European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Instrumentation Framework

ELT ICS Framework - Camera Control Framework- User Manual

Document Number: ESO-319695

Document Version: 2

Document Type: Manual (MAN)

Released on: 2021-05-31

Document Classification: Public

Owner:	Knudstrup, Jens
Validated by PM:	Kornweibel, Nick
Validated by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Appoved by PGM:	Tamai, Roberto
	Name

Release

This document corresponds to `ifw-hl`[1] v3.0.0.

Authors

Name	Affiliation
Jens Knudstrup	ESO/DOE/CSE

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
All	First version

[1] https://gitlab.eso.org/ifw/ifw-hl

Contents

1 Introduction

The purpose of the ICS Camera Control Framework (CCF) is to provide an SDK used for integrating DCS solutions used typically for controlling COTS cameras used in the context of instrument control and for WFS; other use cases of COTS cameras may exist.

Eventhough the main purpose of CCF is to act as an SDK, the necessary tools to cover the most common use cases are provided within the package and in many cases it may be possible to use one of the provided DCS binaries out-of-the-box, but in most cases probably some adaptation in the form of adapters to be implemented, may be needed to cover the specific requirements of the individual context.

1.1 Scope

The scope of this manual is developers, integrating DCS solutions, based on the CCF SDK and for operations personnel, using a given DCS solution, based on the CCF SDK.

1.2 Main Deliverables

In this version of CCF, the following main deliverables are provided by the CCF Project:

- CCF Control Application with GigE Vision support.
- CCF Control Application providing only simulation.
- Data Processing Recipe implementing finding the the centroid of an image.
- Data publishers for FITS files and the DDT.
- A C++ client interface class + command line utility.
- A Python client interface class + command line utility.
- A small tool to generate FITS cube image sequences for the simulation mode.

1.3 Disclaimer

The CCF is a product which is stil under development as well as other ELT SW components. It is expected that modifications will be introduced in the CCF component in connection with future releases, which are not backwards compatible. An example is the configuration handling, which may be re-implemented to use the CII Configuration Service. Other changes will be adoptation of the CII Error Handling Service and CII OLDB Service and final handling of setup parameters. Moreover, some features have not yet been implemented, e.g., some commands of the CII Request/Response interfaces used, are not yet provided.

1.4 Acronyms

DB	Database
CCF	Camera Control Framework
DCS	Detector Control Software
DDT	Data Display Tool
CCS	Central Control System
ELT	Extremely Large Telescope
FITS	Flexible Image Transportation System
GUI	Graphical User Interface
HW	Hardware
ICS	Instrument Control System
LCS	Local Control System
OLDB	Online Database
RAD	Rapid Application Development
ROI	Region Of Interest
SCXML	State Chart XML
SW	Software
WFS	Wavefront Sensor

1.5 Nomenclature

Acquisition Thread	The thread executed internally to handle the acquisition of data from the camera and other interaction.
Adapter	Class derived from a base class provided by the CCF Package, implementing a customisable behaviour and logic of specific CCF DCS instances.
Camera Name Mapping	YAML file used to map the generic names for parameters, used by a Communication Adapter, into the specific parameter names defined by the camera.
Configuration/ Configuration Parameter	The Configuration refers to properties of the system, controlling the behaviour and execution, which are loaded when the application starts up.
Control Application	The CCF Control, main application, ececuted to control and acquire the associated camera. Often referred to as "CCF Control", for short.
Communication Adapter	Implements a specific communication protocol.
Data Publisher Adapter	Implements publishing of data to various types of targets, e.g. FITS files or the DDT. Also referred to as "Data Publisher", for short.

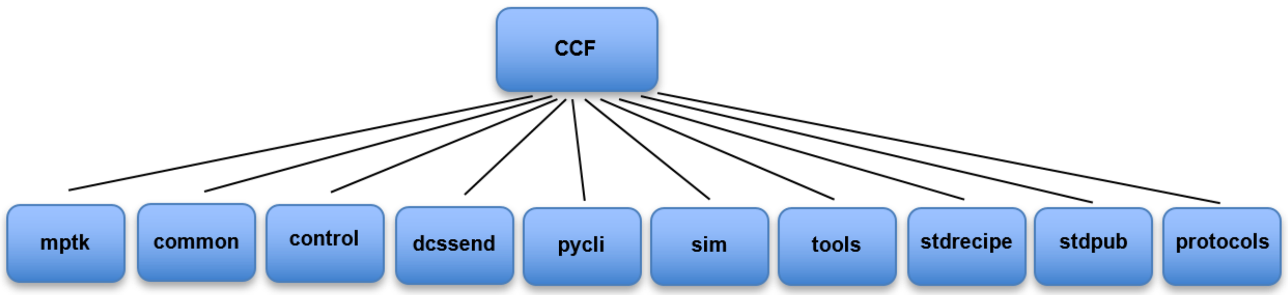
Continued on next page

Table 1.1 – continued from previous page

Deployment Module	A WAF module used for generating a specific version of a CCF Control executable.
Dictionary	Standard ICS Dictionary providing information about parameters to be written in output data product files.
Initialisation Setup	Setup file loaded while starting up the application. It provides default values for the setup parameters.
Metadata Name Mapping	YAML file used to map the generic names for metadata parameters, used (generated) e.g. by a Communication Adapter, into the specific metadata parameter names to be written in the output data product file.
Non-Recording Data Publisher	Data Publisher which transmit the image data to other application software.
Output Data Products	Files generated on disk containing data acquired and possibly processed by the DCS application.
Processing Pipeline	Refers to an internal Processing Thread with its registered Processing Recipes and associated Publisher Thread, in turn, with its registered Data Publishers.
Processing Recipe Adapter	Provides on-the-fly processing of image frames. Also referred to as “Processing Recipe” for short.
Recording Data Publisher	Data Publisher generating Output Data Products hosting the image data.
Recording Session	The execution of Output Data Product generation, following a “recif::RecStart” request.
Setup/Setup Parameter	The Setup refers to properties of the system, controlling the behaviour and execution, which can be altered at run-time.
Standard Adapter	Adapter considered of common interest and thus provided by the CCF Package.
Standard Data Publisher	Data Publisher considered of common interest and thus provided by the CCF Package.

2 Overview

The CCF project is composed by the following modules/packages:



These packages provide the following functionality:

- **mptk: CCF Multiprocessing Toolkit:** A small collection of classes to manage a multithreaded environment, including inter-thread syncrhonisation and communication.
- **common: Common Tools & Utilities:** A set of common utilties used for implementing CCF applications, e.g. an Image Frame Class, a Frame Queue, DB access class, and more.
- **control: CCF Control Application:** Provides implementation of the CCF Control application class and classes/callbacks for handling requests, state machine related activities and implementations of the various types of threads, running in a CCF Control application.
- **dcssend: C++ Request/Response Command Line Utility:** Command line utility to submit requests to a CCF Control application and receive the responses.
- **pycli: Python Client Library & Command Line Utility:** Python module providing a CCF Control client and a command line utility based on that.
- **sim: CCF Control Simulation Deployment Module:** Deployment module for a CCF Control Application, providing simulation only.
- **tools: Miscelleneous Tools:** For now it only provides a prototype of a tool to generate FITS cubes used for execution in simulation mode.
- **stdrecipe: Standard Processing Recipes:** In this version only a centroiding recipe is provided.
- **stdpub: Standard Data Publishers:** In this version a basic FITS CCF Data Publisher and DDT Data Publisher are provided.
- **protocols: Specific Communication Adapters & Deployment Executables:** Provides implementation of CCF Communication Adapters for various protocols and the associated deployment executable for these. Note, in many cases it will be needed to provide context specific data processing and thus a context specific CCF Control Application must be generated, based on the context specific adapters.

Note, in the design of the component, it was foreseen to provide a separate simulator process. This design choice has been changed to providing simulation via a Communication Adapter. It is the intention to provide simulation based on this scheme until further.

Moreover, the design contains a proposal for a Super DCS Application. This has not yet been implemented and will only be assessed if specific requests for such a service will be submitted.

2.1 Architecture

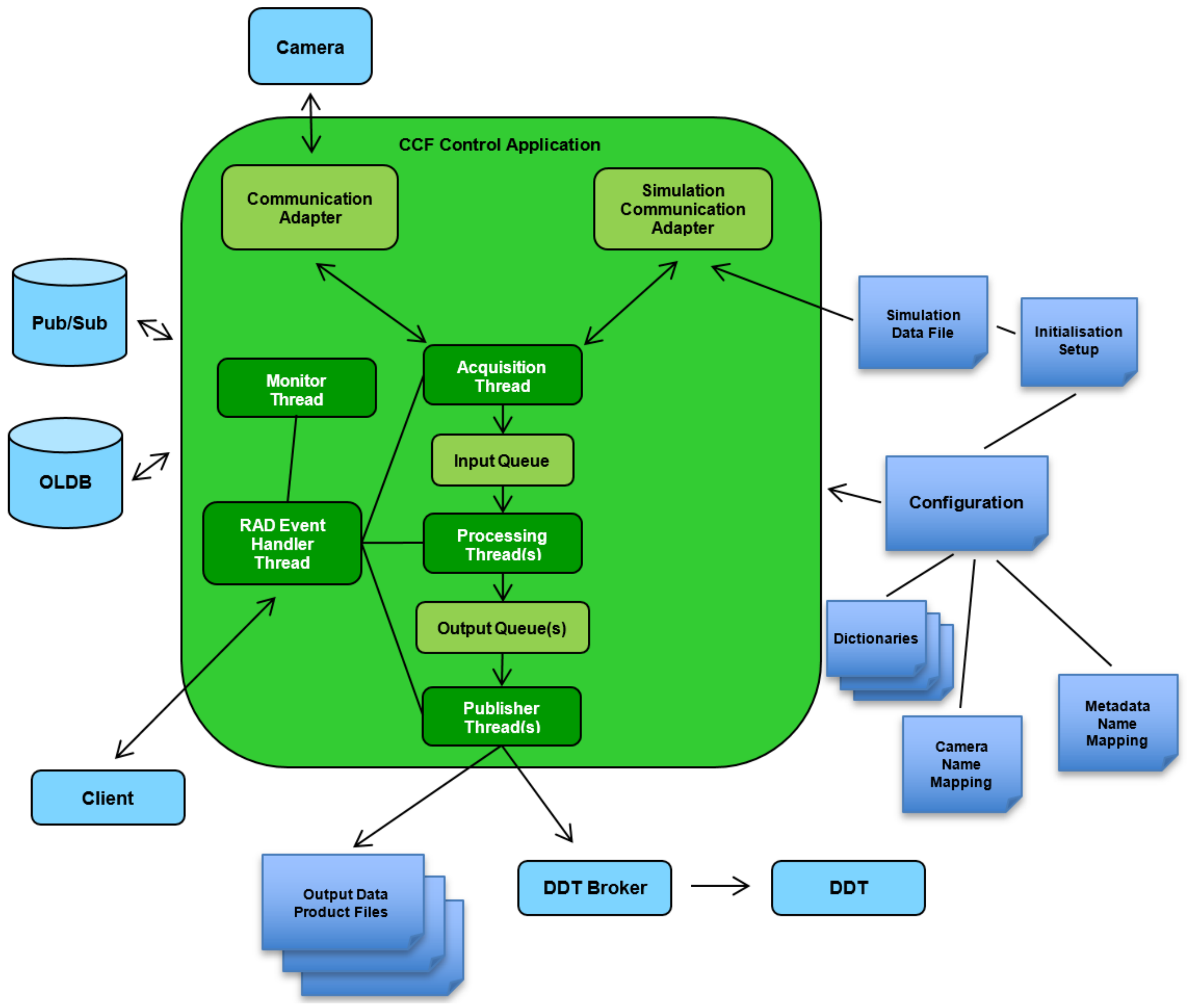
The aim of the CCF Package as such, is not to so much to provide a ready-to-use set of CCF application executables, but rather to facilitate easy integration of customised (context specific) DCS solutions. To achieve this, the CCF SDK is based on the adapter pattern to allow for context specific customization.

The adapters provided are:

- **Communication Adapter:** Implements protocol specific communication.
- **Processing Recipe Adapter:** Implements on-the-fly processing of image frames, executed in a dedicated thread in a CCF Control instance.
- **Data Publisher Adapter:** Implements publishing of data to specific targets, which may be output files or applications subcribing to the data.

For future versions a “Method Invocation Adapter” (for the “dcsif::Execute” request) is provided, and possibly a “Setup Handling Adapter” (for the “dcsif::Setup” request), according to needs.

The anatomy of a CCF Control Application is depicted below.



To use the CCF it is not required to have a very detailed/low-level insight into the architecture. Nonetheless, a basic understanding is needed, and useful, to be able to use it and to implement, deploy and test the various adapters needed.

The internal threads executed in a CCF Control Application are:

RAD Event Handler Thread: The main thread in the application, which handles e.g. incoming requests.

Monitor Thread: The purpose of the Monitor Thread is to supervise the operation of the CCF Control instance. In particular it collects periodically, data about the execution of the various other CCF threads, which is used to compute statistics, which is written in the OLDB.

Acquisition Thread: The Acquisition Thread, as indicated by the name, is responsible for acquiring the image frames from the associated camera. Once a frame has been received, the image data (pixels) and metadata information is stored in the Input Queue and the Processing Thread(s) informed about the availability of a new data frame. Note, at this point in time, it is only possible to receive complete frames and not to handle sub-frames.

Processing Thread(s): The Processing Threads applies processing in the form of one or more Processing Recipes (adapters) on the data. When informed about the availability of a new frame in the Input Queue, it reserves an image buffer in its associated Output Queue and applies the Recipe(s) specified on the image. Once the processing of an image frame has completed, the Processing Thread informs the associated Publisher Threads about the availability of a new, processed image. If no Recipes are specified (configured), the image frames are merely copied from the Input Queue buffer into the Output Queue buffer and the associated Publisher Thread(s) notified. The number of Processing Threads to execute, is configurable.

Publisher Thread(s): The Publisher Threads send the image data available in the associated Processing Thread Output Queue to the specified target(s). This will typically be files or e.g. the DDT. However, as the actual publishing is carried out by a Data Publisher Adapter, other types of targets are possible. In this version it is only possible to specify one Data Publisher Adapters per Publisher Thread.

When inspecting the threads running in a CCF Control instance, e.g. with “htop”, the user will observe more threads than the ones mentioned above. These are threads deployed by CII and e.g. by an SDK used for communicating with the camera.

The other internal elements composing a CCF Control Application are:

Communication Adapter: The Communication Adapter used for communicating with the associated camera. As this adapter may be provided by the user, it could be used for other purposes, or to collect data from other sources than cameras.

Simulation Communication Adapter: The Communication Adapter used for communicating with the simulator. For now, the provided Simulation Communication Adapter actually, is responsible for the simulation and is not communicating with an external simulator. If of relevance, a context specific Simulation Communication Adapter may be provided, either generating internally the simulated data or communicating with an external simulator.

Input Queue: Queue containing image buffer objects (“ccf::common::DataFrame”), with a preallocated set of buffers, used in a ring-buffer fashion by the Acquisition Thread to store the incoming frames, and make these available for the configured Processing Thread(s). The number of buffers in the Input Queue is configurable.

Output Queue: For each Processing Thread, one Output Queue is allocated to hand over processed data to the Publisher Threads. The elements in the Output Queue(s) are pre-allocated instances on the “ccf::common::DataFrame” class, which are re-used in a ring-buffer fashion. The number of image buffers in the Output Queue is configurable.

The external entities shown in the above figure, are as follows:

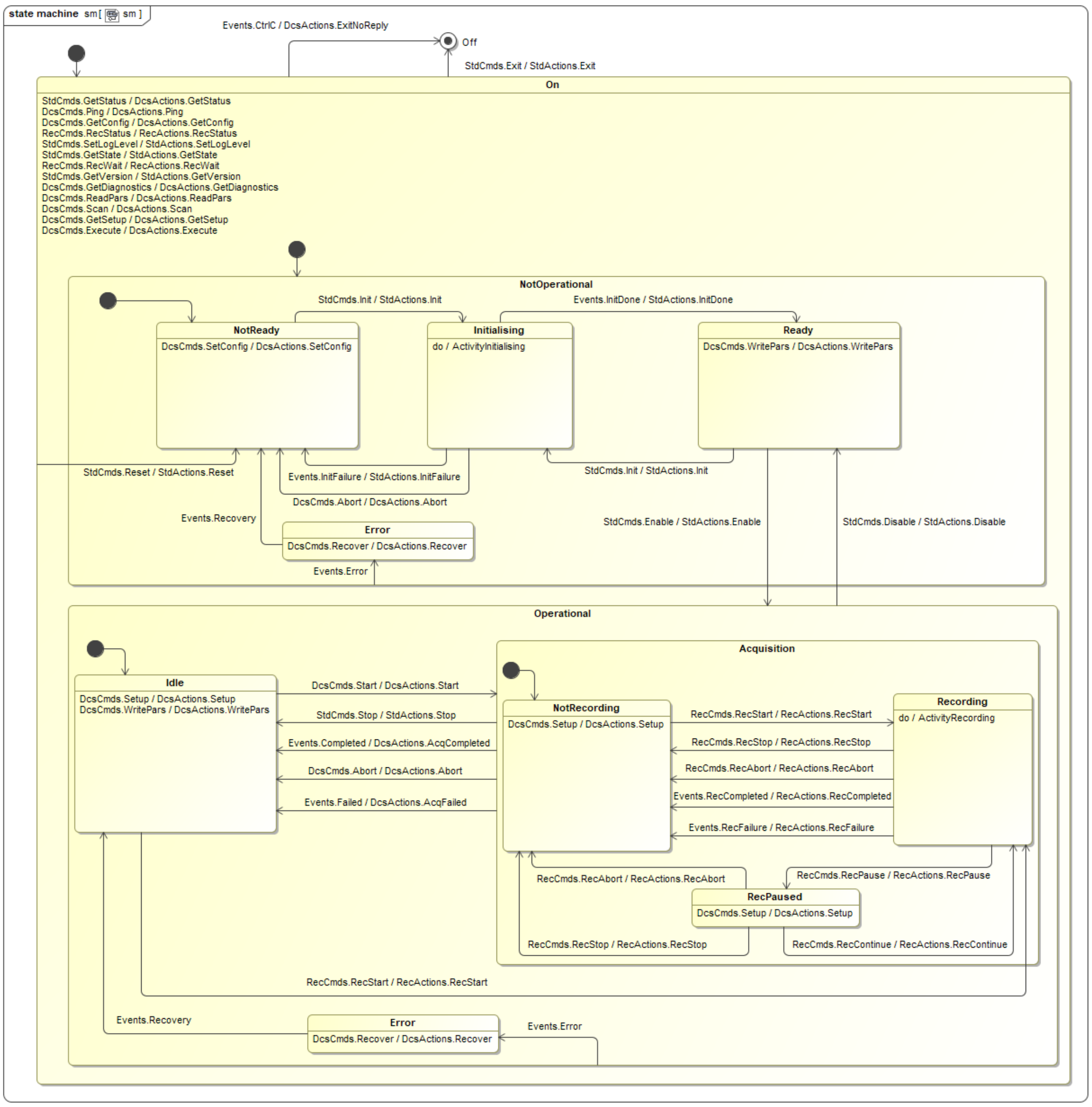
- **Pub/Sub:** The Pub/Sub service supported by CII MAL. For this version, this is only used for publishing the state of the application, in accordance with the Standard Application Interface (“stdif”[1]).

[1] https://gitlab.eso.org/ecs/ecs-interfaces/-/tree/master/std

- **OLDB:** The online database supported by the CII Services. The is is the base line for CCF to publish information for the client applications. An overview of the information updated in the OLDB, is provided in one of the following sections.
- **Client:** A client may here by e.g. the Sequencer, a GUI or a command line tool, commanding the CCF Control instance.
- **Configuration:** The main configuration file (YAML format) referenced when starting the application. The configuration is loaded and applied when the application is starting up, and applied to the associated camera during initialisation. The main configuration refers to a number of other files, which are loaded and applied as well, when the application starts up and initialises.
- **Initialisation Setup:** Contains the default setup when the application starts up (YAML format). This is referenced by the main configuration.
- **Camera Name Mapping:** Contains a mapping from the parameter names used internally in the Communication Adapter and the names defined in the camera name space.
- **Metadata Name Mapping:** Maps internal names for metadata, used in the CCF Control Application to the user metadata names (not used for this release).
- **Dictionaries:** Contains the definition of the keywords to be written in the output data products, normally FITS files.
- **Simulation Data File:** The Simulation Data File used as basis for the simulation. Typically this will be a FITS image cube, providing a sequence of images emulating the usage on sky. The user may provide use case specific Simulation FITS Files, to obtain a realistic emulation. An example Simulation Data File is provided with the CCF Package (“ccf/control/resource/image/ccf/control/testCube16Bit.fits”). If a context specific simulation is provided, e.g. in the form of a Communication Adapter, this adapter may define how to obtain the data used as input for the simulation. A small utility is provided (on a prototype basis) to generate Simulation Data File, FITS image cubes (“ccfGenFitsCube”).
- **Output Data Product Files:** Output files generated by a Data Publisher. Normally this will be FITS files, but if required, it will be possible to provide context specific Data Publishers to generate other formats, e.g. for more efficient storage in ‘burst mode’. A standard FITS Data Publisher is provided by the CCF Package.
- **DDT Broker/Viewer:** The standard DDT image transportation and display facility.

2.2 State Machine

CCF Control is using the standard state machine, defined for DCS applications by the DCS ICD (ESO-305615). The state machine is described in detail in this document, and will not be described further in this manual. A diagram representation of the state machine, is provided here for convenience:


state machine sm[sm]
Events.CtrlC / DcsActions.ExitNoReply
Off
StdCmds.Exit / StdActions.Exit
On
StdCmds.GetStatus / DcsActions.GetStatus
DcsCmds.Ping / DcsActions.Ping
DcsCmds.GetConfig / DcsActions.GetConfig
RecCmds.RecStatus / RecActions.RecStatus
StdCmds.SetLogLevel / StdActions.SetLogLevel
StdCmds.GetState / StdActions.GetState
RecCmds.RecWait / RecActions.RecWait
StdCmds.GetVersion / StdActions.GetVersion
DcsCmds.GetDiagnostics / DcsActions.GetDiagnostics
DcsCmds.ReadPars / DcsActions.ReadPars
DcsCmds.Scan / DcsActions.Scan
DcsCmds.GetSetup / DcsActions.GetSetup
DcsCmds.Execute / DcsActions.Execute
NotOperational
StdCmds.Init / StdActions.Init
Events.InitDone / StdActions.InitDone
NotReady
DcsCmds.SetConfig / DcsActions.SetConfig
Initialising
do / ActivityInitialising
Ready
DcsCmds.WritePars / DcsActions.WritePars
StdCmds.Reset / StdActions.Reset
Events.InitFailure / StdActions.InitFailure
StdCmds.Init / StdActions.Init
DcsCmds.Abort / DcsActions.Abort
Events.Recovery
StdCmds.Enable / StdActions.Enable
StdCmds.Disable / StdActions.Disable
Error
DcsCmds.Recover / DcsActions.Recover
Events.Error
Operational
Acquisition
Idle
DcsCmds.Setup / DcsActions.Setup
DcsCmds.WritePars / DcsActions.WritePars
DcsCmds.Start / DcsActions.Start
StdCmds.Stop / StdActions.Stop
Events.Completed / DcsActions.AcqCompleted
DcsCmds.Abort / DcsActions.Abort
Events.Failed / DcsActions.AcqFailed
NotRecording
DcsCmds.Setup / DcsActions.Setup
Recording
do / ActivityRecording
RecCmds.RecStart / RecActions.RecStart
RecCmds.RecStop / RecActions.RecStop
RecCmds.RecAbort / RecActions.RecAbort
Events.RecCompleted / RecActions.RecCompleted
Events.RecFailure / RecActions.RecFailure
RecCmds.RecAbort / RecActions.RecAbort
RecCmds.RecPause / RecActions.RecPause
RecPaused
DcsCmds.Setup / DcsActions.Setup
RecCmds.RecStop / RecActions.RecStop
RecCmds.RecContinue / RecActions.RecContinue
RecCmds.RecStart / RecActions.RecStart
Events.Recovery
Error
DcsCmds.Recover / DcsActions.Recover
Events.Error

A local copy of the DCS state machine SCXML document can be found in the CCF Package ("ccf/control/resource/config/ccf/control/sm.xml")

2.3 Online Database

The CCF Control Application updates a set of information in the OLDB to be used by client applications, e.g. for displaying the status in control GUIs, or in Sequencer Scripts.

Each CCF Control Application instance, has one configurable root (name), which is used for all the keys, written in the OLDB. It is recommened that this DB root name clearly identifies the CCF Control instance and a format of this form is recommended: “<system><instance ID>”, e.g. “MetisExpoMeter”.

In the following, the keys written in the OLDB are documented, also because it is (probably) mostly self-explanatory. Some keys are described elsewhere in this document.

The main categories of information updated in the OLDB are:

camera	Contains status values for the camera properties. This may e.g. included the property names defined in the camera name space.
config	The current configuration loaded/used by CCF Control
dictionary	The dictionary keys in use.
publishers	Status of each Data Publisher deployed in the CCF Control instance.
recording	The overall status of an ongoing Data Recording.
setup	The current set of setup parameters in use.
sm	The state machine status.
statistics	Run-time statistics about the execution (performance) of the various internal threads. This is described in more details elsewhere in this document.

2.4 Example & Template Files

In addition to the modules mentioned above, the CCF package contains a number of files, which will be installed in the INTROOT.

The files, which *may* be used in a production environment, are:

- **ccf/common/resource/config/ccf/client/log.properties:** Log properties to be applied for the C-Client.
- **ccf/control/resource/config/ccf/control/log.properties:** Log properties for the CCF Control application.
- **ccf/control/resource/config/ccf/control/sm.xml:** Standard DCS SCXML statemachine.
- **ccf/control/resource/config/ccf/control/metaDataMapping.yaml:** Example Meta Data Mapping - not supported in this release.
- **ccf/control/resource/dictionary/ccf/control/ccf.did:** Base dictionary for CCF.
- **ccf/control/resource/image/ccf/control/Test_naxis3_1_512_int16.fits:** Example FITS cube used for demo purposes.
- **ccf/protocols/aravis/lib/resource/config/ccf/protocols/aravis/avtMapping.yaml:** Example Camera Name Mapping file for Allied Vision GigE Vision cameras.

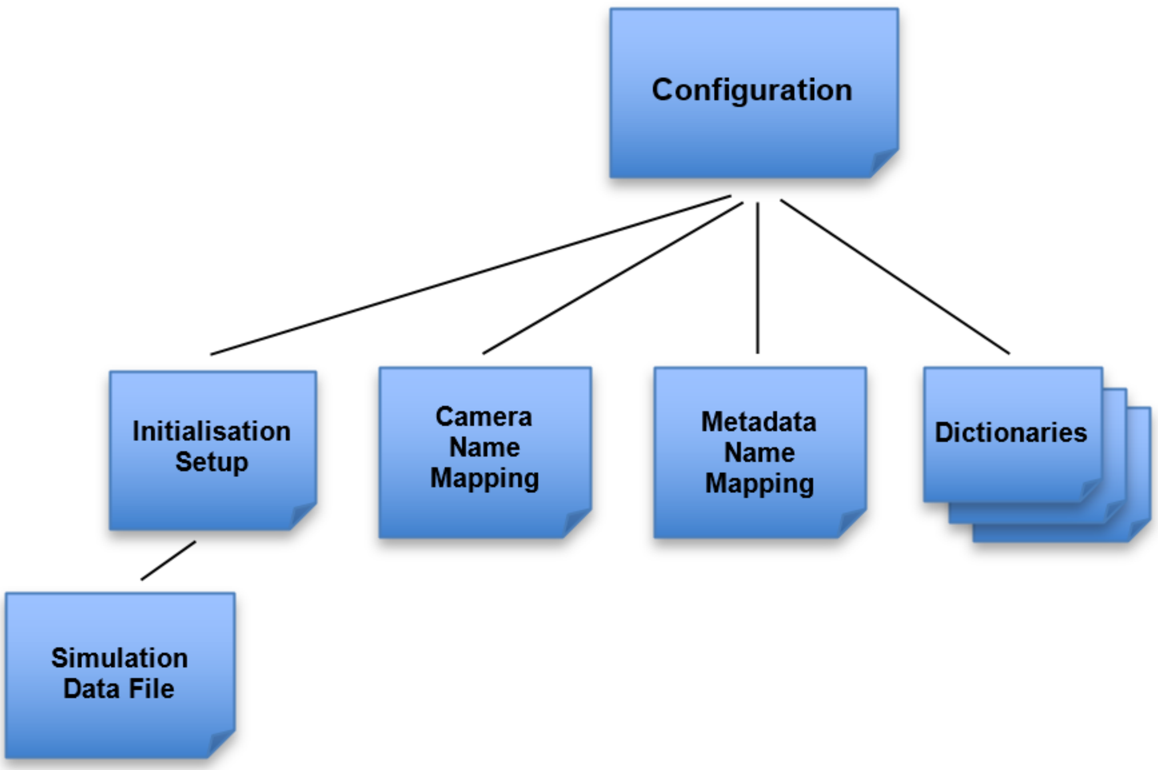
The files, which are provided only for example/demo purposes (**not for production or test deployment**), are:

- **ccf/control/resource/config/ccf/control/config1.yaml:** Example configuration without support for DDT.
- **ccf/control/resource/config/ccf/control/initSetup1.yaml:** Initialisation Setup referenced by “ccf/control/config1.yaml”.
- **ccf/control/resource/config/ccf/control/configDdt1.yaml:** Example configuration with support for DDT.
- **ccf/control/resource/config/ccf/control/initSetupDdt1.yaml:** Initialisation Setup referenced by “ccf/control/configDdt1.yaml”.
- **ccf/control/resource/config/ccf/control/minimalConfig1.yaml:** Example minimal Configuration.
- **ccf/control/resource/config/ccf/control/minimalInitSetup1.yaml:** Example minimal Initialisation Setup.

3 Configuration

In this chapter, the configuration of a CCF Control instance is described. The configuration is loaded and installed during start-up and initialisation of the application. The configuration (set of parameters) may be extended, by adding new adapters, requiring additional configuration information.

The configuration files needed for one instance of CCF Control, are shown in the following diagram:



The logging carried out by CCF is based on the “log4cplus” tool and consequently, also a “logging properties” file is installed and used by the application.

3.1 Main Configuration

This version of the configuration is based on YAML and a plain hierachical parameter naming scheme.

The following categories of parameters are defined:

- System
- Monitoring
- Camera
- Acquisition

- Processing Pipeline

An example of a configuration, is available in CCF: “ccf/control/resource/config/ccf/control/config1.yaml”.

3.1.1 System Configuration Parameters

In the following the table the system related configuration parameters are listed:

sys.name (String)	Name allocated to the system. It is recommended that the name, uniquely identifies the instance system wide. A name of the form “<system><instance>”, is proposed, whereby “<system>” could be the name of an instrument and “<instance>”, the instance name within that context.
sys.req.endpoint (String)	Endpoint URI for the CII MAL Request/Reponse service. E.g., “zpb.rr://127.0.0.1:12092/”.
sys.db.endpoint (String)	Endpoint URI for the CII OLDB service. E.g., “127.0.0.1:7878”.
sys.db.prefix (String)	Prefix for all OLDB keys handled by this instance of CCF. May be allocated like “sys.name”.
sys.db.timeout.sec (Integer)	Timeout in seconds to apply when access the OLDB.
sys.sm.scxml (String)	Name of the state machine SCXML definition. Normally the default one shall be used but if a new application class is implemented, derived from the CCF class, it is possible to specify an alternative state machine for the internal state machine engine.
sys.pub_sub.endpoint (String)	Endpoint URI for the CII Pub/Sub service, used for publishing status information. Example: “zpb.ps://127.0.0.1:52234/”.
sys.log.properties (String)	Name of the log properties file for the “log4cplus” logging service. Example: “ccf/control/log.properties”, which is the default logging properties configuration delivered with CCF. It is possible to provide an alternative log properties file for the individual CCF instance, if of relevance.
sys.log.level (String)	Default log level to apply. May be overwritten with the “-l <level>” command line option, given to the application. Possible values: “ERROR”, “WARNING”, “STATE”, “EVENT”, “ACTION”, “INFO”, “DEBUG”, “DEBUG2”, “DEBUG3”, “TRACE”.
sys.init.setup (String)	Name of the Initialisation Setup to load during application start-up and initialisation. It provides default values for the setup parameters. A specific Initialisation Setup shall normally be provided for a specific instance. Example: “ccf/control/initSetup.yaml”.
sys.image.dir (String)	Location where image output files will be stored. The complate path for the storage location is: “$DATAROOT/<sys.image.dir>”, which consequently must exist.

Continued on next page

Table 3.1 – continued from previous page

sys.did# (String)	Names of dictionaries to load for the generation of the metadata in the output data products. “sys.did1 “ccf/control/ccf.did”. Note, “ccf/control/ccf.did” shall normally be loaded. New keys shall be provided via instance specified dictionaries.
sys.metadata.mapping (String)	Name mapping mapping internal names for parameters into the metadata names, typically FITS keywords to write in Data Product headers. Not supported for this version.
sys.simulation (String)	Defines whether to run the instance in normal or simulation mode. From the perspective of CCF, switching between normal and simulation mode, boils down to which Communication Adapter to use.
sys.req.hist.size (Integer)	Number of requests to keep in the internal history. For now, it’s only used for caching Recording Session status info for the “recif::RecStatus” request.
sys.req.hist.expiration (Integer)	Expiration time, in seconds, to apply for requests cached in the internal history.

3.1.2 Monitoring Configuration Parameters

In the following the table the monitoring related configuration parameters are listed:

mon.period (Integer)	Period in seconds for scheduling the Monitor Thread. Recommended value is 1s. Note, this determines how frequent the thread execution statistics is computed and updated in the OLDB.
mon.nb_of_samples	Number of samples to use for the sliding window of samples, used for calculating the thread execution statistics.

3.1.3 Camera Configuration Parameters

In the following the table the camera related configuration parameters are listed:

cam.name (String)	Name assigned to the camera.
cam.id (String)	ESO wide, unique ID assigned to the camera.
cam.model (String)	Camera model.
cam.chip.model (String)	Chip model.
cam.type (String)	Type of camera, e.g. “CCD”, “IR, “CMOS”.
cam.manufacturer (String)	Camera manufacturer.
cam.adapter.protocol (String)	Protocol used for communicating with the camera, e.g. “GigE Vision”.
cam.adapter.mapping (String)	Name mapping, mapping the internal names, used in the communication adapter, into the specific names, deined in the camera name space.

Continued on next page

Table 3.2 – continued from previous page

cam.adapter.api (String)	API (or SDK) used for implementation the communication with the camera. Example: "CCF/Aravis".
cam.adapter.address (String)	Address used for connecting to the camera. Is typically an IP address + port number, for Ethernet/socket based communication protocols.
cam.adapter.sim_address (String)	Address of the CCF simulator, if an external simulator is used. Note, for the present version of CCF, simuilation is provided via a Simulation Communication Adapter.
cam.adapter.property# (String)	Property for the Communication Adapter. Example: "cam.adapter.property1: StreamBytesPerSecondMax=124000000".
cam.chip.width (Integer)	Width in pixels, of detector chip.
cam.chip.height (Integer)	Height in pixels, of detector chip.
cam.chip.max_resolution (Integer)	The maximum possible resolution (bits/pixel) for the given camera.
cam.chip.resolution (Integer)	Resolution used in given set up.

Note, some of the camera category configuration parameters may be available via the camera interface.

3.1.4 Acquisition Configuration Parameters

In the following the table the acquisition configuration parameters are listed:

acq.priority (Integer)	Priority of the thread. Reserved for possible future deployment on RT Linux.
acq.max_rate (Double)	Maximum rate for the data acquisition (Hz).
acq.inputq.size (Integer)	Number of frame buffer elements in the Input Queue. In the optimal case, two buffers would be sufficient. However, due to the deployment on a non real-time platform, it is necessary to allocate extra buffers to average out jittering.
acq.allow_lost_frames (Boolean)	If true, frames lost at the level of interaction with the camera, are silently ignored.
acq.allow_frame_skipping (Boolean)	If true, in case there are no free buffers available in the Input Queue, these are siliently discarded. Otherwise a warning log is produced.

3.1.5 Processing Pipeline Configuration Parameters

In the following the table the Processing Pipeline configuration parameters are listed:

proc#.name (String)	Name of the Processing Thread with the given index assigned to it ("proc#": "#" defines the instance, e.g. "Processing Thread 1".
proc#.priority (Integer)	Priority of the thread. Reserved for possible future deployment on RT Linux.
proc#.outputq.size (Integer)	Number of frame buffer elements in the Output Queue allocated to the Processing Thread 1. In the optimal case, two buffers would be sufficient. However, due to the deployment on a non real-time platform, it is necessary to allocate extra buffers to average out jittering.
proc#.allow_frame_skipping (Boolean)	If true, in case there are no free buffers available in the Processing Thread's Output Queue, these are siliently discarded. Otherwise a warning log is produced.
proc#.recipe#.adapter (String)	Name of the Processing Recipe Adapter to be applied on the image frames, for processing purposes. The name must match the name allocated in the Processing Recipe Adapter implementation, e.g. "ccf::stdrecipe::RecipeCentroid".
proc#.recipe#.name (String)	Logical name allocated to the Processing Recipe instance. Serves mainly to get meaningly and easy understandable logging output from the recipe.
proc#.pub#.adapter (String)	Name of the Data Publisher Adapter used to publish the processed image frames, for processing purposes. The name must match the name allocated in the Processing Recipe Adapter implementation, e.g. "ccf::stdpub::PubFits". "proc#" refers to the Processing Thread instance. "pub#" refers to the Data Publisher Thread instance.
proc#.pub#.name (String)	Logical name allocated to the Data Publisher instance. Serves mainly to get meaningly and easy understandable logging output from the recipe.
proc#.pub#.priority (Integer)	Priority of the thread. Reserved for possible future deployment on RT Linux.

3.2 Initialisation Setup

An Initialisation Setup must be specified in the Main Configuration to define the default setup of the system after start-up of the application.

The Initialisation Setup merely lists the various possible Setup Parameters, defined for the given instance. The issue of the setup is handled in the next chapter.

An example of an Initialisation Setup, is provided within the CCF Package: "ccf/control/resource/config/ccf/control/initSetup1.yaml".

3.3 Camera Name Mapping

The Camera Name Mapping file is a file in YAML format, used to map the names for camera properties, used in the Communciation Adapter, into the specific names defined for the associated camera.

Using a Camera Name Mapping makes sense, if a generic Communication Adapter can be re-used to control multiple cameras, if it is avoided to hardcode names in the Communication Adapter.

If the names are hardcoded in the Communication Adapter, or not used in connection with the given camera/SDK, a Camera Name Mapping is not needed.

An example of a Camera Name Mapping is provided by the CCF Package: “ccf/protocols/aravis/lib/resource/config/ccf/protocols/aravis/avtMapping.yaml”.

3.4 Metadata Name Mapping

The purpose of the Metadata Name Mapping file is to map parameters registered in connection with a given image frame, into the actual metadata names, required in the Output Data Product File.

When a frame is received, the Communication Adapter may register metadata, describing the image, in its frame buffer object. In addition, Processing Recipes may add such metadata, which should be added in the Output Data Product Files. When a Metadata Name Mapping is provided, the target Output Data Product metadata keys, shall be registered in a Dictionary, defined in the configuration.

Note: The Metadata Name Mapping feature is not supported for this version.

3.5 Dictionaries

A Dictionary is used to describe the properties of the metadata keywords to be written in the Data Product Output Files. The format and tools to handle Dictionaies, is provided by the ICS Core Data Interface Tools (DIT) Package.

A base Dictionary is provided by the CCF Package: “ccf/control/resource/dictionary/ccf/control/ccf.did”. This Dictionary shall normally always be specified in the configuration and used.

3.6 Simulation Data File

Whether or not a Simulation Data File, typically a FITS cube, is used as basis for the simulation, depends on the implementation of the simulated behaviour in the given context.

In the present version, the simulation is implemented within the Simulation Communication Adapter and it is based on a FITS file as input. The FITS file is then played back, according to the given setup/exposure parameters.

If a context specific simulator is provided, it may use another type of file, or no input file at all.

A small tool is provided to allow the users of CCF to create their own FITS data cubes, with an emulated star moving in the field of view. The tool is named “ccfGenFitsCube”.

4 Setup

The system setup defines the set of system properties, which can be changed during run-time, with some exceptions.

This version of the setup is based on YAML and a plain hierachical parameter naming scheme.

The following categories of parameters are defined:

- Exposure Control
- Simulation
- Pipeline Processing

An example of a setup, in this case the Initialisation Setup, is available in CCF: “ccf/control/resource/config/ccf/control/initSetup1.yaml”.

Adding new adapters, it is possible to also add new Setup Parameters. The ones defines in this chapter are the standard/common parameters.

4.1 Exposure Control Parameters

In the following the table the exposure control related setup parameters are listed:

expo.mode (String)	Defines the mode to acquire the camera. Possible value are “Continuous” and “Finite”.
expo.nb (Integer)	Defines the number of image acquisitions to carry out in “Finite” Exposure Mode. Note: Not supported for the ICSv3 release.
expo.time (Double)	Exposure time in seconds to apply for the image acquisition.
expo.frame.rate (Double)	The requested frame rate to apply. Note the frame rate is correlated with the exposure time. Note: Not supported for the ICSv3 release.
expo.win.start_x (Integer)	Defines the lower left x-coordinate of the current ROI. The first pixel is 1.
expo.win.start_y (Integer)	Defines the lower left y-coordinate of the current ROI. The first pixel is 1.
expo.win.width (Integer)	The width of the current ROI in pixels.
expo.win.height (Integer)	The height of the current ROI in pixels.
expo.bin_x (Integer)	The binning in X.
expo.bin_y (Integer)	The binning in Y.

4.2 Simulation Parameters

In the following the table the exposure control related setup parameters are listed:

sim.type (String)	Defines the type of the simulation. Not relevant for the simulated behaviour provided by the Simulation Communication Adapter in this release as it supports play-back of an FITS Data Simulation File.
sim.file (String)	Name of Simulation Data File to use as basis for the simulated behaviour.

4.3 Pipeline Processing

In the following the table the exposure control related setup parameters are listed:

proc#.enabled (Boolean)	Used to toggle the given Processing Pipeline on/off.
proc#.recipe#.enabled (Boolean)	Used to enable/disable the individual Processing Recipe registered and defined.
proc#.recipe#. recipe_base.delay (Double)	Engineering parameter used to introduce an artificial delay, given in seconds, in the Base Recipe Class (“ccf::common::RecipeBase”).
proc#.pub#.enabled (Boolean)	Used to enable/disable the individual Data Publisher Thread.
proc#.pub#.pub_base.delay (Double)	Engineering parameter used to introduce an artificial delay, given in seconds, in the Base Recipe Class (“ccf::common::RecipeBase”).

Note, for each Data Publisher Adapter, specific setup Parameters may be defined. The parameters for the standard Data Publisher Adapters provided by the CCF Package, are documented in the chapter “Standard Adapters”.

5 Request & Pub/Sub Interfaces

The CCF implements the following CII MAL interfaces:

- **Standard Application Interface - "stdif":** Defines the set of requests and pub/sub topics that shall be implemented by all instrument applications.
- **DCS Application Interface - "dcsif":** Defines the set of requests that shall be implemented by DCS type of applications like CCF or NGC. No pub/sub topics are defined for this interface at this point in time.
- **Recorder Application Interface - "recif":** Defines the set of requests that shall be implemented by data recording type of applications like CCF or NGC. A data recording application, is typically an application sampling data from a source and storing this data in output Data Product Files. No pub/sub topics are defined for this interface at this point in time.

The common CII interfaces used by CCF are provided from within the ELT Development Environment. In Gitlab they are available at this URL: https://gitlab.eso.org/ecs/ecs-interfaces.

These interfaces are documented in detail in the user manuals provided for each interface separately. This information will not be repeated in the CCF user's manual, but some additional implementation detailed mentioned in this chapter instead.

In this release not all requests have been implemented. The requests not implemented are:

DCS Interface ("dcsif"):

- Abort
- Execute
- GetDiagnostics
- GetDpData
- GetSetup
- ReadPars
- Recover
- SetConfig
- WritePars
- GetConfig

Recording Interface ("recif"):

- RecAbort
- RecContinue
- RecPause

6 Adapters

The CCF Package is an SDK, providing a number of tools to build DCS solutions. The “ccf::control::Application” implements the actual CCF Control Application. This can be adapted to accomodate context specific needs, by means of the following adapters:

- **Communication Adapter:** Implements protocol specific communication. It is also used, for now, for implementing the simulated behaviour of CCF. New Communication Adapters shall be derived from the base class provided, “ccf:common::ComAdptBase”.
- **Processing Recipe Adapter:** Implements on-the-fly processing of image frames, executed in a dedicated thread in a CCF Control instance. Processing Recipes shall be derived from the base class provided, “ccf::common::RecipeBase”.
- **Data Publisher Adapter:** Implements publishing of data to specific targets, which may be output files or applications subcribing to the data. The Data Publishers are invoked after the on-the-fly processing. Data Publishers shall be derived from the provided base class, “ccf::common::PubBase”.

The Doxygen documentation available for the three adapter base classes, describe the purpose and function of each method in the classes. In general, the virtual methods of the form “virtual <name>User(. . .)”, i.e., tagged with “User” are expected supposed to be implemented in the derived, specific adapter classes.

The specific set of adapters to be used in the target CCF application, shall be specified in the “main()” function, whereby factory classes shall be registered in the CCF Application object, before running the application. This is described in the chapter “Installation & Deployment”.

For all Adapter types, it is important that the method “SetClassName(<adapter name>)” is invoked in the constructor to define the name of the Adapter. The name allocated to the Adapter is used when instantiating it from their factory class instances. The name of the Adapter to instantiate, initialise and install, is defined by the “<prefix>.adapter” parameter in the configuration.

In the following more details about each type of Adapter is provided.

6.1 The Communication Adapter

The implementation of a Communication Adapter may involve quite some work, depending on the complexity of the camera protocol/interface and the possible associated SDK.

A number of obligatory Configuration and Setup Parameters are defined for the Communication Adapter. It is possible to define new, Communication Adapter specific Configuration and Setup Parameters. When defining these, it is the obligation of the adapter to check the validity of its specific parameters and to retrieve these.

In the following, an overview of the methods of Communication Adapter base class is provided.

The user methods which probably shall be implemented when developing a new Communication Adapter, are marked with “>>. . . <<”.

Other *user* methods *may* be implemented according to the needs. It is not attempted to provide the detailed information about each method, as this can be found in the Doxygen documentation. The adapter methods have been split into the following categories:

- General Household Keeping.
- Connection Handling.
- Parameter Handling.
- Camera Performance & Properties.
- Acquisition & Frame Handling.

6.1.1 Basic Household Keeping

```
ccf::common::ConfigBase& Cfg();
```

Convenience method providing access to the Singleton instance of the configuration.

```
void Initialise(const bool re_init = false);
>>virtual void InitialiseUser();<<
```

The Initialise() method is called after the Communication Adapter has been created. The "InitialiseUser() method shall be implemented to provide protocol specific initialisation.

```
const std::string& GetId() const;
```

Unique ID assigned to the adapter (used for internal purposes).

```
void SetHostAdddress(const std::string& host_address);
const std::string& GetHostAddress();
```

"GetHostAddress()" returns the address of the host system. It is normally necessary to set this via "SetHostAddress()" as it is derived automatically.

```
void SetIsSimAdapter(const bool is_sim);
bool GetIsSimAdapter() const;
```

The "SetIsSimAdapter()" shall be invoked in the constructor of Communication Adapters to indicate for CCF if the adapter is a simulation adapter.

6.1.2 Parameter Handling

```
template <class TYPE> TYPE GetCameraProperty(const std::string& property);
bool HasCameraProperty(const std::string& property, std::string& value);
virtual bool HasCameraPropertyUser(const std::string& property, std::string&␣
↪value);
```

The methods can be used to retrieve any parameter from the camera. If called before the camera is connected, it will try to retrieve the information from the associated configuration. This is e.g. needed if the maximum frame size shall be computed before a connection to the camera has been established. “HasCameraPropertyUser()” normally, shall not be implemented.

```
template <class TYPE> bool HasAdapterProperty(const std::string& property, TYPE&␣
↪value);
```

Used to probe for and retrieve a specific adapter property, defined in the configuration using the key “cam.adapter.property#”.

```
void Read(const std::string& name, ctd::param::Parameter& par);
void Read(const std::vector<std::string>& names, std::vector
↪<ctd::param::Parameter>& pars);
std::string Read(const std::string& name);
template <class TYPE> TYPE Read(const std::string& name)
>>virtual void ReadUser(const std::string& name, ctd::param::Parameter& par);<<
virtual void
  ReadUser(const std::vector<std::string>& names, std::vector
↪<ctd::param::Parameter>& pars);
```

Used to read parameters from the associated camera. Requires the camera to be connected (see also “HasCameraProperty()”). Normally only the “virtual void ReadUser(const std::string& name, ctd::param::Parameter& par)” needs to be implemented.

```
void Write(const ctd::param::Parameter& par);
void Write(const std::vector<ctd::param::Parameter>& pars);
template <class TYPE> void Write(const std::string& name, const TYPE& value)
>>virtual void WriteUser(const ctd::param::Parameter& par);<<
virtual void WriteUser(const std::vector<ctd::param::Parameter>& pars);
```

Used to write parameters to associated the camera. Normally only the “virtual void WriteUser(const ctd::param::Parameter& par)” needs to be implemented.

```
void Scan(std::vector<ctd::param::Parameter>& pars);
>>virtual void ScanUser(std::vector<ctd::param::Parameter>& pars);<<
```

Used to scan the name space of the camera (retrieve all parameters defined in the interface).

```
void HandleParPreWrite(ctd::param::Parameter& par);
virtual void HandleParPreWriteUser(ctd::param::Parameter& par);
```

Can be implemented to carry out a protocol/camera specific conversion/computation of the parameter to be written to the associated camera.

```
void HandleParPostWrite(ctd::param::Parameter& par);
virtual void HandleParPostWriteUser(ctd::param::Parameter& par);
```

Can be implemented to carry out a protocol/camera specific handling after a parameter was written to the associated camera.

```
void HandleParPostRead(ctd::param::Parameter& par);
virtual void HandleParPostReadUser(ctd::param::Parameter& par);
```

Can be implemented to carry out a protocol/camera specific conversion/computation of a parameter read from the associated camera.

```
void MapParForDevice(ctd::param::Parameter& par) const;
void MapParFromDevice(ctd::param::Parameter& par) const;
```

Map a parameter before writing it to the device or after having read it from the device. The mapping is done using the associated Camera Name Mapping. These methods are invoked automatically by CCF core and need normally not to be invoked by the user.

```
void HandleSetup();
>>virtual void HandleSetupUser();<<
```

Method invoked when a setup request has been received and needs to be applied to the camera.

6.1.3 Camera Performance & Properties

```
double GetTheoreticFrameRate();
virtual double GetTheoreticFrameRateUser();
```

Used to calculate the estimated frame rate in connection with handling a “recif::RecStart” request and other “recif” requests, to estimate the remaining time for executing the Recording Session.

```
uint32_t GetMaxFrameSize();
virtual uint32_t GetMaxFrameSizeUser();
```

Shall return the maximum frame size that can be produced by the given camera, in the given configuration. This is used to allocate various buffers to handle frames.

```
void CheckStatus(ccf::common::HwStatus& status, std::string& message);
>>virtual void CheckStatusUser(ccf::common::HwStatus& status, std::string&␣
↪message);<<
```

Invoked periodically by the Monitor Thread when the camera when no image acquisition is on-going.

6.1.4 Connection Handling

```
const std::string& GetCameraAddress() const;
void SetCameraAddress(const std::string& address);
>>virtual void SetCameraAddressUser(const std::string& address);<<
```

Used to get/set the address used to connect to the camera. The “SetCameraAddressUser()” shall be implemented to chgeck the validity of the specific connection URI.

```
void CheckConnection(bool& connected);
>>virtual void CheckConnectionUser(bool& connected);<<
```

Invoked to check that the connection to the camera seems to be properly established.

```
bool IsConnected() const;
virtual bool IsConnectedUser() const;
void SetIsConnected(const bool connected);
```

```
void Connect();
>>virtual void ConnectUser();<<
```

“ConnectUser()” must be provided to carry out the protocol/SDK specific handling of creating a connection to the associated camera.

```
void Disconnect();
>>virtual void DisconnectUser();<<
```

“ConnectUser()” must be provided to carry out the protocol/SDK specific handling of disconnecting from the associated camera, in a clean way.

6.1.5 Acquisition & Frame Handling

```
bool     GetAllowLostFrames() const;
uint64_t IncLostFramesCount();
uint64_t GetLostFramesCount() const;
void     ResetLostFramesCount();
```

Indicates if it has been configured that frames may be lost silently. The Communication Adapter may then choose to use this for deciding whether to notify about this event or not. “IncLostFramesCount()” shall be invoked when a frame is lost. “GetLostFramesCount()” indicates the number of frames lost since the last acquisition was started. “ResetLostFramesCount()” shall be invoked when starting a new image acquisition sequence.

```
void Start();
>>virtual void StartUser();<<
```

“StartUser()” shall be implemented to handle the protocol/SDK specific logic to start a new image acquisition sequence.

```
void Stop();
>>virtual void StopUser();<<
```

“StopUser()” shall be implemented to handle the protocol/SDK specific logic to stop a new image acquisition sequence.

```
void Receive(ccf::common::DataFrame& frame,
             bool& frame_received,
             const double timeout,
             bool& timed_out);
>>virtual void ReceiveUser(ccf::common::DataFrame& frame,
                           bool& frame_received,
                           const double timeout,
                           bool& timed_out);<<
```

“ReceiveUser()” shall be implemented to receive the next frame. If a timeout is specified the method may time out but shall not throw an exception. Instead the “timed_out” parameter shall indicate if a timeout condition was encountered. When a frame has been received, the “frame_received” shall be true to indicate this.

Note: This version of CCF only handles entire frames. It may be evaluated if support for handling of sub-frames shall be provided.

6.2 The Processing Recipe Adapter

The Processing Recipe Adapters are invoked on the image frames to reduce them or to extract features from these.

It is strongly recommended to use the image processing routines provided by the ICS ODP Package to simplify the Processing Recipes and to make maintenance easier. However, if needed, other image processing packages may be used. It must be assured however, that these are threadsafe if it is the intention to run multiple Processing Pipelines in parallel, possibly leading to race conditions.

When a Processing Recipe is invoked on a CCF frame object, the copy handed over to the Recipe is a ‘private’ copy of that Recipe and it is ‘free’ to do the change the data and other information in the frame object, to the extend it does not render this invalid.

It is possible to specify a series of Processing Recipes to be invoked on the image frames in the Processing Pipeline. The sequence and parameters of these, is defined in the configuration, for the given Processing Pipeline.

A number of obligatory Configuration and Setup Parameters are defined for the Processing Recipe Adapter.

It is possible to define new adapter specific Configuration and Setup Parameters. When defining these, it is the obligation of the adapter to check the validity of its specific parameters and to retrieve these. New Recipe specific keys added, shall have a name of the form “proc#.recipe#.<name>”. The data type and format of the value is implementation dependent.

6.2.1 Basic House Keeping

```
RecipeBase(const uint16_t proc_thread_nb,
           const uint16_t recipe_nb,
           const std::string& recipe_name);
virtual ~RecipeBase();
```

The Processing Thread Recipe Numbers are derived from the Configuration parameter (indeces).

```
static const std::string& GenId(const uint16_t proc_thread_nb,
                                const uint16_t recipe_nb);
```

Generate the unique (within CCF application) ID for the given Processing Recipe. The ID is of the form: “Proc#Recipe#”, where the index refers to the number of the Processing Thread and the number of the Recipe, respectively.

```
void SetStatus(const ProcStatus status);
ProcStatus GetStatus() const;
```

Set/get the status of the Recipe: “ccf::common:ProcStatus”: UNDEFINED, IDLE, PROCESSING, FINISHED, FAILED.

```
const std::string& GetRecipeId() const;
const std::string& GetRecipeName() const;
uint16_t GetProcThreadNb() const;
uint16_t GetRecipeNb() const;
```

The Recipe ID corresponds to what is generated by GenId(). The Recipe Name is the name allocated via the configuration (key: “proc#.recipe#.name”). The Processing Thread Number refers to the index allocated to the given Processing Thread according to the configuration. The same applies to the Recipe Number.

6.2.2 Processing Recipe Adapters Instance Management

```
>>virtual void CreateObjectUser(const uint16_t proc_thread_nb,
                                const uint16_t recipe_nb,
                                const std::string& recipe_name,
                                RecipeBase** new_object);<<

static void AddRecipeFactoryObj(RecipeBase& recipe_factory_obj);

static void CreateRecipeObj(const std::string& class_name,
                            const uint16_t proc_thread_nb,
```

(continues on next page)

(continued from previous page)

```
                                       const uint16_t recipe_nb,
                                       const std::string recipe_name,
                                       RecipeBase** new_obj);

static void GetRecipeObj(const uint16_t proc_thread_nb,
                         const uint16_t recipe_nb,
                         RecipeBase** recipe_obj,
                         const bool initialise = true);

static void GetRecipeObjs(std::vector<RecipeBase*>& recipe_objs);

static bool HasRecipeObj(const uint16_t proc_thread_nb,
                         const uint16_t recipe_nb,
                         RecipeBase** recipe_obj);

static void FreeRecipeObjs();
static void FreeRecipeFactoryObjs();
```

The “CreateObjectUser()” method *must* be implemented to create the type specific Processing Recipe object, which is subsequently used by CCF when applying the procesing to the frames handled. The remaining methods are usually not needed by the specific Communication Adapter implementation

6.2.3 Initialisation & Enabling/Disabling

```
void Initialise();
virtual void InitialiseUser();
bool GetInitialised() const;
```

Initialise the Recipe instance. Reciupe specific Configuration parameters shall be parsed/handled during the initialisation.

```
void Enable();
virtual void EnableUser();
void Disable();
virtual void DisableUser();
bool GetEnabled() const;
```

Methods to enable/disable the Recipe. Only enabled Recipes are invoked on the data frames. It is possible to provide context specific handling in connection with the enabling/disabling, but usually this will not be necessary.

6.2.4 Data Processing

```
void Process(DataFrame& frame);
>>virtual void ProcessUser(DataFrame& frame);<<
```

Invoked by the Processing Thread to execute the processing on the image frames. The data contained in the frame object 'belongs' to the Processing Thread and may be modified by the Recipe, without creating a new copy.

6.3 The Data Publisher Adapter

Data Publishers are classified in two main categories:

- **Recording Data Publisher:** Recording Data Publishers, are publisher recording data into Output Data Products. This normally is triggered on demand from client software or user, by submitting a "recif::RecStart" request.
- **Non-Recording Data Publisher:** Non-Recording Data Publisher, are publishers which inject the data into other application software. This type of Data Publisher typically continuesly publishes data, but may implement internal rules for how and when to publish data. I.e., a publisher may implement a rule to only publish data at a certain frequency, and to discard the rest of the image frames without further actions.

Whether a Data Publisher is Recording or Non-Recording is defined in the constructor of the Data Publisher.

It shall also be defined whether a Data Publisher is activated by default in the constructor. Recording Publishers are typically disabled by default and enabled on demand, by means of the "recif::RecStart" request. The Non-Recording Data Publishers may continuesly activated.

The three properties described above, are exemplified in the FITS and DDT Data Publishers provided as Standard Data Publishers:

```
PubFits::PubFits(const uint16_t proc_thread_nb,
                 const uint16_t pub_thread_nb,
                 const std::string& pub_name) :
    PubBase(proc_thread_nb, pub_thread_nb, pub_name) {
  CCFTRACE;
  SetClassName("ccf::stdpub::PubFits");
  m_publisher_type = ccf::common::PubType::RECORDING;
  ...
  SetActivated(false);
}
```

```
PubDdt::PubDdt(const uint16_t proc_thread_nb,
               const uint16_t pub_thread_nb,
               const std::string& pub_name) :
    PubBase(proc_thread_nb, pub_thread_nb, pub_name) {
```

(continues on next page)

(continued from previous page)

```
  CCFTRACE;
  SetClassName("ccf::stdpub::PubDdt");
  m_publisher_type = ccf::common::PubType::NOT_RECORDING;
  ...
  SetActivated(true);
}
```

A number of obligatory Configuration and Setup Parameters are defined for the Data Publisher Adapter. It is possible to define new adapter specific Configuration and Setup Parameters. When defining these, it is the obligation of the adapter to check the validity of its specific parameters and to retrieve these. New Data Publisher specific keys added, shall have a name of the form “proc#.pub#.<name>”. The data type and format of the value is implementation dependent.

The “ccf::common::PublisherStatus” is used to keep track of the publishing, which is used by CCF core for various statistics and status purposes.

6.3.1 Basic House Keeping

```
PubBase(const uint16_t proc_thread_nb,
        const uint16_t pub_thread_nb,
        const std::string& pub_name);
virtual ~PubBase();
```

The Processing and Publisher Thread numbers are derived from the Configuration parameter (indeces).

Note, in this release it is only possible to add one Data Publisher per Publisher Thread. For the best performance, it is probably better to have only one Publisher per Publisher Thread, at least when Publishing into Output Data Products.

```
static const std::string& GenId(const uint16_t proc_thread_nb,
                                const uint16_t pub_thread_nb);
```

Generate the unique (within CCF application) ID for the given Processing Recipe. The ID is of the form: “Proc#Pub#”, where the index refers to the number of the Processing and Publisher Threads.

```
void Dismantle();
virtual void DismantleUser();
```

Invoked when terminating the application to clean up memory etc., which might have been allocated by the Data Publisher during execution.

```
const uint8_t GetProcThreadNb() const;
const uint8_t GetPubThreadNb() const;
const std::string& GetPubId() const;
const std::string& GetPubName() const;
```

The Publisher ID corresponds to what is generated by GenId(). The Data Publisher Name is the name allocated via the configuration (key: "proc#.pub#.name"). The Processing Thread Number refers to the index allocated to the given Processing and Publisher Threads according to the configuration.

```
PubType GetPublisherType() const;
```

Type of Data Publisher: Recording or not Recording, where Recording refers to that Output Data Product files are generated.

```
PublisherStatus& GetPubStatus();
void GetPubStatus(PublisherStatus& pub_stat) const;
```

Get the Publisher status information, with information about the execution of the publishing. The method "GetPubStatus(PublisherStatus& pub_stat)" is the preferred way to get a snapshot of the progress, but it makes a copy of the internal "ccf::common::PublisherStatus" object.

6.3.2 Processing Recipe Adapters Instance Management

```
>>virtual void CreateObjectUser(const uint16_t proc_thread_nb,
                                const uint16_t pub_thread_nb,
                                const std::string& pub_name,
                                PubBase** new_object);<<
static void AddPubFactoryObj(PubBase& pub_factory_obj);
static void CreatePubObj(const std::string& class_name,
                         const uint16_t proc_thread_nb,
                         const uint16_t pub_thread_nb,
                         const std::string& pub_name,
                         PubBase** new_object);
static void GetPubObj(const uint16_t proc_thread_nb,
                      const uint16_t pub_thread_nb,
                      PubBase** pub_obj,
                      const bool initialise = true);
static void GetPubObjs(std::vector<PubBase*>& pub_objs);
static bool HasPubObj(const uint16_t proc_thread_nb,
                      const uint16_t pub_thread_nb,
                      PubBase** pub_obj);
static void FreePubObjs();
static void FreePubFactoryObjs();
```

The "CreateObjectUser()" method must be implemented to create the type specific Data Publisher object, which is subsequently used by CCF when applying the procesing to the frames handled. The remaining methods are usually not needed by (of interest to) the specific Data Publisher Adapter implementation

6.3.3 Initialisation & Enabling/Disabling

```
void Initialise()/virtual void InitialiseUser();
bool GetInitialised() const;

void SetEnabled(const bool enabled);
bool GetEnabled() const;
```

Initialisation is invoked when creating a Data Publisher instance.

Whether or not to apply a Data Publisher on a given data frame, is defined by the ‘enabled state’.

6.3.4 Activation/Deactivation

```
void Activate()
virtual void ActivateUser()
void Deactivate()
virtual void DeactivateUser()
void SetActivated(const bool activated)
bool GetActivated() const
virtual void CheckForDeactivationUser();
```

The ‘activation flag’ indicates if the Publisher actually is publishing data, if it is enabled. This is needed for Recording Publishers as they are activated, e.g. when a “recif::RecStart” request is received, and remain activated until the specified set of Output Data Products have been generated.

6.3.5 Data Publishing

```
void Publish(DataFrame& frame,
             const bool force);
virtual void PublishUser(DataFrame& frame);
```

The “PublishUser()” method is the actual method to publish the data. The method should not attempt to release the Data Frame Object in the Output Queue when the publishing is completed; this is done automatically by CCF core.

6.3.6 Status Handling

```
void EstimateExecution(double& duration,
                       int64_t& nb_of_frames);
virtual void EstimateExecutionUser(double& duration,
                                   int64_t& nb_of_frames);
```

The method is used to provide an estimate for the recording status computation, for how long time the Recording Session is expected to execute.

```
void ResetDbStatus() const;
void UpdateDbStatus() const;
```

Resets the status key values in the DB (OLDB), and updates these during executation of the Data Publisher. By default, the information in the “ccf::common::PublisherStatus” is written to the OLDB at every cycle of the publishing.

7 Standard Adapters

The CCF shall be considered as an SDK, used to develop DCS solutions. It is therefore considered that for specific CCF instances, it will be necessary to develop one or more adapters of various types. This is typically Communication Adapters, to support a specific camera properties.

However, some adapters considered of common interest, will be provided by the CCF Package. It may be possible to use these, or to use them as base classes for new adapters, adjusted to the specific use cases.

In general it is recommended to attempt to use camera models already in use and to attempt to re-use the Standard Adapter provided by the CCF Package.

If developers identify adapters that might be of general interest, they are encourage to design/develop these in a way that they can be re-used and to propose these for inclusion in the CCF Package.

The Standard Adapters of the three categories 1) Communication Adapters, 2) Processing Recipe Adapters, and 3) Data Publisher Adapters, are described in the following sections.

7.1 Standard Communication Adapters

In this release, the following Standard Communication Adapters are provided:

- **Aravis SDK Communication Adapter - "ccf::protocols::aravis::ComAdptAravis":** This is based on the Aravis SDK[2], and implements the GigE Vision[3] protocol.
- **Simulation Communication Adapter - "ccf::control::ComAdptSim":** This provides in-application simulation. For now only play-back of an existing FITS file, typically a FITS cube, is provided.

These adapters are described in more detail in the following sections.

Note: At this point in time only support for GigE Vision is provided by CCF. For the moment, users of CCF will have to implement a customised solution for other protocols.

7.1.1 Aravis (GigE Vision) Communication Adpater

This adapter is based on the GenICam[4] standard and provides support for the GigE Vision communication protocol. Eventhough Aravis is a GenICam implementation, the CCF Aravis Communication Adapter only supports the GigE Vision protocol. Moreover, the present implementation is not a 'true' generic GigE Vision adapter; there are a few camera specific features implemented in the adapter and in general, there will usually always be differences between the various cameras to deal with

The Camera Name Mapping feature provided by CCF, may be capable of coping with at least simple differences in the parameter namespaces. It is being investigated how to make the present CCF

[2] https://github.com/AravisProject/aravis
[3] https://en.wikipedia.org/wiki/GigE_Vision
[4] https://www.emva.org/standards-technology/genicam/

Aravis Adapter more generic, e.g. to make it possible to provide support for the Pleora iPort[5] device (work on-going). This requires a re-engineering of the present adapter. Once this work is successfully completed an update to CCF and this manual will be provided. Providing support for the Pleora iPort device, would facilitate interfacing with CameraLink and possibly USB cameras.

Note: The whole issue in the context of the communication with COTS cameras is under-going evaluation, with the aim of establishing a better and simpler way of implementing the communcation with system specific cameras. It goes without saying that it is not a straightforward task to provide common tools to cover all possible physical interfaces, communication protocols and specific camera SDKs. This raises some concerns in terms of the long-term prospects for ESO, having to handle a plethora of different cameras, interfaces and SDKs. It is the hope that more information and maybe more viable solutions in this context, will be available in connection with the release of ICSv4 (TBD). Users of CCF shall therefore be prepared to replace a possible communication solution, put in place, with the new (more generic) solution, provided by ESO.

To use the CCF Aravis Communication Adapter to communicate with a camera with a GigE Vision interface, a first quick test may be carried out:

- Connect and configure the camera according to the camera user's guide.
- Launch the example Aravis Communication Adapter executable, "ccfCtrlAravis". The IP address in the configuration shall be set to the allocated IP address of the camera (refer to "Installation & Deployment" for information about how to execute a CCF Control instance). The "sys.sim" parameter in the configuration should be set to "false".
- Bring CCF Control to "Operational::Idle" by submitting an "stdif::Init" and "stdif::Enable" request.
- If this was successful, start the image acquisition by issuing a "dcsif::Start" request.
- If executing these commands are successfully executed, it could be attempted to display frames in the DDT Viewer (see "Installation & Deployment").

If issues are encountered, which seem related to the invalid names, it may be attempted to provide a camera specific Name Mapping; an example can be found within the CCF Package: "protocols/aravis/lib/resource/config/ccf/protocols/aravis/avtMapping.yaml".

If this does not work, it will probably be necessary to implement a dedicated Communication Adapter for the camera in question. Minor modifications of general interest, may be accepted for the CCF Aravis Standard Communication Adapter.

[5] https://www.pleora.com/products/frame-grabbers/iport-cl-ten/

7.1.2 Simulation Communication Adpater

This adapter provides in-application simulation. This means that the Acquisition Thread, executing the Communication Adapter “Receive” method, may have a higher CPU load compared to if the simulation is implemented in an external process.

The simulation provided in this release, is based on playing back the image frame contained in a FITS image cube.

The simulation respects the exposure time, but frame rate and windows parameters are not taking into account in this release. This will be added in a future release as well as other types of simulation, based on generating articificial data. Latter means it is not necessary to provide a FITS file to be used as input for the simulation, but predictable simlated images can be generated for test purposes.

A small tool, “ccfGenFitsCube”, is provided to generate a simple FITS cube to be used for the simulator.

7.2 Standard Processing Recipe Adapters

For this release only one Standard Processing Recipe is provided:

- Centroiding Recipe.

The provided Standard Recipes are based on the ICS ODP Component, which again is based on CPL.

7.2.1 Centroiding Recipe

The Centroiding Recipe finds the coordinates of an object, located in the center of the given images, within a certain window. To actually find the centroid, it uses internally the function “clipm_centroiding_gauss()”.

The Configuration Parameters are as follows (example):

```
proc1.recipe1.adapter:        "ccf::stdrecipe::RecipeCentroid"
proc1.recipe1.name:           "TestCentroidRecipe"
```

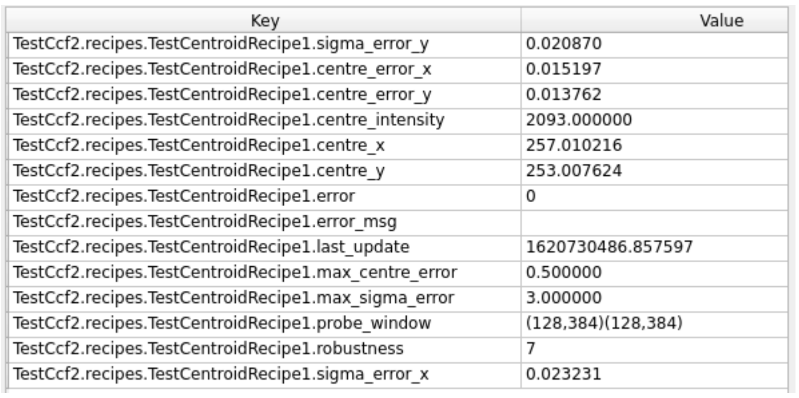
The Setup Parameters are as follows (example):

```
proc1.recipe1.enabled:            true
proc1.recipe1.max_centre_error:   0.5
proc1.recipe1.max_sigma_error:    3
proc1.recipe1.robustness:         7
```

proc#.recipe#.enabled (Boolean)	Enables/disables the recipe
proc#.recipe#.max_centre_error (Double)	Maximum error from the center to consider the search for the cemtroid as successful.
proc#.recipe#.max_sigma_error (Double)	Maximum sigma error of the fitting the Gaussian.
proc#.recipe#.robustness (Integer)	Number of iteration for obtaining the centroid accurately.

Note, the size of the central window in which the location of the centroiding is executed, is computed as (lower left, upper right) corners of window: “(((width / 4), (width / 4) + (width / 2)), (((height / 4), (height / 4) + (height / 2)))”.

While running, the Centroiding Recipe updates the results in the OLDB, e.g.:



Key	Value
TestCcf2.recipes.TestCentroidRecipe1.sigma_error_y	0.020870
TestCcf2.recipes.TestCentroidRecipe1.centre_error_x	0.015197
TestCcf2.recipes.TestCentroidRecipe1.centre_error_y	0.013762
TestCcf2.recipes.TestCentroidRecipe1.centre_intensity	2093.000000
TestCcf2.recipes.TestCentroidRecipe1.centre_x	257.010216
TestCcf2.recipes.TestCentroidRecipe1.centre_y	253.007624
TestCcf2.recipes.TestCentroidRecipe1.error	0
TestCcf2.recipes.TestCentroidRecipe1.error_msg	
TestCcf2.recipes.TestCentroidRecipe1.last_update	1620730486.857597
TestCcf2.recipes.TestCentroidRecipe1.max_centre_error	0.500000
TestCcf2.recipes.TestCentroidRecipe1.max_sigma_error	3.000000
TestCcf2.recipes.TestCentroidRecipe1.probe_window	(128,384)(128,384)
TestCcf2.recipes.TestCentroidRecipe1.robustness	7
TestCcf2.recipes.TestCentroidRecipe1.sigma_error_x	0.023231

The OLDB status keys are of the form: “<CCF instance name>.recipes.<recipe instance name>.<key>”. The keys are:

sigma_error_x/y (Double)	Output sigma error in X/Y.
centre_error_x/y (Double)	Output centre error X/Y.
centre_intensity (Double)	Intensity (value) of centre pixel.
centre_x/y (Double)	X/Y coordinate of centre pixel.
error (Boolean)	If true, indicates that the centroid searching failed.
error_msg (String)	Error message (if available), in case an error occurred.
max_centre_error (Double)	Maximum centre error specified in the configuration. If the error is above, this will lead to failure of recipe.
max_sigma_error (Double)	Maximum sigma error specified in the configuration. If the error is above, this will lead to failure of recipe.
last_update (Double)	Time since epoch of las update.
probe_window (String)	Status of probe window used ((x1, y1), (x2, y2)).
robustness (Integer)	Maximum number of retries.

7.3 Standard Data Publisher Adapters

In this release, two Standard Data Publisher Adapters are provided:

- FITS File Publisher.
- DDT Publisher.

These are described in the following.

7.3.1 FITS File Publisher

The FITS File Publisher provided for this release, is a basic implementation that only generates FITS files without extensions nor does it generate FITS cubes; this is planned for a later release.

The specific Setup Parameters provided for the adapter are:

proc#.pub#.basename (String)	Basename used for generating the output filenames. The names are of the form: "<proc#.pub#.basename>[_<ISO 8601>].fits", where the ISO is prepended to make the name unique, when "proc#.pub#.overwrite" is false.
proc#.pub#.max_size (Integer)	Indicates the maximum volume in MB of the output data generated during a Recording Session.
proc#.pub#.nb_of_frames (Integer)	Indicates the maximum number of image frames to be handled during a Recording Session.
proc#.pub#.format (String)	Format of the Output Data Product files. Valid options are: Single, Cube, MEF. Only "Single" is supported in this release.
proc#.pub#.overwrite (Boolean)	If set to "true", the same output filename is used, which is basically the specified "proc#.pub#.basename". This is used for test purposes to avoid generating a lot of files on disk.
proc#.pub#.rec_mode (String)	The recording mode. Valid options are: 1: All: All frames are recorded. 2: I:<#>: Every '#'th frame is taken. 3: P:<period>: A frame every <period> second is recorded. In this release, only "1: All" is supported.

Note, the FITS Data Publisher is not responsible for any 'cleaning up' such that all output files generated, must be removed by the client/user. When 'overwrite mode' is used, however, the same file is always written.

7.3.2 DDT Publisher

The DDT Publisher publishes data into the DDT data handling infrastructure to allow for distributed access and for displaying in the DDT image widgets.

When using the CCF DDT Publisher, it is important to ensure that the properly configured instance of the DDT Broker is running, otherwise CCF Control will not start.

The DDT system as such, is not documented here. Refer to the DDT user's manual for further information.

The specific Setup Parameters provided for the adapter are:

proc#.pub#.ddt.id (Integer)	Name of the DDT stream used for publishing the data.
proc#.pub#.ddt.broker (Integer)	The URI of the DDT Broker to use when publishing. E.g., "zpb.rr://127.0.0.1:11011/broker/Broker1".
proc#.pub#.ddt.max_rate (Double)	Can be used to limit the rate with which data is published in the DDT system, to avoid overloading the DDT and the execution host.

8 Installation & Deployment

In this chapter some information in connection with the installation and usage of the CCF are provided.

8.1 Installation

The CCF Package is a standard waf/wtools build project and shall be configured, built and installed accordingly.

The CCF Control is relying on the following environment variables:

- **DATAROOT:** Directory on the host machine in which Output Data Products will be generated. The storage location will be rendered as "$DATAROOT/<cfg key: sys.image.dir>". The 'image' sub-directory in "DATAROOT" will be created automatically by CCF Control, if not existing.
- **CFGPATH:** The "CFGPATH" environment variable, is a colon separated list of paths, pointing to possible Resource Directories in which resource data of different kinds are located.
- **INTROOT:** In this release, the example configurations delivered with the CCF Package as well as the libraries, header files and binaries, are installed into the location pointed to by "INTROOT". This may change in the future.

Note, in accordance with the "Instrument Software Specification", one system, typically an insrument, shall deliver a ready-to-use Resource Tree as part of the software package. This Resourece Tree shall be referenced to by the "CFGPATH" variable.

8.2 Deployment Module

Details about the configuration of a CCF Control instance are given in the "Configuration" chapter in this manual.

The CCF Package is a tool used to integrate specific DCS solutions. Normally, specific Adapters will be provided for the given context, and used to deploy (generate) a new CCF Control executable.

Two examples of deployment modules are provided by the CCF Package:

- ccf/sim
- ccf/protocols/aravis/exe

The example from "ccf/sim" is shown here:

```
#include <google/protobuf/stubs/common.h>

#include <ccf/stdpub/pubFits.hpp>
#include <ccf/stdpub/pubDdt.hpp>
#include <ccf/control/application.hpp>
#include <ccf/control/comAdptSim.hpp>

```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[]) {

  ccf::control::Application& application = ccf::control::Application::Instance();

  CCFINFO("Application started.");

  //>>>> Normal Mode and Simulation Communication Adapter factory objects:
  ccf::control::ComAdptSim com_adapter;
  ccf::control::ComAdptSim sim_com_adapter;
  application.RegisterComAdapters(com_adapter, sim_com_adapter);

  //>>>> Register Processing Recipe Adapter factory objects:
  CCFINFO("Allocate and register Processing Recipe Adapter factory objects ...");
  ccf::common::RecipeBase dummy_rec_object;
  ccf::common::RecipeBase::AddRecipeFactoryObj(dummy_rec_object);

  //>>>> Register Publisher Adapter factory objects:
  CCFINFO("Allocate and register Publisher Adapter factory objects ...");
  ccf::stdpub::PubFits fits_pub_factory_object;
  ccf::common::PubBase::AddPubFactoryObj(fits_pub_factory_object);
  ccf::stdpub::PubDdt ddt_pub_factory_object;
  ccf::stdpub::PubDdt::AddPubFactoryObj(ddt_pub_factory_object);
  ccf::common::PubBase dummy_pub_factory_object;
  ccf::common::PubBase::AddPubFactoryObj(dummy_pub_factory_object);

  //>>> Execute the application:
  CCFINFO("Execute the application ...");
  int stat = application.Execute(argc, argv);

  CCFINFO("Application terminating ...");
  return stat;
}
```

As can be seen from the example of a CCF Control, C++ main function above, a deployment module consists of the following main parts:

- Include the necessary header files.
- Instantiate the CCF Control “Application” class.
- Create instances of the Communication Adapters for normal and simulated operation and register these.
- Create factory instances of the Processing Recipes to be deployed according to the configuration and register these.
- Create factory instances of the Data Publishers to be deployed according to the configuration and register these.
- Execute the CCF Control application by invoking the “ccf::control::Application::Execute()” method. This will handle the parsing of the command line options, loading, parsing and installa-

tion of the various configuration data, and instantiation of internal threads, queues, Recipes and Data Publishers according to the Configuration specified.

8.3 CCF Control - Execution - Example

In the following an example is shown, which can be executed directly from the shell after the ICS software package have been installed. To do this, carry out the following steps:

1. Ensure “DATAROOT”, “CFGPATH” and “INTROOT” are defined. Example:

```
$ echo $DATAROOT
/scratch/jknudstr/DATAROOT
$ ll /scratch/jknudstr/DATAROOT
drwxr-xr-x 2 jknudstr vlt 80 May 11 14:48 image/
$ echo $INTROOT
/scratch/jknudstr/INTROOT
```

2. Start Redis if not already running: The configuration parameter “sys.db.endpoint”, defines the URI.

3. Start the DDT Broker (if desirable) if not already running: The parameter “proc1.pub1.ddt.broker” in the Initialisation Setup defines the URI of the broker, e.g. in this case:

```
$ nohup ddtBroker --uri zpb.rr://*:12011/broker &> /dev/null &
```

4. Execute the example CCF binary “ccfCtrlSim”:

```
$ ccfCtrlSim --config ccf/control/configDdt1.yaml -l INFO
INFO - ../sim/src/main.cpp:26:main:ccfCtrlSim: Application started.
INFO - ../sim/src/main.cpp:34:main:ccfCtrlSim: Allocate and register Processing␣
↪Recipe Adapter factory objects ...
INFO - ../common/src/recipeBase.cpp:40:AddRecipeFactoryObj:ccfCtrlSim:␣
↪Registering Processing Recipe factory object. Classname:␣
↪ccf::common::RecipeBase
INFO - ../sim/src/main.cpp:39:main:ccfCtrlSim: Allocate and register Publisher␣
↪Adapter factory objects ...
INFO - ../common/src/pubBase.cpp:175:AddPubFactoryObj:ccfCtrlSim: Registering␣
↪Data Pub factory object. Classname: ccf::stdpub::PubFits
INFO - ../common/src/pubBase.cpp:175:AddPubFactoryObj:ccfCtrlSim: Registering␣
↪Data Pub factory object. Classname: ccf::stdpub::PubDdt
INFO - ../common/src/pubBase.cpp:175:AddPubFactoryObj:ccfCtrlSim: Registering␣
↪Data Pub factory object. Classname: ccf::common::PubBase
INFO - ../sim/src/main.cpp:47:main:ccfCtrlSim: Execute the application ...
INFO - ../control/src/config.cpp:127:LoadConfig:ccfCtrlSim: Loading␣
↪configuration: ccf/control/configDdt1.yaml ...
INFO - ../control/src/config.cpp:129:LoadConfig:ccfCtrlSim: Loaded␣
↪configuration: ccf/control/configDdt1.yaml
15:45:30.276:INFO:ccf: ../control/src/config.cpp:99:ParseOptions:ccfCtrlSim:␣
↪Process name: ccfCtrlSim
```

(continues on next page)

(continued from previous page)

```
15:45:30.276:INFO:ccf: ../control/src/config.cpp:106:ParseOptions:ccfCtrlSim:␣
↪Configuration: ccf/control/configDdt1.yaml
15:45:30.304:INFO:ccf: ../control/src/application.
↪cpp:428:LoadInitSetup:ccfCtrlSim: Initialisation Setup specified: ccf/control/
↪initSetupDdt1.yaml
15:45:30.414:INFO:ccf: ../control/src/application.
↪cpp:285:CreateThreads:ccfCtrlSim: Creating Monitor Thread: CcfMonThr ...
15:45:30.414:INFO:ccf: ../control/src/application.
↪cpp:293:CreateThreads:ccfCtrlSim: Creating Acquisition Thread: CcfAcqThr ...
...
15:48:39.493:INFO:ccf: ../common/src/pubBase.cpp:193:CreatePubObj:ccfCtrlSim:␣
↪Creating Data Publisher object of type: ccf::stdpub::PubFits
15:48:39.493:INFO:ccf: ../common/src/pubBase.cpp:193:CreatePubObj:ccfCtrlSim:␣
↪Creating Data Publisher object of type: ccf::common::PubBase
15:48:39.494:INFO:ccf: ../common/src/db.cpp:90:UpdateSmStatus:ccfCtrlSim: State␣
↪changed: || -> |On::NotOperational::NotReady/On::NotOperational/On/ |
```

Note, some logging information has been removed. In general the logging and error handling will be improved for future releases.

5. Start DDT Viewer if desirable: In this case:

```
$ nohup ddtViewer -s "zpb.rr://127.0.0.1:12011/broker/Broker1 CcfTest11" &
```

6. Bring Operational, Start Acquisition:

```
$ dcsSend zpb.rr://127.0.0.1:12092 Init
OK
$ dcsSend zpb.rr://127.0.0.1:12092 Enable
OK
$ dcsSend zpb.rr://127.0.0.1:12092 GetState
On::Operational::Idle/On::Operational/On/
$ dcsSend zpb.rr://127.0.0.1:12092 Start
OK
$ dcsSend zpb.rr://127.0.0.1:12092 GetState
On::Operational::Acquisition::NotRecording/On::Operational::Acquisition/
↪On::Operational/On/
```

6. Execute a Recording Session:

```
$ dcsSend zpb.rr://127.0.0.1:12092 RecStart
Recording Status:
ID:                          RecId-25615696-ceec-461b-a45d-5b73b8ed2278
Status:                      Active
Frames Processed:            0
Frames Remaining:            7
Start Time:                  2021-05-11T14:48:07.390597
Time Elapsed:                0.000000
Remaining Time:              7.000000
```

(continues on next page)

(continued from previous page)

```
Estimated Completion Time: 2021-05-11T14:48:14.390597
Volume Recorded (bytes):   0
Files Generated:           0
Info:
Output Files: |

$ dcsSend zpb.rr://127.0.0.1:12092 RecStatus
Recording Status:
ID:                        RecId-25615696-ceec-461b-a45d-5b73b8ed2278
Status:                    Active
Frames Processed:          5
Frames Remaining:          2
Start Time:                2021-05-11T14:48:07.390597
Time Elapsed:              5.007639
Remaining Time:            1.992361
Estimated Completion Time: 2021-05-11T14:48:14.390597
Volume Recorded (bytes):   5242880
Files Generated:           2
Info:
Output Files: |
  /scratch/jknudstr/DATAROOT/image/PubFitsTest1.fits
  /scratch/jknudstr/DATAROOT/image/PubFitsTest2.fits
```

8.4 CCF Generate FITS Cube Tool ("ccfGenFitsCube")

The "ccfGenFitsCube", provided by the CCF Package, can be used to generate FITS cubes to be used as input for the Simulation Communication Adapter, which plays back the images in a FITS cubes. The man-page speaks for itself:

```
> ccfGenFitsCube --help
usage: ccfGenFitsCube [-h] [--size SIZE] [--noise NOISE]
                      [--magnitude MAGNITUDE]
                      [--data-type {INT16,INT32,FLOAT,DOUBLE}]
                      [--nb-of-images NB_OF_IMAGES]
                      [--max-shift-per-image MAX_SHIFT_PER_IMAGE]
                      [--max-total-shift MAX_TOTAL_SHIFT]
                      [--output-file OUTPUT_FILE]

CCF Test FITS Cube Generator

optional arguments:
  -h, --help                            show this help message and exit
  --size SIZE                           pixels in X/Y
  --noise NOISE                         noise to apply [0; <noise>]
  --magnitude MAGNITUDE                 magnitude of simulated star (max␣
↪value)
  --data-type {INT16,INT32,FLOAT,DOUBLE}    type of pixel values in output file
```

(continues on next page)

(continued from previous page)

```
  --nb-of-images NB_OF_IMAGES                 number of frames in the cube
  --max-shift-per-image MAX_SHIFT_PER_IMAGE   max shift of sim star in pixels per␣
↪image
  --max-total-shift MAX_TOTAL_SHIFT           max total shift of sim star in␣
↪pixels per image
  --output-file OUTPUT_FILE                   name of output FITS file
```

Note: The CCF Simulation Communication Adpater provided, only supports Int8 and Int16.

8.5 System Tuning

CCF Control has been designed to achieve maximum throughput by using three levels of parallelism for the

- data acquisition,
- data processing, and
- data publishing.

In this way, the latency is reduced. However, CCF only handles entire image frames at this point in time. I.e., it is not possible to handle sub-frames (of images), which would make it possible to reduce the latency futher.

As mentioned in the "Overview" chapter, a number of internal frame buffer queues are used. These are:

- **Input Queue:** Used by the Acqusition Thread, in a ring-buffer fashion, to store incoming image frames, to make them available for the Processing Pipelines.
- **Output Queues:** Onces the Processing Recipes have been executed on the image frames, these are available in the Output Queues, one per Processing Thread, for the Publisher Threads.

In a perfectly balanced/calibrated system, in theory, two buffers in each queue would be enough. However, when deploying CCF on a non real-time OS, a certain amount of jitter is to be expected. In order to average out the effect of this jitter of the frame handling, it is recommended to allocate additional buffers, to compensate. If no free frames are available in a queue, the thread handling the frame object, will have to skip (drop) that frame. Whether such frame skipping is acceptable, can be configured via the configuration parameters:

- "acq.allow_frame_skipping"
- "proc#.allow_frame_skipping"

If set to true, frames will be skipped/dropped silently. Otherwise, a log will be produced, indicating the frame skipping. This log will be produced with a maximum periodicity of 10 s to avoid possible flooding of the logging system.

The frames skipped is updated in the OLDB, in the statistics section for each type thread.

If frames are skipped in either type of queues (Input Queue or Output Queues), this may be an indication that the processing of the image frames or the publishing thereof, is too slow.

It could be investigated if this can be optimized to increase the throughput and reduce the amount of frames skipped.

Apart from being forced to skip frames due to lack of free space in the internal queues, the Acquisition Thread may detect that frames are lost at the level of receiving these from the camera. This is mostly caused by issues with a poor/inadequate communication channel to the camera, typically the network connection, or that the Acquisition Thread is not fast enough to pick up all frames. When this happens, a log is generated, indicating this. It is possible to suppress this log via the Configuration Parameter "acq.allow_lost_frames".

Like for the frame skipping, the number of lost frames is updated in the statistics of the Acquisition Thread so it is easy to check if the system is running as expected.

8.6 Performance & Statistics

Each of the threads Acquisition, Processing and Publishing, record run-time information about the frame handling.

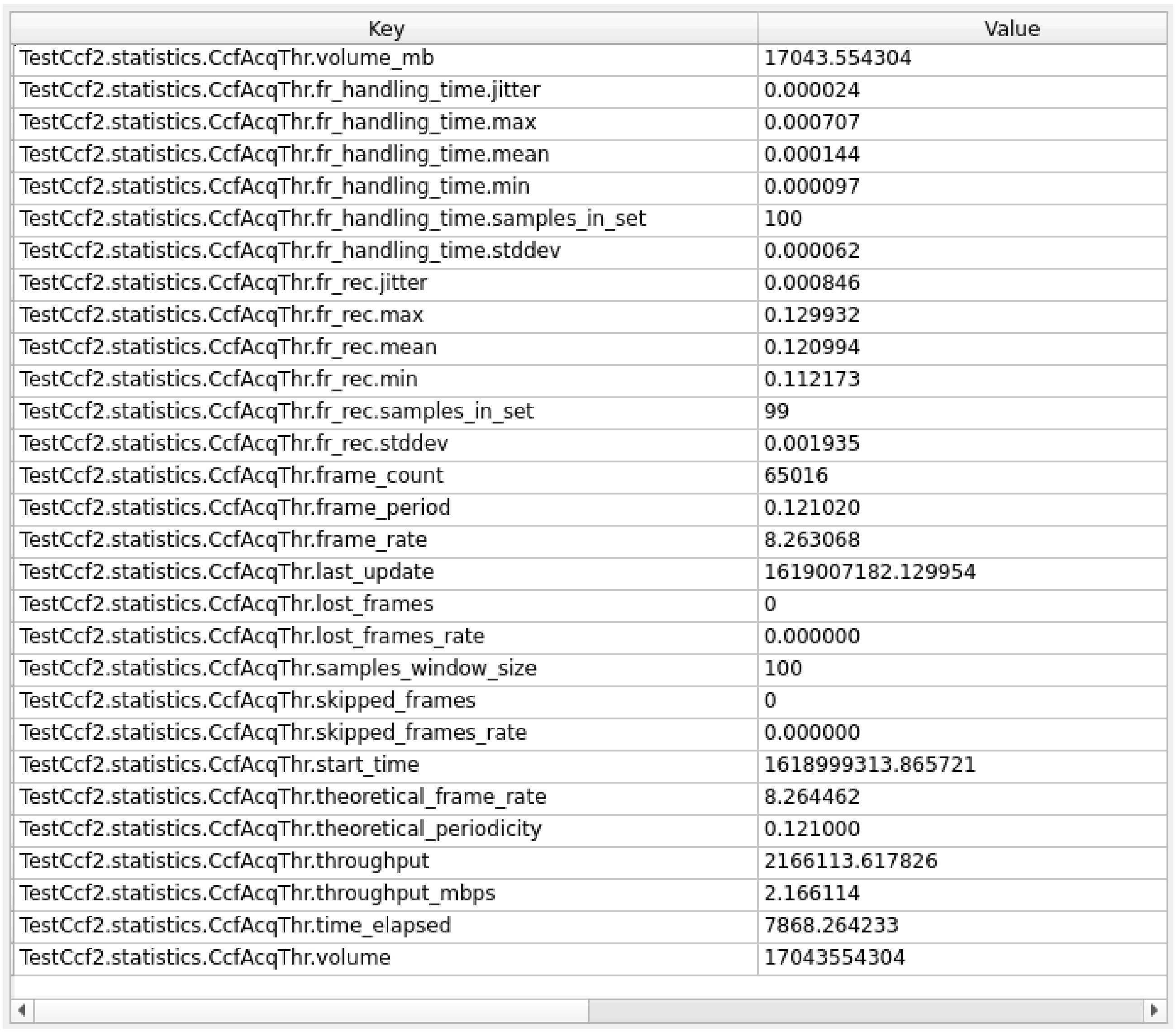
This information, is collected periodically by the Monitor Thread. Subsequently it computes the performance parameters for the execution of the thread, with focus on the frame handling.

The generation of this statistics, is controlled via the Configuration Parameters:

- "mon.period": Period applied for the execution of the Monitor Thread in seconds.
- "mon.nb_of_samples": Number of the samples in the sets (sliding windows) used as basis for computing the statistics; a size of 100 seems to be a reasonable value, whereby this depends on the frame rate. For very slow frame rates, it may take a long time to complete the set.

The statistics is running from the moment the data acquisition was started. It is reset when starting a new data acquisition. It is also reset when updating setup parameters, as these often may influence the statistics, e.g. when the exposure time or frame rate are changed.

An example of the statistics parameters for a thread in shown below (as displayed in the "dbbrowser"):



Key	Value
TestCcf2.statistics.CcfAcqThr.volume_mb	17043.554304
TestCcf2.statistics.CcfAcqThr.fr_handling_time.jitter	0.000024
TestCcf2.statistics.CcfAcqThr.fr_handling_time.max	0.000707
TestCcf2.statistics.CcfAcqThr.fr_handling_time.mean	0.000144
TestCcf2.statistics.CcfAcqThr.fr_handling_time.min	0.000097
TestCcf2.statistics.CcfAcqThr.fr_handling_time.samples_in_set	100
TestCcf2.statistics.CcfAcqThr.fr_handling_time.stddev	0.000062
TestCcf2.statistics.CcfAcqThr.fr_rec.jitter	0.000846
TestCcf2.statistics.CcfAcqThr.fr_rec.max	0.129932
TestCcf2.statistics.CcfAcqThr.fr_rec.mean	0.120994
TestCcf2.statistics.CcfAcqThr.fr_rec.min	0.112173
TestCcf2.statistics.CcfAcqThr.fr_rec.samples_in_set	99
TestCcf2.statistics.CcfAcqThr.fr_rec.stddev	0.001935
TestCcf2.statistics.CcfAcqThr.frame_count	65016
TestCcf2.statistics.CcfAcqThr.frame_period	0.121020
TestCcf2.statistics.CcfAcqThr.frame_rate	8.263068
TestCcf2.statistics.CcfAcqThr.last_update	1619007182.129954
TestCcf2.statistics.CcfAcqThr.lost_frames	0
TestCcf2.statistics.CcfAcqThr.lost_frames_rate	0.000000
TestCcf2.statistics.CcfAcqThr.samples_window_size	100
TestCcf2.statistics.CcfAcqThr.skipped_frames	0
TestCcf2.statistics.CcfAcqThr.skipped_frames_rate	0.000000
TestCcf2.statistics.CcfAcqThr.start_time	1618999313.865721
TestCcf2.statistics.CcfAcqThr.theoretical_frame_rate	8.264462
TestCcf2.statistics.CcfAcqThr.theoretical_periodicity	0.121000
TestCcf2.statistics.CcfAcqThr.throughput	2166113.617826
TestCcf2.statistics.CcfAcqThr.throughput_mbps	2.166114
TestCcf2.statistics.CcfAcqThr.time_elapsed	7868.264233
TestCcf2.statistics.CcfAcqThr.volume	17043554304

The statistics data values are stored in keys with names of the form:

"<sys.db.prefix>.statistics.<thread name>[.<category>].<key>"

The values reported for each thread are (OLDB key prefix "<sys.db.prefix>.statistics.<thread name>" left out):

volume volume_mb	Indicates the accumulated volume handled by the thread in bytes/mega bytes since the data aquisition was started.
frame_handling_time.<key>	The Frame Handling Time refers to the time (in seconds) the frames was handled/processed by the thread, from its reception, until it was delivered to the destination. The values are: *jitter:* Calculated as the average deviation from the mean value. *max:* The maximum frame handling time found in the set. *mean:* The mean value of the samples in the set. *min:* The minimum frame handling time found in the set. *samples_in_set:* The number of samples in the sliding window. *stddev:* The std. deviation of the samples in the set, computed with the std. formula.
fr_rec.<key>	The Frame Reception Time refers to the absolute point in time, frames were received from the camera. The statistics is calculated from the difference in time between two consecutive receptiom time stamps. Refer to **frame_handling_time.<key>** for an explanation of the values computed.
frame_count frame_period frame_rate	The number of frames handled since the acquistion was started, the period of the frame handling (seconds between each set of frames) and the frame rate (frames/second).
last_update	Last time the statistics was updated (seconds since epoch).
lost_frames lost_frames_rate	Number of frames lost receicing these from the camera since the acquisition was started, the frequency of loosing the frames (frames lost/second).
samples_window_size	The size of the sliding window used as basis for calculating the statistics in samples.
skipped_frames skipped_frames_rate	Number of frames skipped (dropped), since the current image acquisition was started, due to lack of space in the queue assocated with the thread + the rate with which frames are being skipped (frames/second).
start_time	Absolute time since epoch for starting the current image acquisition.
theoretical_frame_rate theoretical_periodicity	The theoretical frame rate (frames/seconds) and periodicity (seconds/frame) according to the current setup.
through_put through_put_mbps	The average throughput in bytes/seconds and MB/second.
time_elapsed	Time elapsed in seconds since the current acqusition sequence was started.

CCF provides no service to retrieve the data, nor to generate a report from it. It is up to the client applications to record the data and store it, if of interest.