (See 1.2)

1.3.2 Configuration file

The configuration file is called **DEV_TPNT.cfg,** and is located in the standard config directory of the application. This is a very simplke config file, as all the points are usually suscribed at run time by other modules (see the TPCTRL config file descrioption)

The relvant parts are

```
[LOGS]
maxsize(kb)=1024
```

States the maximum size of a testpoint logfile before being renamed (by the renaming date)

```
[LOG]
log=yes
file=_LOGPATH_/DEV_TPNT.log
```

Where to leave the device's own log file

```
[COMMS]
enabled=true
port=7000
async=true
```

This is important. Here it states that the asynchronous tcp/ip server will be enabledat port 7000, and that asynchronous messsages will be sent every time a new reading is available (so it will keep sending an async. Stream with the value of the testpoints). This is where the TEMP Device on the CHIRON application connects to in order to get the value of the testpoins and format the header information. See the TEMPDEV information on the CHIRON application manual

1.3.3 Header Information

In this application the header information is generated by the TEMPDEV device in the Chiron application,based on the values obtained fro the TPNT async.port. So no header information directly here.

1.3.4 Available commands

<> indicates an obligatory field
[] indocates an optional field
| separates argument options
Commands are case sensitive.

Prefix: TPNT

sample
description
forces a reading of all the testpoints. This then forces an asynchronous message with all the new values, causing the connected clients (in this application, TEMPDEV at CHIRON application) to update themselves.
return value
DONE, or ERROR [message]

getall
description
gets all the defined (suscribed) testpoints
response
<values> or ERROR [message]

1.4 ENV (core)

1.4.1 Description

ENV is the core of the application. It does not maps to any specific hardware or functionality, but defines what devices will be available and started at software startup time. It also provides a way of talking to all the devices at a time, as for broadcasting a system command (Offline/shutdown, etc. See documentation on SML devices).

1.4.2 Configuration file

The configuration file is where the standard directory for configurations is. This directory is actually defined here. The configuration file is called ENV.cfg, and can be considered the first, or master configuration file

```
[APP]
name=CHIRON             // defines application name
path=../ArcVIEW/                  // define path to application (sources) root
[ENVIRONMENT]
MainVisible=False                 // show main window?
[TRANSLATIONS]
APP=PAN                           // defines translations, or "aliases" to the devices. See below for more details
[VARS]                            // This defines some "global" variables, that any module can see
__MODPATH=__APPPATH/modules       //where the modules (devices) are
__CONFPATH=./                     // where the configuration directory is.
__LOGPATH=../log                  // where the log directory is.
[DEVICES]                         // defines where the devices are
file=ENV_DEVICES.cfg              // or the file where the available devices is defined.
```

Note that here is defined the configuration directory as "./", which means "this directory". This means that where this directory is, all the config files for the other modules will also be. Note also that here it is defined the global variable "__LOGPATH" that all the devices are using to define their log directory. The "translations" entry defines other names that the device can have, meaning that if a command with those names arrives it will be routed to the defined device.

The file that defines what devices are available is here set as ENV_DEVICES.cfg (which is the same as ./ENV_DEVICES.cfg -in the current directory-
In this file each section defines a device. The name of the section is the device's name.

```
[SYNC]                                              // device name
Path=__MODPATH/SYNCDEV/public/vis/SYNC_Device.vi    // path to the device's main vi (API)
Commands="START; INIT"                              //commands to pass at load time
[COMSTCP]
Path=__MODPATH/COMSTCPDEV/public/vis/COMSTCP_Device.vi
Commands="START; INIT"
[TPCTRL]
Path=__MODPATH/TPCTRLDEV/public/vis/TPCTRL_Device.vi
Commands="START; INIT"
[TPNT]
Path=__MODPATH/TPNTDEV/public/vis/TPNT_Device.vi
Commands="START; INIT"
```

Note that __MODPATH was defined in ENV.cfg. The command "START" means "load the device". The command "INIT" means "initialize" it. What each device does on initialization depends on the device.

Chapter 2: Software Tree / directories

Here we will give a view of the locations of the different software components and configuration files

2.1 Software tree

Figure 2.1 shows a diagram of the software tree structure

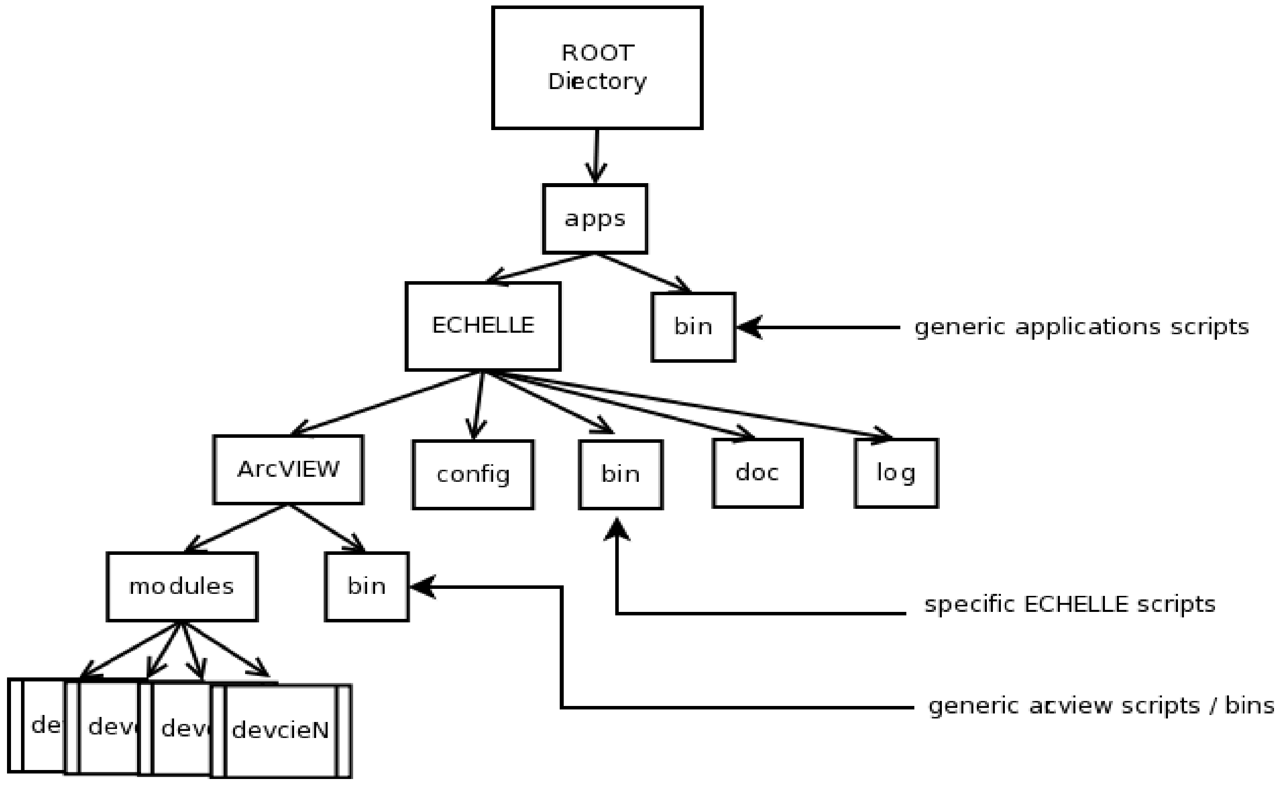

Figure 2.1: Software tree

ROOT directory can be anything. In the case of this application, it is the home directory of the observer's account, **/home/observer/**

2.2 Directories description

apps is the directory where all the applications are. If more than one instrument is in use with the same computer,here it would appear in parallel to the specific CHIRTEMP application

-> **bin:** location of “generic” application scripts. The most important are:

- *start_application* <*name*>: starts the named application ("start_application CHIRON")
- *shutdown_application* <*name*>: shutdown any application ("shutdown_application CHIRTEMP")
- *CHIRTEMP*: this is a wrapper *created at boot time* to talk to the CHIRTEMP application from a command line. See the scripting reference document (***ECH60S-3.0***) ("CHIRTEMP TPCTRL get CCDTEMP")

-> **CHITEMP**: directory of specific CHITEMP application

- ArcVIEW: directory where sources are
  - modules: all devices sources, one directory per device
  - bin: generic arcview scripts (startup and shutdown, etc)
- doc: specific echelle application documentation
- log: all log files of devices (__LOGPATH)
- bin: specific chitemp application scripts:
  - *start_CHIRTEMP*: starts echelle application. This is a "start_application CHIRTEMP" plus some process checking, polling, etc. This is the script actually used to start the application
  - *shutdown_CHIRON*: shutdown chiron application. This is a "shutdown_application CHIRTEMP" plus some process checking, poling, etc. This is the script actually used to shutdown the application
  - sendsockcmd: binary (executable) that is used by the wrappers to talk to the application (opens a socket, sends the command, prints the response, closes the socket).
- **config**: this is the directory where all the device's configuration files are: DEV_TPNT.cfg, DEV_TPCTRLcfg, etc

In general, the user (observer) should not edit any of this. The administrator (maintenance) should be aware mostly of the configuration directory (apps/CHIRTEMP/config) and the specific chirtemp binaries (apps/CHIRTEMP/bin)

The wrapper CHIRTEMP, in apps/bin, provides an easy way of interacting with the application using simple command line (and hence, scripting). This is a simple csh built at boot time, that calls the binary “sendsockcmd” to open the communications channel (socket) and get the response. Example:

```
CHIRTEMP TPCTRL set CCDSETP -115
```

Will open a socket to the application, send the command “TPCTRL set CCDSETP -115”, wait for the response and print it. In this way, using this as part of a bigger script is very easy. All the details in can be found on the scripting reference document ***CHI60S-3.0***

References

- SML documentation
- TCS 60 inches documentation / command list

Glossary

Device*:*

A software component that encapsulates a specific functionality. It must have a very standard and well-defined internal structured and inputs/outputs, so they can be “pluged” into any application that uses devices

SML:

A communications protocol used by the devices to talk to each other. This is a simple protocol that sends ASCII commands and headers. The protocol makes it transparent if the application's devices are in the same machine or distributed among several ones.