European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Instrumentation Framework

ELT ICS Framework Data Acquisition User Manual

Document Number: ESO-396401

Document Version: 3

Document Type: User Manual (MAN)

Released on: 2024-12-11

Document Classification: Public

Owner:	Rosenquist, Calle
Validated by PM:	Kornweibel, Nick
Validated by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Approved by PGM:	Tamai, Roberto
	Name

Release

This document corresponds to ifw-daq[1] v3.1.0.

Authors

Name	Affiliation
Rosenquist, Calle	ESO

Change Record from Previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1517
All	All sections updated
3.1,9.1.3,9.3,11.4	New sections added
14-16	Sections removed

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

Contents

1 Introduction

1.1 Scope

This document is the user manual for the ELT ICS Framework products *OCM* and *DPM* and covers:

- *Overview* of the design and concepts
- *Software context* and *deployment*
- *Data Acquisition guide*
- Interface documentation

The intended audience are instrument or instrument framework developers. Readers are assumed to be familiar with core ELT technologies:

- CII MAL (request/reply and publish/subscribe)
- FITS (Header/Data Units, keywords, extensions)

1.2 Acronyms

CII
Core Integration Infrastructure

DAQ
Data Acquisition

DPM
Data Product Manager

In some contexts DPM may colloquially be referring to the application daqDpmServer.

DICD
Data Interface Control Document

ELT
Extremely Large Telescope

FITS
Flexible Image Transport System

DHS
Data Handling Server where *DPM* can deliver merged products to On-Line Archive System

OCM
Observation Coordination Manager.

In some contexts OCM may colloquially be referring to the application daqOcmServer.

OCS
Observation Coordination System

OLAS
On-Line Archive System

OLDB
On-Line Database

RAD
Rapid Application Development toolkit. See also *RD9*.

ICD
Interface Control Document (may also refer to MAL XML Interface document)

ICS
Instrument Control System

JSON
JavaScript Object Notation

MAL
Middleware Abstraction Layer

TBC
To be confirmed

TBD
To be determined

1.3 Definitions

Archive File Name
Archive File Name is the unique name of the *Data Product* in the ESO Science Archive, which follows the pattern $\{\text{fileId}\}.\text{fits}$, where $\{\text{fileId}\}$ has the pattern $\{\text{insPrefix}\}.\{\text{timestamp}\}$ and $\{\text{insPrefix}\}$ is the configured instrument prefix and $\{\text{timestamp}\}$ is the ISO 8601 timestamp format $\text{YYYY-MM-DDThh:mm:ss.sss}$.

The Archive File Name is recorded in primary HDU FITS keyword ARCFILE in all *Data Products* created by *OCM*.

Example name: $\text{KMOS.2023-01-08T09:27:03.966.fits}$

See also:

- *File Id*
- *RD7*

Config Path
Term used in this document to refer to configuration paths resolved using the environment variable $\$\text{CFGPATH}$. The path can either be absolute or relative. In case of relative path the file is resolved by testing the combination of each path in $\$\text{CFGPATH}$ with the provided relative path.

For example if $\$\text{CFGPATH}=/\text{tmp/a}:/\text{tmp/b}$ and Config Path is the file c the tested paths are

(in order):

1. /tmp/a/c
2. /tmp/b/c

Commentary Keyword
In this document a FITS Commentary Keyword refers to those keywords that are neither *Value Keyword*s or *ESO Keyword*s. Commentary Keywords are special because the same keyword name may occur multiple times in the same HDU, to e.g. continue a comment over multiple records.

Note: Strictly speaking the HIERARCH keyword is also a commentary keyword but is treated specially in applications that support the convention.

Examples are:

```
COMMENT  Commentary keyword names may occur multiple times in a header, it may also
COMMENT  be contextual, as in this case, where a comment is continued over multiple
COMMENT  records.
HISTORY  File modified by user 'USER' on host on 2021-09-03T01:28:26
```

Data Acquisition
In this document it refers to the process of acquiring data as coordinated by *OCM* which results in one or more FITS files that contain the primary HDU and zero or more extensions such as binary tables, and or FITS keywords only. These are then merged together used to create a Data Product by *DPM*.

Refer to section *Conceptual Model* for a high level conceptual model of the process involved and *Process Overview* for a concrete and detailed overview.

Data Product
In this document it refers to the FITS science data product that is produced by DPM using the acquired data (see *Data Acquisition*) and normally archived in the ESO Online Archive System.

Data Product Specification
Specifies how to create a *Data Product* from input sources. This specification is created by *daqDpmServer* and used by *daqDpmMerge*.

ESO Keyword
See *ESO Hierarch Keyword*.

ESO Hierarch Keyword
Refers to FITS keywords following the ESO HIERARCH keyword conventions *RD6*, i.e. keywords of the form:

```
HIERARCH ESO INS FILT1 ENC = 2 / Filter wheel absolute position [Enc].
HIERARCH ESO INS FILT1 ID = 'OUT' / Filter unique id.
```

The first token, INS in the example above, is referred to as the *category*.

File Id

Unique identifier for all *Data Products* in the ESO Science Archive and basis for the *Archive File Name*.

File Id has the pattern {insPrefix}.{timestamp} and {insPrefix} is the configured instrument prefix and {timestamp} is the ISO 8601 timestamp format YYYY-MM-DDThh:mm:ss.sss.

When a new *Data Acquisition* is initialized *daqOcmServer* will allocate the id which can be read from e.g. fileId.

Example name: KMOS.2023-01-08T09:27:03.966

Uniqueness is ensured by having *daqOcmServer* create the id and that the following assumptions hold true:

- There is only one *daqOcmServer* instance for each instrument prefix.
- The time used to generate the identifier is correct.

See also:

- *Archive File Name*
- *RD7*

Keyword Classification

Refers to the CFITSIO classification for a given keyword. This can e.g. be structural keywords like SIMPLE or NAXIS, WCS like CTYPEn or CUNITn. Keywords that are not known to CFITSIO are referred to as *user* keywords. See *RD8* for a complete list of categories.

Keyword Type

Refers to the different FITS standard[*RD5*] keyword record specifications which can be any of:

- *Value Keyword*
- *ESO Keyword*
- *Commentary Keyword*

Logical Keyword Name

The logical name of a FITS keyword depends on the type of keyword, but there are common traits: Trailing white spaces are insignificant and are not part of the Logical Keyword Name.

For FITS *Value Keyword* and *Commentary Keyword* the logical name is the white-space trimmed name component. The logical names for

```
NAXIS1  =                 2048 / # of pixels in axis1
COMMENT This table was written by 'APPLICATION'
```

are NAXIS1 and COMMENT respectively.

For *ESO Keyword* the logical name is the part between HIERARCH ESO up to =. For example the logical name is INS FILT1 ID for the HIERARCH keyword:

```
HIERARCH ESO INS FILT1 ID = 'OUT' / Filter unique id.
```

metadaqif
: Standard metadata source interface used by e.g. FCF and other components [*RD3*].

daqOcmServer
: Name of the OCM main server executable.

daqOcmCtl
: Name of the OCM command line client executable.

recif
: Standard primary data source interface implemented by DCSs [*RD4*].

Value Keyword
: A FITS Value Keyword are those keywords that have a value indicator in bytes 9 and 10 (c.f. *RD5*). Example keywords are:

```
SIMPLE  =                    T / Standard FITS
BITPIX  =                    8 / # of bits per pix value
NAXIS   =                    0 / # of axes in data array
```

2 Related Documents

2.1 Reference Documents

[RD1]

Data Interface Control Document;
ESO-044156 v7[1]

[RD2]

Technical Note on Inter-ICS Data Acquisition;
ESO-356883 v1[2]

[RD3]

Metadata Acquisition Interface;
https://gitlab.eso.org/ecs/ecs-interfaces/-/tree/master/metadaq

[RD4]

DCS Recording Interface;
https://gitlab.eso.org/ecs/ecs-interfaces/-/tree/master/rec

[RD5]

Definition of the Flexible Image Transport System (FITS);
https://fits.gsfc.nasa.gov/standard30/fits_standard30aa.pdf

[RD6]

The ESO HIERARCH Keyword Conventions;
https://fits.gsfc.nasa.gov/registry/hierarch/hierarch.pdf

[RD7]

ICD between ICS and OLAS;
ESO-384590 v1[3]

[RD8]

[1] https://pdm.eso.org/kronodoc/HQ/ESO-044156/7
[2] https://pdm.eso.org/kronodoc/HQ/ESO-356883/1
[3] https://pdm.eso.org/kronodoc/HQ/ESO-384590/1

CFITSIO User's Reference Guide;
https://heasarc.gsfc.nasa.gov/docs/software/fitsio/c/c_user/cfitsio.html

[RD9]

ELT ICS Framework - Application Framework - User Manual;
ESO-363137 v4[4]

[RD10]

ELT ICS Framework - Data Interface Tools - User Manual;
ESO-319696 v2[5]

[RD11]

Data Acquisition Interface - daqif
https://gitlab.eso.org/ifw/ifw-daqif/

[4] https://pdm.eso.org/kronodoc/HQ/ESO-363137/4
[5] https://pdm.eso.org/kronodoc/HQ/ESO-319696/2

3 Style Conventions

3.1 Interactive Shell Sessions

When examples are given in the following form it means the command `command` is executed with argument `--argument` which produces output `output` to the console:

```
$ command --argument
output
```

In limited cases the syntax highlighting can be incorrect, but it should be clear from the context what is meant.

3.2 JSON Data Structures

This manual documents a number of JSON data structures, their "schema", using these common conventions. In some cases a more formal documentation in the form of JSON Schema[6] is also provided.

Typically a concrete example is first provided:

Listing 3.1: Example JSON data structure with sum-type list

```
{
  "name": "example",
  "optional": 3.0,
  "nestedAnonymous": {
    "nestedProperty": "value"
  },
  "listOfObjectsProperty": [
    {
      "type": "unionTypeA",
      "property": "value"
    },
    {
      "type": "unionTypeB",
      "anotherProperty": "value"
    }
  ]
}
```

This top level structure is then documented, sometimes with named sub-objects. The types of each property uses Python Variable Annotation[7] syntax. For example a string would be `str`, a JSON object

[6] https://json-schema.org/
[7] https://www.python.org/dev/peps/pep-0526/

would be object, a list of strings would be List[str], a union of two alternative named objects like above would use Union[type1, type2], a list of union of named types would be List[Union[type1, type2]].

Unions are used when multiple types alternatives are allowed. The way it is typically used for is to have multiple object alternatives. In this case the union variant is discriminated using a type property with a fixed literal string value. The example in Listing 3.1 contain a list of union type using this pattern.

Each property of an object, consisting of a <name>, <type> and <default> is then documented as follows:

<name> **(<type>) <default>**
: Documentation for property with name “<name>” type “<type>” and optionally a default value <default>.

  If <type> must have a specific value the literal value may be indicated instead.

  If property is optional the <type> can specify this using Python annotation syntax: Optional[<type>]. If there is a default value <default> specifies that as [default DEFAULTVALUE]. Lists are often optional and in this case it is simply indicated by a default empty list value: [default: []]. When a property is optional and no value should be provided the property should simply be omitted from the object rather than specify a null value.

  If a property is an anonymous object with nested properties it will be documented inline with nested indentation (see example below).

Documenting the example structure above would look like:

Root object

name **(str)**
: Name property.

optional **(Optional[float]) [default: 1.0]**
: Optional property of float type with a default value of 1.0.

nestedAnonymous **(object)**
: Example of how nested anonymous objects are documented.

  nestedProperty **(str)**
  : A nested property of this anonymous object.

listOfObjectsProperty **(List[Union[UnionTypeA, UnionTypeB])**
: Contains a list of either UnionTypeA or UnionTypeB types. See below for their structure.

UnionTypeA

Documents named object “UnionTypeA”.

type **(“unionTypeA”)**
: Union discriminator and must have the literal value "unionTypeA".

property **(str)**
: Example property of UnionTypeA.

UnionTypeB

Documents named object “UnionTypeB”.

type **(“unionTypeB”)**
Union discriminator and must have the literal value "unionTypeB".

anotherProperty **(str)**
Example property of UnionTypeB.

4 ELT for VLT Developer

This section aims to help introduce developers familiar with VLT software to similar or related ELT concepts.

Table 4.1: Concept mapping between VLT and ELT

VLT	ELT	Comment
BOSS	OCM	The main component coordinating the exposures/Data Acquisitions. In VLT it also had a supervisory and role for which commands were forwarded through BOSS. BOSS was also a common point of customization to add new commands for e.g. slow guiding. This is not the case in ELT where direct setup is the norm and custom behaviour is implemented as separate components.
BOSS Archiver	DPM	Component responsible for creating the final Data Product and deliver it to the Archive (*OLAS*). In ELT DPM will also be able to deliver the final Data Product to more than one receiver, but the standard and default receiver will remain to be *OLAS*.
Exposure	*Data Acquisition*	The term has been generalized for ELT to better suit the wide range of data that can be acquired. See glossary *Data Acquisition* for a description.
Expoid	*Data Acquisition* Id	Uniquely identifies an exposure/*Data Acquisition*. This is a mandatory parameter in most Data Acquisition commands in *OCM*.
dxfFileSend / dxfFileReceive	*DPM*	In VLT BOSS Archiver requires that all files are available on locally mounted file system. DXF applications dxfFileSend and dxfFileReceive are used to send files to the workstation where merging takes place. In ELT *DPM* take over this responsibilty and will automatically retreive all remote files before executing the merge.
VCSOLAC	*DPM*	In VLT VCSOLAC is responsible for monitoring used to transfer the merged file from BOSS Archiver to DHS workstation. In ELT *DPM* is designed to be deployable on DHS workstation (see *Deployment*) in which case the file is delivered to OLAS locally. In case *DPM* is not deployed on the DHS workstation the file is transferred by *DPM* as part of the file delivery.

5 Components

Figure Fig. 5.1 show the application components provided by this software package.

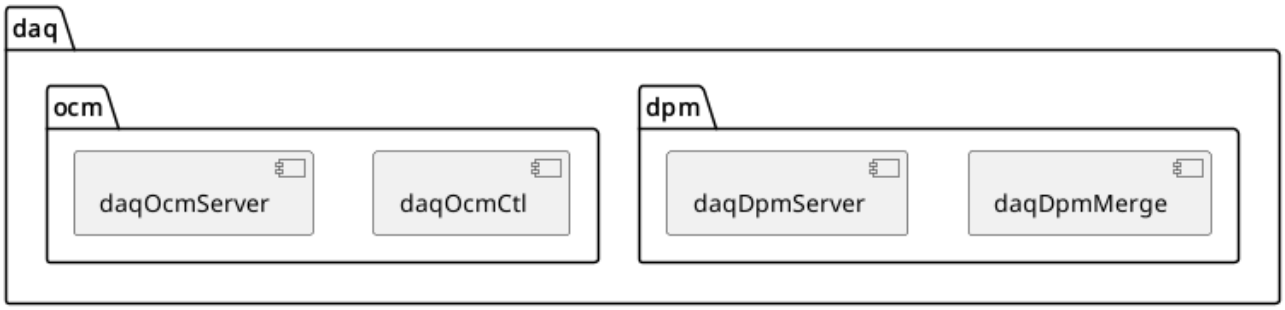


Fig. 5.1: Package and main components overview

daqOcmServer
Coordinates the full *Data Acquisition* life-cycle.

Main responsibilities:

- Act as interface for creating *Data Acquisitions*.
- Provide life-cycle control of *Data Acquisitions*.

daqOcmCtl
Controls *daqOcmServer* from command line.

daqDpmServer
Data Product Manager server component.

Main responsibilities:

- Queues *Data Acquisitions* and schedules them for merging.
- Copy inputs for *Data Product* from source nodes to local storage.
- Execute *daqDpmMerge* to create *Data Product*.
- Deliver *Data Product* to specified receivers (normally *OLAS*).

daqDpmMerge
Standalone (command line) executable.

Main responsibilities:

- Create *Data Product* from *Data Product Specification* referencing and containing input sources (FITS files and keywords).

6 Software Context

This section describes the software context of *OCM* and *DPM*, focusing on provided and used software interfaces. Fig. 6.1 shows the components with their relations to interfaces which also specifies their dependencies. The individual interfaces are described for *daqOcmServer* in Table 6.1 and *daqDpmServer* in Table 6.2.

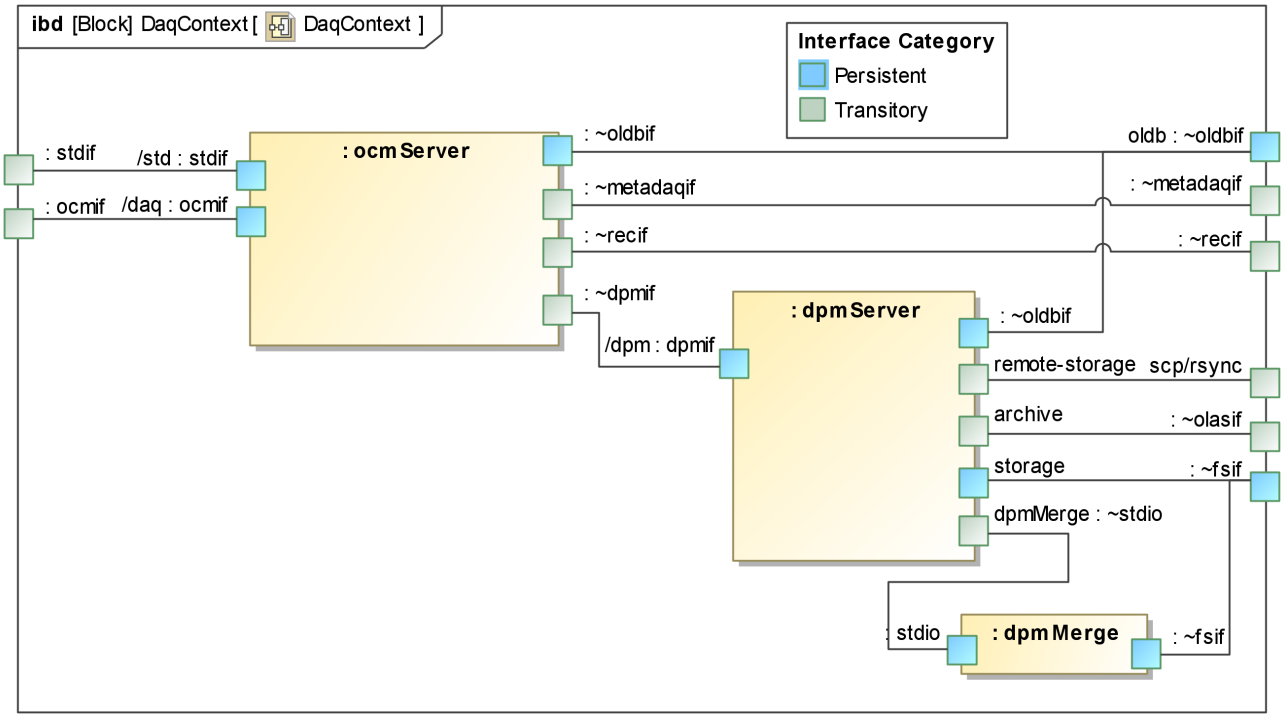


Fig. 6.1: Software context for *daqOcmServer* and *daqDpmServer*. The interface category refers to whether the interface is used (or provided) persistently or in a transitory manner e.g. because interface is used only for the duration of an operation. Non-conjugated ports (without ~) indicate that the interface is provided whereas conjugated ports (with ~) indicate a interface is used.

Table 6.1: Interface Description: daqOcmServer

Interface	Description
stdif	Standard interface implemented and provided by *OCM* which enables operational life-cycle control and supervision.
daqif.OcmDaqControl	Interface implemented and provided by *OCM* to manage *Data Acquisitions*.
~oldb	*OCM* has a persistent dependency to the Online Database to update it with current configuration and application status.
~metadaqif	*OCM* use the metadaqif to acquire data from metadata sources for the duration of each *Data Acquisition*.
~recif	*OCM* use the recif to acquire data from primary data sources for the duration of each *Data Acquisition*.
~daqif.DpmDaqControl	If *DPM* is operational *OCM* will send commands to schedule the individual *Data Acquisitions* to be merged into *Data Products*.

Table 6.2: Interface Description: daqDpmServer

Interface	Description
daqif.DpmDaqControl	Interface provided by *DPM* for scheduling *Data Acquisition* to be merged into *Data Product*.
~oldb	*DPM* has a persistent dependency to the Online Database (redis-server) to update it with current configuration, application status and status of all *Data Acquisitions*.
remote-storage	Interface to remote file systems used to fetch the individual files to be merged into *Data Products* or send final *Data Product* to a receiver.
archive	Interface used to deliver *Data Products* to *OLAS*.
storage	*DPM* stores all files on a file system before merging is performed as well as the resulting *Data Products*.
daqDpmMerge	To perform the merge *daqDpmServer* will execute *daqDpmMerge* as a subprocess and monitor it using standard I/O pipes. To abort merging signals will be used.

7 Deployment

7.1 Reference

Two reference models for deployment are provided below in Fig. 7.1 and Fig. 7.2. The node *Instrument Workstation* below refers to any instrument provided node. There are no particular requirements or constraints unless *daqDpmServer* is deployed there.

Both *daqOcmServer* and *daqDpmServer* have a private workspace on the local file system that persist independently of the deployed process and is used for state persistence and recovery. For additional details see sections for *OCM* and *DPM* respectively.

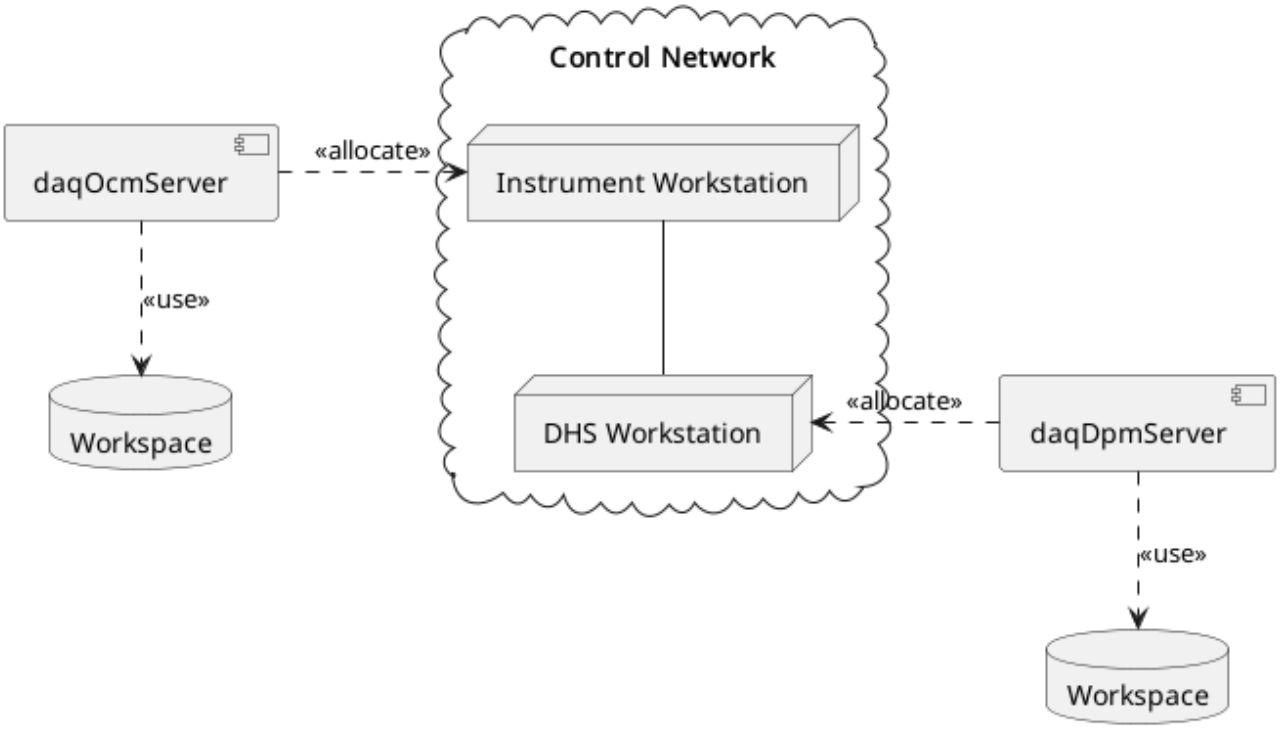


Fig. 7.1: Reference deployment showing how daqDpmServer is deployed on the Data Handling Server Workstation. Advantage of this deployment is that daqDpmServer does not need to transfer the *Data Product* to DHS Workstation. Both daqDpmServer and daqOcmServer have a private workspace on the local file system where they are deployed.

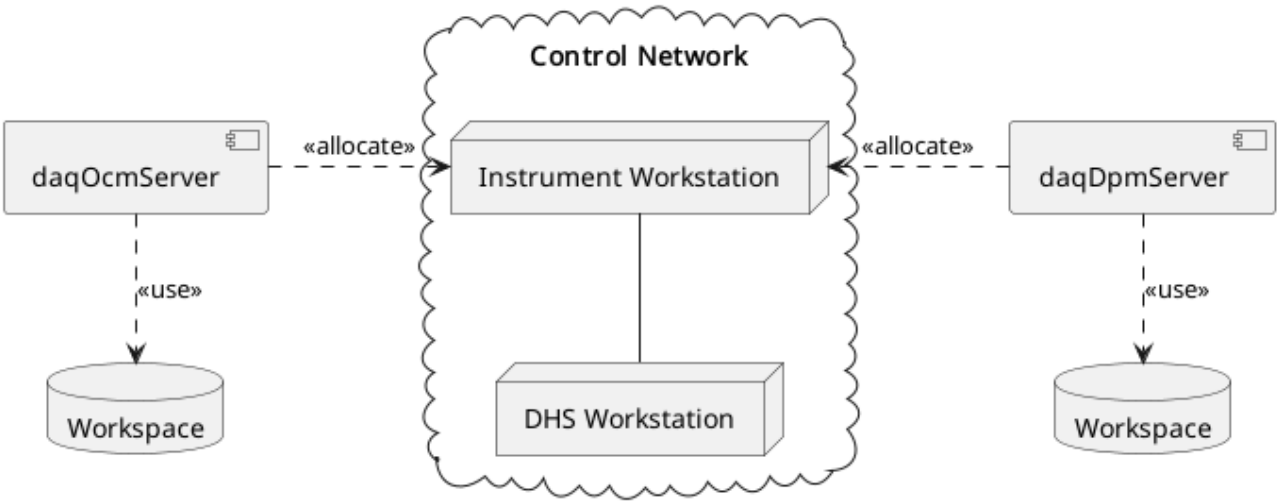


Fig. 7.2: Alternative deployment showing how daqDpmServer is deployed on an Instrument Workstation. In this deployment daqDpmServer needs to transfer the *Data Product* to DHS Workstation for archiving. Both daqDpmServer and daqOcmServer have a private workspace on the local file system where they are deployed.

7.2 Deployment Constraints

Software Context also show dependencies to neighbouring components or systems, as anonymous interfaces, as well as the temporal dimension with the interface category. If a dependency is transitory because it e.g. is only required to complete an operation, for the duration of the operation, this will also be noted below.

The following constraints needs to be taken into consideration when deploying each component:

daqOcmServer

Apart from satisfying runtime dependencies and services enumerated in Table 6.1 there are no specific deployment constraints for *OCM*. The only persistent service used by *daqOcmServer* is the oldb (redis server).

To acquire data from a data source *daqOcmServer* requires request/reply access for the duration of $\mathrm{StateAcquiring}$. Once data has been acquired there is no connection or dependency towards any data source.

daqDpmServer

daqDpmServer and its subprocesses may consume a lot of I/O resources. Both network I/O when collecting source files to the local host or delivering the *Data Product* to receivers and disk I/O when *Data Product* is created from the input source files. As such this needs to be dimensioned.

daqDpmServer is special in that it does not require an operational ICS to perform its function. This is also the reason why it does not provide the standard interface stdif, because it does not require state management; if it is running it is operational. However, since *daqOcmServer*

continuously receives *Data Acquisition* status updates from *daqDpmServer* it is strongly recommended to operate *daqDpmServer* with *daqOcmServer* running to have consistency between them (state *NotOperational* is also ok).

daqDpmServer can also be deployed multiple times on the same host, each with their own private workspace. It is e.g. a valid option to deploy multiple instances on the same *OLAS* workstation, that may be shared by multiple instruments. In this scenario it is recommended that each *daqDpmServer* runs as an instrument specific user. There is no coordination between *daqDpmServer* instances and each may compete with available resources.

To create data products from the individually created FITS files *daqDpmServer* requires access to the files using either locally mounted file system access or remotely with properly configured unattended *rsync* access. The same is true for releasing the completed *Data Product* to remote hosts such as the *DHS*.

At this time public key authentication should be used.

Example configuration:

```
# ~/.ssh/config

# DPM access to hosts providing FITS files
Host 10.0.129.1
    IdentityFile /path/to/public/key/id_rsa
Host 10.0.129.10
    IdentityFile /path/to/public/key/id_rsa

# To simplify configuration logical names can be used
# which then correspond to a concrete host.
Host dhs
    HostName 10.0.129.11
    IdentityFile /path/to/public/key/id_rsa
```

8 Overview

This section gives an overview of the components involved in acquiring data, which is performed by each component that provides data to the final data product and is coordinated by *OCM* and creating the data product to be archived and notifying the Online Archive System, which is performed by *DPM*. This means that other components in OCS are regularly not included.

8.1 Introduction

8.1.1 Stylistic Conventions

Note: Although diagrams sometime follow UML styling they are not created to be formally correct, but to convey information efficiently.

The following visual convention is used for components (or systems), control and data flow:

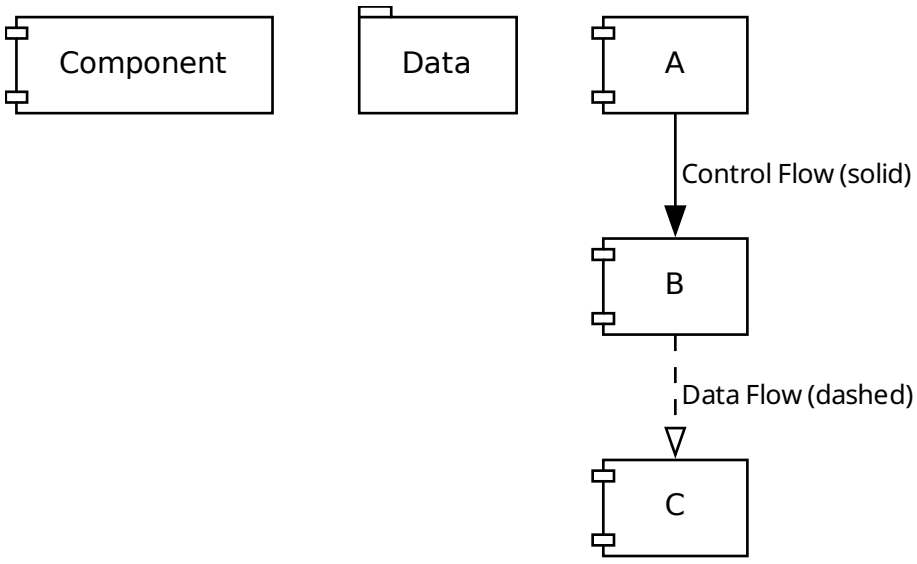


Fig. 8.1: Stylistic conventions

The direction of control refers to who the initiator is. The most common case is to use request/reply in which case the diagram shows how requestor initiates the control from A to command B.

Data flow is typically reserved for the out-of-band information that is not carried as part of the control

flow. In practice data flow in this manual usually means FITS files being created by a component and that are later transferred for consumption in another component. Strictly speaking the actual transfer then depends on how components are deployed. The data flow can be thought both as the logical information transfer and the physical information transfer, depending on deployment and context.

Note: There are exceptions to using request/reply for e.g. the interaction with OLAS, but in this overview it can be considered conceptually equivalent.

8.1.2 Conceptual Model

A conceptual data model can be useful as a basis for understanding the more detailed documentation. The following diagram show data flow, data entities and multiplicities for how a single *Data Product* is created for a single *Data Acquisition*:

- There is one instance of OCM coordinating the *Data Acquisition* and acts as the interaction point for clients.
- For each *Data Acquisition* there are 1 to any number of *Data Sources* providing data. A *Data Source* can be any process that implement the supported MAL interfaces from [*RD3*,*RD4*].

  Tersely the *Data Sources* implementing Metadata Acquisition Interface [*RD3*] are started/stopped automatically based on the state change of primary *Data Sources*:

  - Are informed by *OCM* when to start/stop. This happens before any primary *Data Sources* are started and after all primary *Data Sources* are stopped.
  - Can provide FITS keywords via FITS file or via MAL command response.
  - Can provide one or more FITS files containing keywords and/or extensions.
  - Examples: FCS Device Manager and RTCTK Metadata Collector.

  The *Data Sources* implementing DCS Recording Interface [*RD4*]:

  - Are informed by *OCM* when to start (either directly or synchronized to e.g. an absolute time).
  - Can be stopped by *OCM* by the request of a user with $\mathrm{StopDaq}()$. It will not be stopped automatically by *OCM*.
  - Can decide to stop by itself (if e.g. configured with a fixed integration time), in which case it informs *OCM* via the $\mathrm{RecWait}$ command.
  - Can provide one or more FITS files containing keywords and/or extensions.
  - Examples: NGC2 and CCF Technical Cameras
- Each *Data Source* may be provide 0 to any number of FITS files or keywords encoded in JSON format. FITS files are stored on a file system whereas the JSON keywords are provided to OCM via the MAL interface.

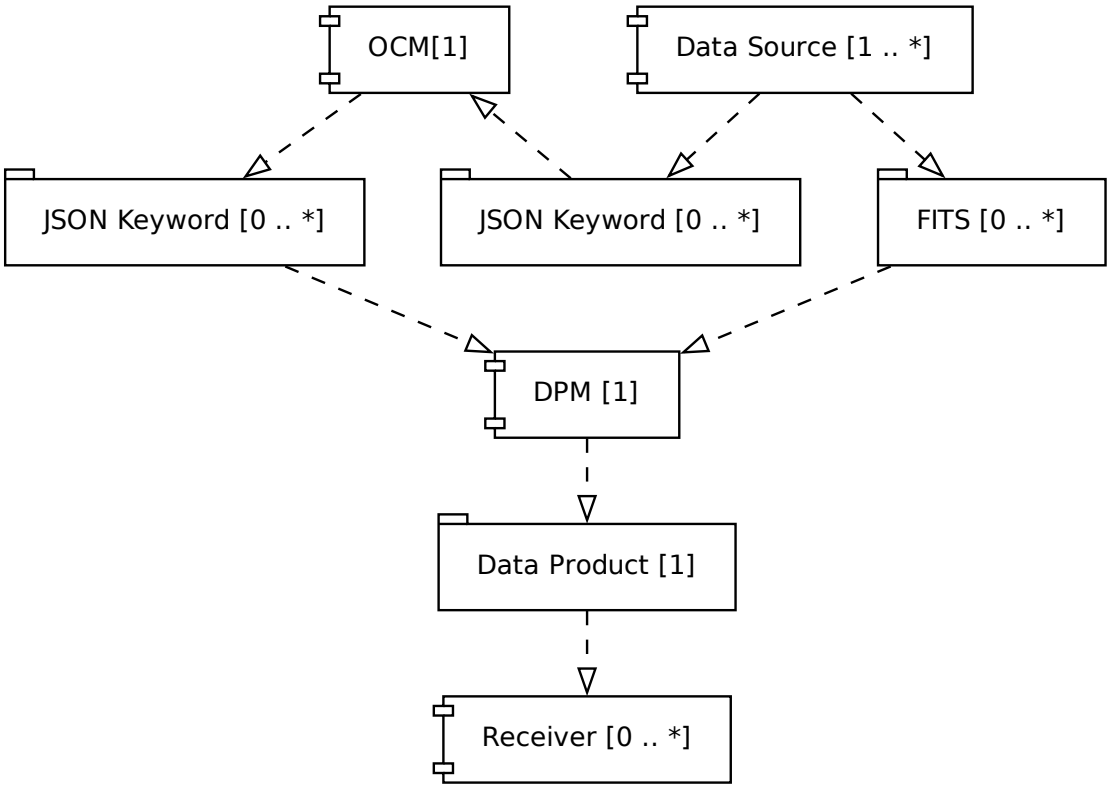


Fig. 8.2: Conceptual model of a Data Acquisition which results in a Data Product delivered to configured receivers. For sake of readability the figure does not include the details surrounding control flow. A Data Source is any component that is configured to be used for the specific Data Acquisition. Colloquially it may be referred to as a subsystem like the telescope or detector subsystem. More accurately the Data Sources is one or more processes from those subsystems that implement a supported interface for acquiring data.

- There is one instance of *DPM* handling a *Data Acquisition* that receives all JSON keywords from OCM and transfers all FITS files from the origin filesystem to where *DPM* is deployed.
- DPM then produce a single *Data Product* using input sources in a process referred to as *merging*.
- *DPM* can deliver that *Data Product* to any number of receivers using post-processing recipes. In practice this will almost always be one recipe with one receiver and that is to deliver the *Data Product* to *OLAS*.

There is a special case for OCM where it is always included as a data source implicitly. OCM will always deliver standard and user provided FITS keywords for the final *Data Product*.

For a more detailed overview see section *Data Product Creation*.

Important: All configuration that is related to a Data Acquisition is a per-Data Acquisition property. Data Acquisitions are designed to be independent to allow concurrency without surprising side-effects. This also means there is no static data source configuration in OCM. This and other parameters are provided when creating the Data Acqusition.

Note: OCM and/or DPM is not responsible for deleting *Data Products* that might no longer be useful, after post-processing. This activity falls within the scope of an operational procedure to free disk space of files after confirming they can be removed.

8.2 Control and Data Flow

The following sections provides a simplified overview of the control and data flow from *Data Acquisition* to *Data Product* delivered to the Online Archive System (OLAS)[8].

Normally it is the Sequencer that is the client when interacting with OCM, but of course any client will function the same. It requests new *Data Acquisitions* from OCM, specifying the sources to acquire data from and other parameters.

OCM coordinates the *Data Acquisition* by commanding a number of data sources such as science detectors, function controllers and telescope. In the diagram these are abstracted as the *Data Source(s)* component. When the *Data Acquisition* completes OCM commands DPM to create the *Data Product* from the acquired data.

There are no constraints on number or locality of data sources involved in a Data Acquisition. If a component implements supported interfaces correctly and is reachable over network, OCM can control it to acquire data. Refer to [*RD2*] for options when it comes to Data Acquisitions that span multiple ICSs.

[8] DPM supports per *Data Acquisition* configurable post-processing recipes, but the standard, and also default, is to interface with OLAS to archive the *Data Product*. Refer to section *Data Acquisition Process* for additional details.

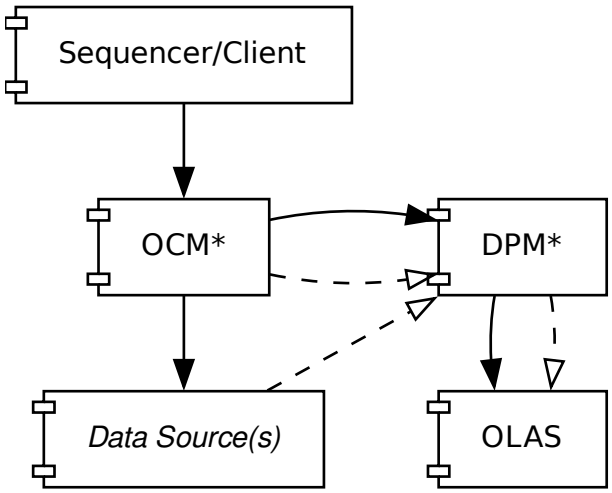


Fig. 8.3: Process overview. Components marked with * are covered by this manual.

Once the *Data Product* is complete it is delivered to the archive system OLAS or which ever post-processing recipe is configured.

8.2.1 Control Flow

This section provides an overview of the resulting control flow for individual *Data Acquisitions*. OCM supports any number of concurrent, but independent, *Data Acquisitions*. For additional details on the *Data Acquisition* process and how to control it c.f. section *Data Acquisition*.

Description of the control flow:

1. The client initiates a new *Data Acquisition*, specifying which data sources to acquire data from using the command $\mathrm{StartDaq()}$ or $\mathrm{StartDaqV2()}$. The client continues to be able to control the *Data Acquisition* using the OCM *Data Acquisition* MAL interface $\mathrm{OcmDaqControl}$.
2. OCM coordinates the *Data Acquisition* by commanding data sources to start, stop or abort, as requested by client.
3. When data has been acquired OCM commands DPM to produce a *Data Product* from a specification on how to merge the data together.
4. When DPM has created the *Data Product* it is delivered to OLAS. This done using a special purpose interface and not a normal request/reply MAL control interface.

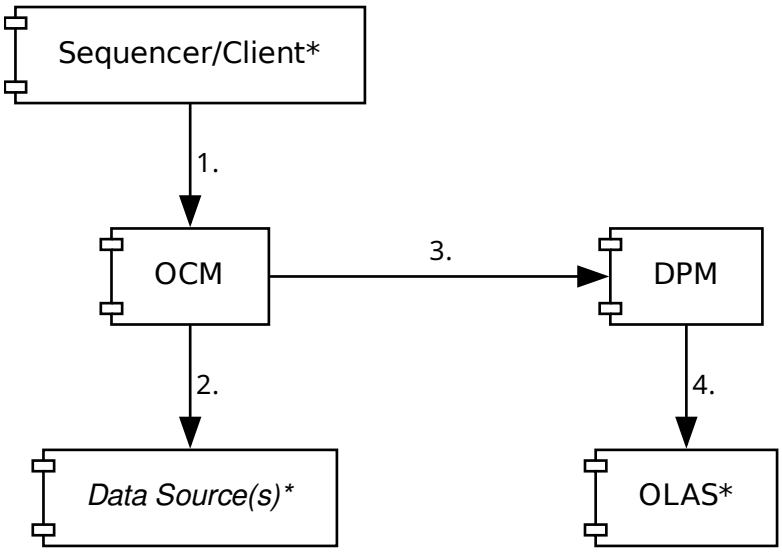


Fig. 8.4: Control flow overview.

8.2.2 Data Flow

This section provides additional details on the data flow. To give the full picture of how the *Data Products* are formed the following diagram also show how *Data Product* FITS keywords can be provided as part of the control flow from the client.

Description of the data flow:

1. When a new *Data Acquisition* is initiated FITS keywords and files can be provided at the very beginning with the $\mathrm{StartDaq}()$ or $\mathrm{StartDaqV2}()$ commands. Additionally keywords can be provided and after it has started with the $\mathrm{UpdateKeywords}()$ command.

   See *Data Validation* for how FITS keywords are validated.

2. The individual FITS files created during *Acquiring* as well as any initial FITS files are transferred by DPM to the host where it is deployed, to be merged into the final *Data Product*. This also include OCM, which provides primary HDU keywords to be merged.

   The individual FITS files follow ESO guidelines and specifications and may contain, apart from mandatory FITS keywords, also ESO hierarchical keywords and/or FITS extensions.

Note: Files are transferred *explicitly* using *rsync* if source files are not reachable on a DPM local mount. Files are transferred *implicitly* if files are located on a distributed file system, but

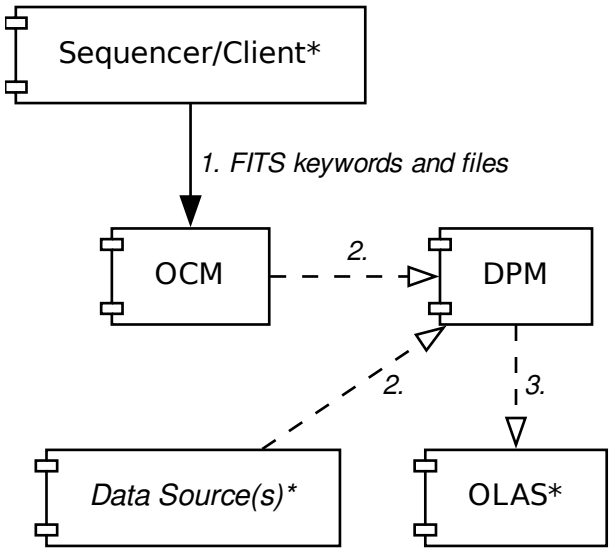


Fig. 8.5: Data flow overview.

reachable from the DPM host (i.e. reachable on a locally mounted filesystem).

3. Once *Data Product* is created by DPM it is delivered to OLAS.

   If DPM is deployed on the same file system where files are delivered to OLAS, then no additional transfer is made. If the destination file system is either different or remote, another *Data Product* file transfer is made.

8.3 Data Product Creation

This section provide an overview of how the final Data Product is created from individual files. The process is fairly simple and mechanical to reduce configuration complexity. In addition the foreseen data volumes makes complicated processing prohibitively expensive. The rule of thumb is that acquired data should be created in the desired format rather than modifying it afterwards.

Given a list of *sources* (FITS files or JSON keywords) and a *target*[9]:

1. *Sources* are provided in priority order from high to low.

[9] One of the source FITS files may be designated as the target to allow *in-place* merge, where that source file will act as the base for the subsequent sources to merge into.
If no source is designated as the target an empty FITS file will automatically be created.

The order is significant in the following ways:

- It determines the relative order of *Value Keyword*s (i.e. value keywords from first source always precede value keywords from subsequent sources).
- It determines the HDU extension order.

2. *Target* primary header is (re-)created.

   Keywords are merged to *target* primary HDU, using default or user provided keyword rules. If target also contains keywords those will be included first.

   - The keywords are taken from the primary HDU if the source is a FITS file.
   - If no keyword rules are provided all non-structure keywords are used.
   - If multiple source provide keywords with the same name the keyword from the highest priority source is kept.

3. FITS extensions from *sources* are copied to *target*.

   The extensions are copied in priority order from *sources* as they are. No modifications are done to the extensions.

Important: Merging multiple single-HDU FITS files with data is not supported. Data sources should instead be configured to produce FITS file with one or more extensions that can be appended to the same *target* file. I.e. there is no support for converting primary a HDU to an HDU extension.

New in version 3.1.0: If a FITS source file contains primary HDU data that is not merged an alert will be raised for that *Data Acquisition* by default. This can be disabled with property $\mathrm{alertUnmergeable}$ for each of json-schema-primarydatasource, json-schema-metadatasource or json-schema-fitsfilesource.

---

4. Special keywords are added or updated.

   For each HDU the FITS checksum keywords are computed and added/updated:

   - $\mathrm{ORIGFILE}$
   - $\mathrm{ARCFILE}$
   - $\mathrm{CHECKSUM}$
   - $\mathrm{DATASUM}$

User is able to specify the source order and other aspects such as keyword rules with the specification provided in $\mathrm{StartDaqV2()}$.

See also:

For details see json-schema-startdaqv2 and *Data Product Specification*.

9 The Data Acquisition Process

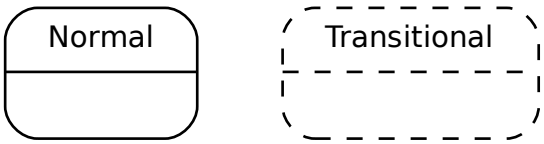
9.1 Introduction

This section provides an overview of the *Data Acquisition* process, both in terms of what is currently implemented in *OCM* and what is planned for with *DPM*.

For an overview of conceptual model, processes and the relationships between components see the section *Overview*.

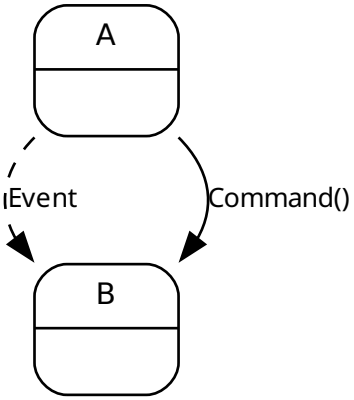
9.1.1 Stylistic Conventions

The following visual convention is used for states and transitions.



Normal
Normal state.

Transitional
Transitional states are states where the *Data Acquisition* remain during handling of a command, if it cannot be completed immediately.



Command()
Transition that occurred when initiating the handling of a command. That it is a request/command can be identified by the suffix *"()"*.

Event
Transition that occurred due to an internally generated event, internal or external. This can also occur due to completion of the handling a command and is where the reply is sent.

9.1.2 Process Overview

Each successful *Data Acquisition* includes the following states (represented in the MAL API with DaqState) which spans the following high-level activities:

StateAcquiring
From the initial point it is created in *OCM* through the phase where data actually has been acquired by data sources.

StateMerging
Through the phase where a *Data Product* is created from the acquired data and released to the Online Archive System (*OLAS*) or other configured recipients.

StateCompleted
To the final state for a *Data Acquisition*. The two possible substates are:

- Completed and
- Aborted,

which indicates a completed or user aborted *Data Acquisition* respectively.

These states, each with their own substates with an overview of transitions are visualized in Fig. 9.1.

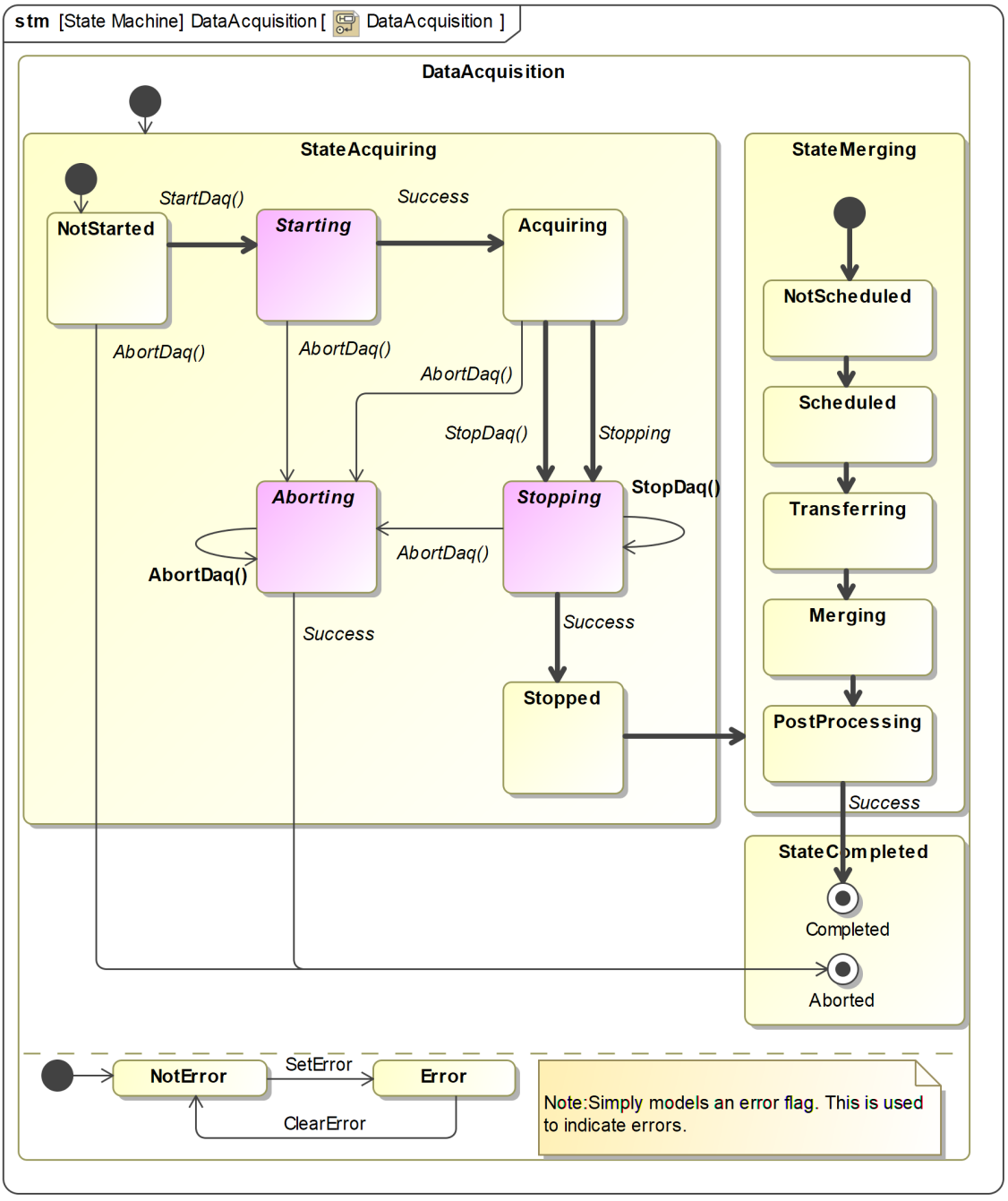


Fig. 9.1: SysML diagram of *Data Acquisition* states and transitions. Transitions in bold indicate the transition path for a successful *Data Acquisition*. Purple states are transitional states used for the duration of an operation. The orthogonal region with states NotError and Error is exclusively a means to model the error flag used in the implementation. Expect further refinement in StateMerging in following releases.

Important: The possible final states for a *Data Acquisition* is either *Completed* or *Aborted*. *OCM* and *DPM* will never abort a successfully started *Data Acquisition* on its own volition; only at the request of a client (normally the instrument operator).

The only scenario where *OCM* aborts is if StartDaq() fails, in which case the potentially *partially* started *Data Acquisition* is forcibly aborted.

Note: Error is possible in any state which is indicated with an error flag in the *Data Acquisition* status structure DaqStatus (this can be considered an orthogonal error state as shown in Fig. 9.1). This means for example that a *Data Acquisition* may be *Aborted* with or without error.

Errors may prevent forward progress in some cases and in other cases errors may be ignored by forcing forward progress with commands such as ForceStopDaq() (and accepting degraded *Data Product* as a result).

The enumeration DaqState defines the top-level states and all the sub-states are enumerated together in DaqSubState. See section *OCM Data Acquisition Control* for description of the states.

The next sections provide further details of the *Data Acquisition* life-cycle and how to interact with *OCM* for control.

9.1.3 Data Validation

Although there is no general facility for validating all input data to detect problems early, there is support for using the Data Interface Dictionaries from Data Interface Tools [*RD10*] in *daqOcmServer* to validate FITS keywords as they are received in JSON format from the following sources:

Table 9.1: Places where keyword validation is made.

Command	Direction	Description
StartDaq() and StartDaqV2()	Request	Request is rejected and *Data Acquisition* is not started.
UpdateKeywords()	Request	Request is rejected and *Data Acquisition* is not updated with any keywords.
metadaqif.MetaDaq.StopDaq()	Reply	If invalid keywords are provided as part of the reply structure DaqStopReply, from the metadaqif StopDaq() request, this is treated as an error.

Keywords are validated and formatted against configured dictionaries and if this fails the command will be rejected as a whole without starting or modifying *Data Acquisition*. Formatting is made by using the format provided in dictionary. If no format is provided the built-in standard format is used. Similarly the keyword comment is used from the dictionary if none is provided.

For configuration of dictionaries see *dictionaries*.

Note: If no dictionaries are configured no validation or formatting is made.

9.2 StateAcquiring

This section documents the details of the top-level $\mathrm{StateAcquiring}$ state of a *Data Acquisition*; the observable states, the transitions and how it relates to the *OCM* command interface described *here*.

Warning: Although the OCM API disambiguates on which *Data Acquisition* to operate on (by requiring user to provide the *Data Acquisition* identifier), the interface to detector data sources *recif* does not, and will always operate on its *current* recording whether it is correct or not. This means that users must be extra careful and e.g. make sure to only have one active *Data Acquisition* at a time which use the same primary data source.

9.2.1 Overview

The following diagram shows an overview for a nominal *Data Acquisition* where commands succeed:

Note: For brevity $\mathrm{ForceAbortDaq}()$ is not shown in the diagram as it performs the same transitions as $\mathrm{AbortDaq}()$ except that when error occcurs it still performs the transition.

The transition for the event *Stopping* is performed automatically by *OCM* when all primary data sources stop or if there are no stateful data sources[10]. If e.g. detector is configured to integrate for 10 seconds it will automatically stop, which is then observed by *OCM*, which then triggers the transition to *Stopping* state where metadata sources are being stopped.

9.2.2 Starting

The following diagram show the initial states of a *Data Acquisition* created with $\mathrm{StartDaq}()$, from the initial state *NotStarted* which is the point where the *Data Acquisition* has been created and registered internally in *OCM* but has not yet initiated any actions. When ready *OCM* transitions to state *Starting* in which all data sources are requested to start their data acquisition. When all sources acknowledge successfully the *Data Acquisition* transitions to *Acquiring*.

Note: $\mathrm{StartDaq}()$ creates a new data acquisition and starts it in one command, so the state *Not-Started* is never observed from outside. If a use-case requires it this can be changed to a two-step command, one that creates the *Data Acquisition* and one command that starts it.

[10] *OCM* can create a *Data Product* exclusively using existing FITS files for example.

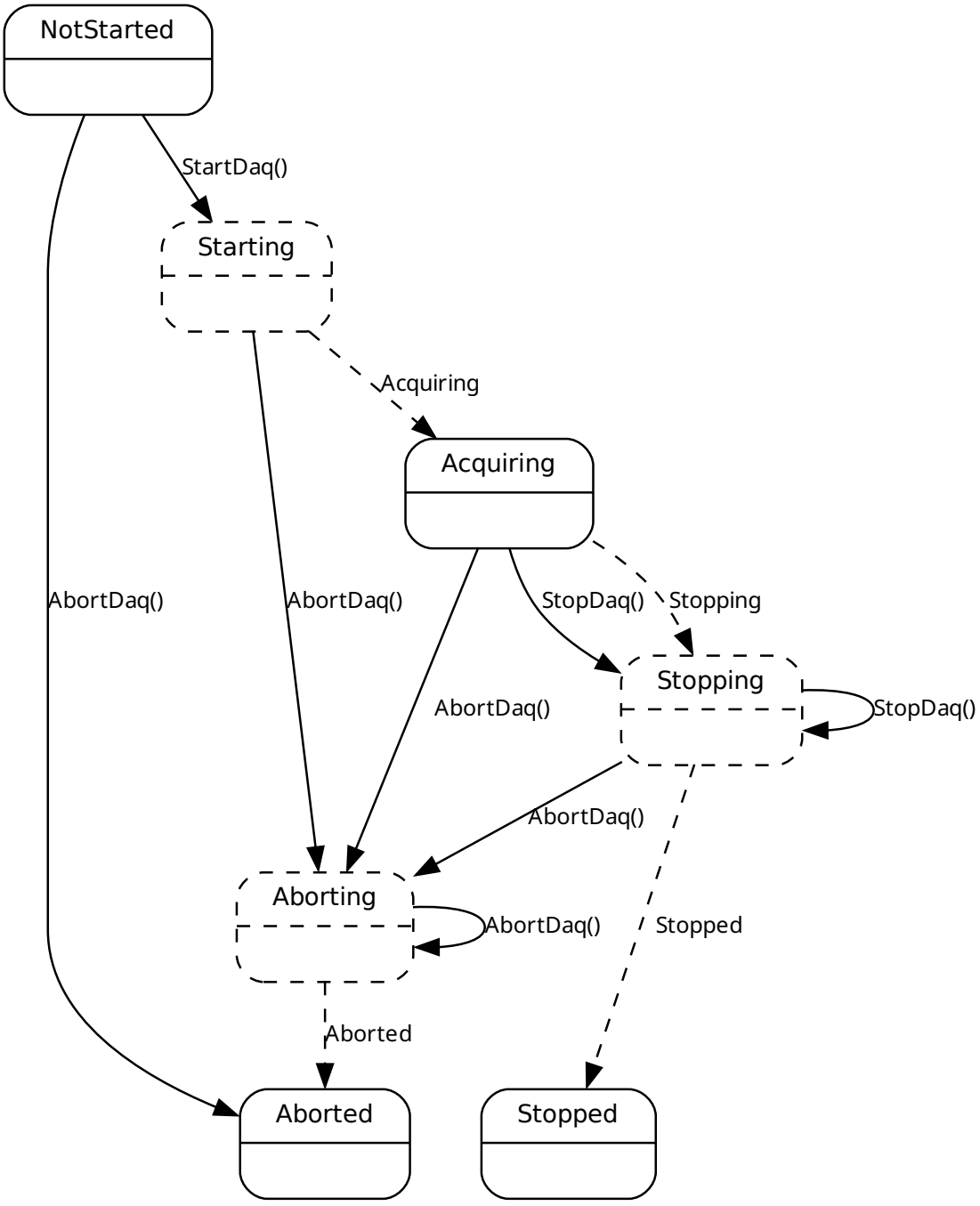
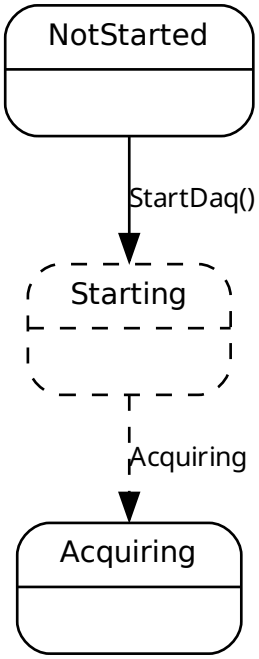


Fig. 9.2: *Data Acquisition* state and transition overview



Important: If StartDaq() does not succeed the user should clean up the failed data acquisition by aborting it. This will abort any partially started acquisitions for the configured sources. If a data source is not responding or otherwise report error, this will cause AbortDaq() to fail. For these cases ForceAbortDaq() can be used to force the transition to *Aborted*.

9.2.3 Stopping

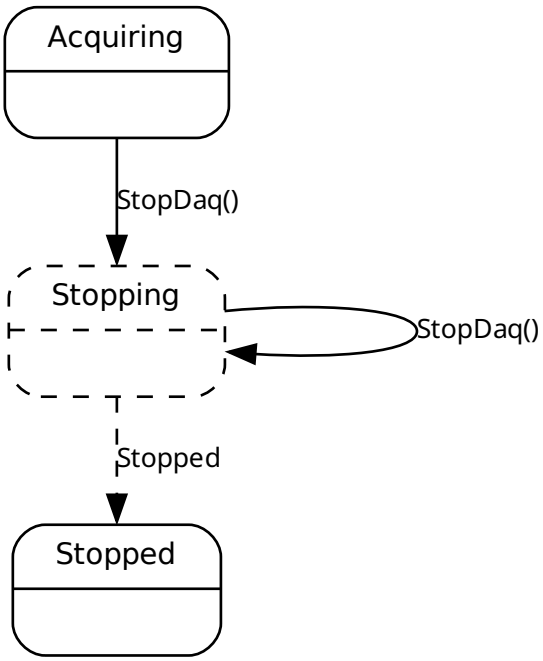
The following diagram show from which states StopDaq() and ForceStopDaq() is valid. The diagram shows only StopDaq() but is valid also for ForceStopDaq().

Using ForceStopDaq() command the only difference is that the transition to *Stopped* is forcefully performed even in the presence of errors from e.g. data sources.

Note: If StopDaq() fails the *Data Acquisition* remains in *Stopping* state. At this point it is possible to retry StopDaq() or force it with ForceStopDaq().

Important: Since ForceStopDaq() stops a *Data Acquisition* even if data sources fail to stop, it means

that user might have to perform manual error recovery on the faulty components.



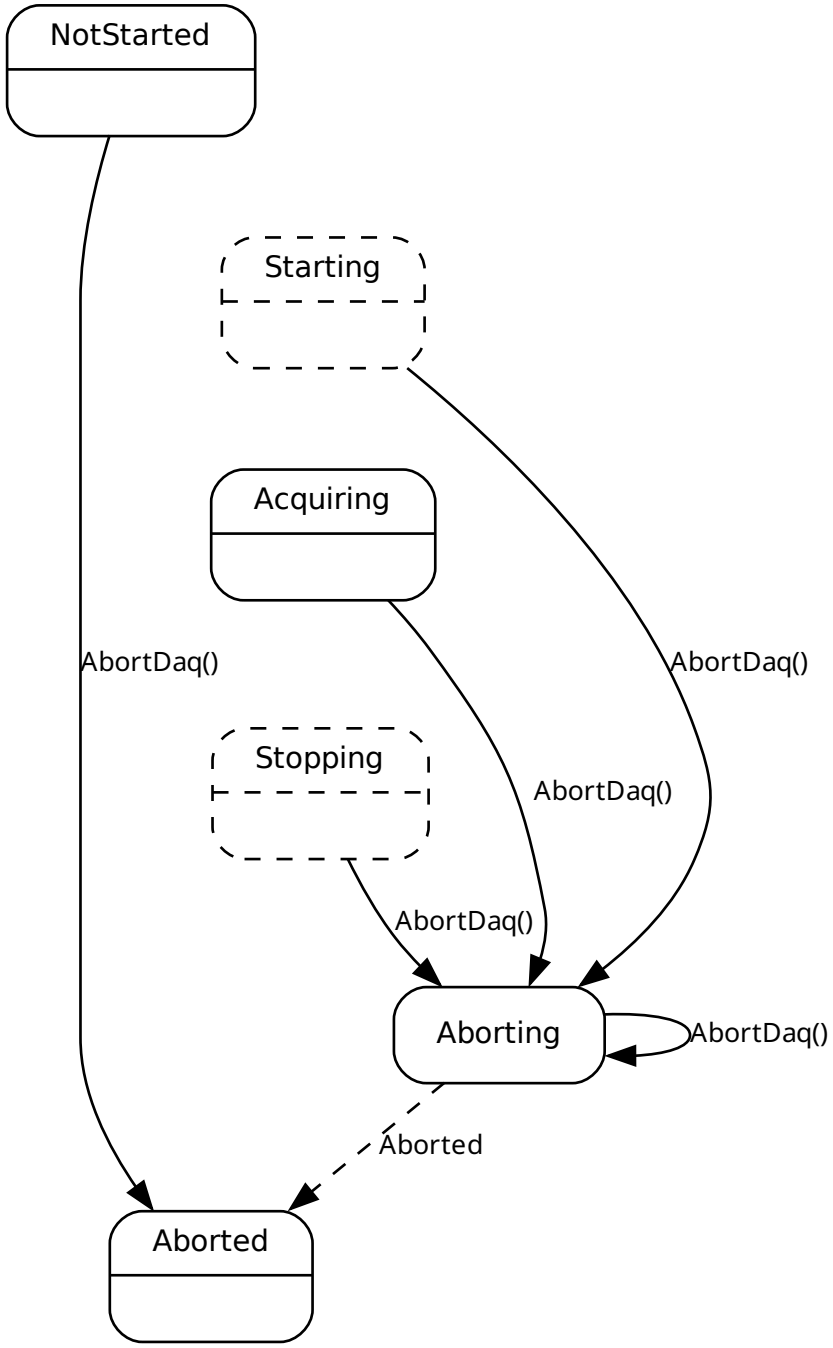
9.2.4 Aborting

The following diagram show from which states $\mathrm{AbortDaq}()$ and $\mathrm{ForceAbortDaq}()$ is valid. The diagram shows only $\mathrm{AbortDaq}()$ but is valid also for $\mathrm{ForceAbortDaq}()$.

Using $\mathrm{ForceAbortDaq}()$ command the only difference is that the transition to *Aborted* is forcefully performed even in the presence of errors from e.g. data sources.

Note: If $\mathrm{AbortDaq}()$ fails the *Data Acquisition* remains in *Aborting* state. At this point it is possible to retry $\mathrm{AbortDaq}()$ or force it with $\mathrm{ForceAbortDaq}()$.

Important: Since $\mathrm{ForceAbortDaq}()$ aborts a *Data Acquisition* even if data sources fail to abort, it means that user might have to perform manual error recovery on the faulty components.


NotStarted
Starting
Acquiring
AbortDaq()
AbortDaq()
Stopping
AbortDaq()
AbortDaq()
Aborting
AbortDaq()
Aborted
Aborted

9.3 StateMerging

This section documents the details of the top-level state $\mathrm{StateMerging}$ state of a *Data Acquisition*. Compared to $\mathrm{StateAcquiring}$ the successful sequence is always autonomous and can complete unattended. The exception to this is if the *Data Acquisition* should be aborted in which case the request $\mathrm{AbortDaq}()$ can be sent to *OCM*, which will delegate to *DPM* if required.

> If a failure occurs e.g. because of misconfiguration so that file transfer fails, *DPM* will attempt again next time it is started. As a last resort manual recovery can be attempted.

Note: It is worth clarifying that clients are not required and are never expected to have to interact directly with *DPM*.

9.3.1 Overview

9.3.2 NotScheduled

OCM will attempt to schedule *Data Acquisition* for merging. If *DPM* is offline or otherwise unreachable it will remain in this state.

As *Data Acquisition* has not yet been scheduled it is possible to abort the *Data Acquisition* without a connection to *daqDpmServer*.

9.3.3 Scheduled

Responsibility for completing the *Data Acquisition* is from this point on *DPM* and authoratitive *Data Acquisition* status originates from *DPM*, but still published by *OCM*.

If a request to abort *Data Acquisition* is made the normal behaviour is to forward the request to *DPM*. If *DPM* is offline the *Data Acquisition* can only be aborted with $\mathrm{ForceAbortDaq}()$, but this will be unknown to *DPM*:

Warning: There is a risk of *Data Acquisition* state inconsistency if *Data Acquisition* is forcibly aborted. As *DPM* is offline or unreachable it may independently of *OCM* complete the merge process. As such it is possible the *Data Acquisition* status is inconsistent or may change after new information from *DPM* is available again.

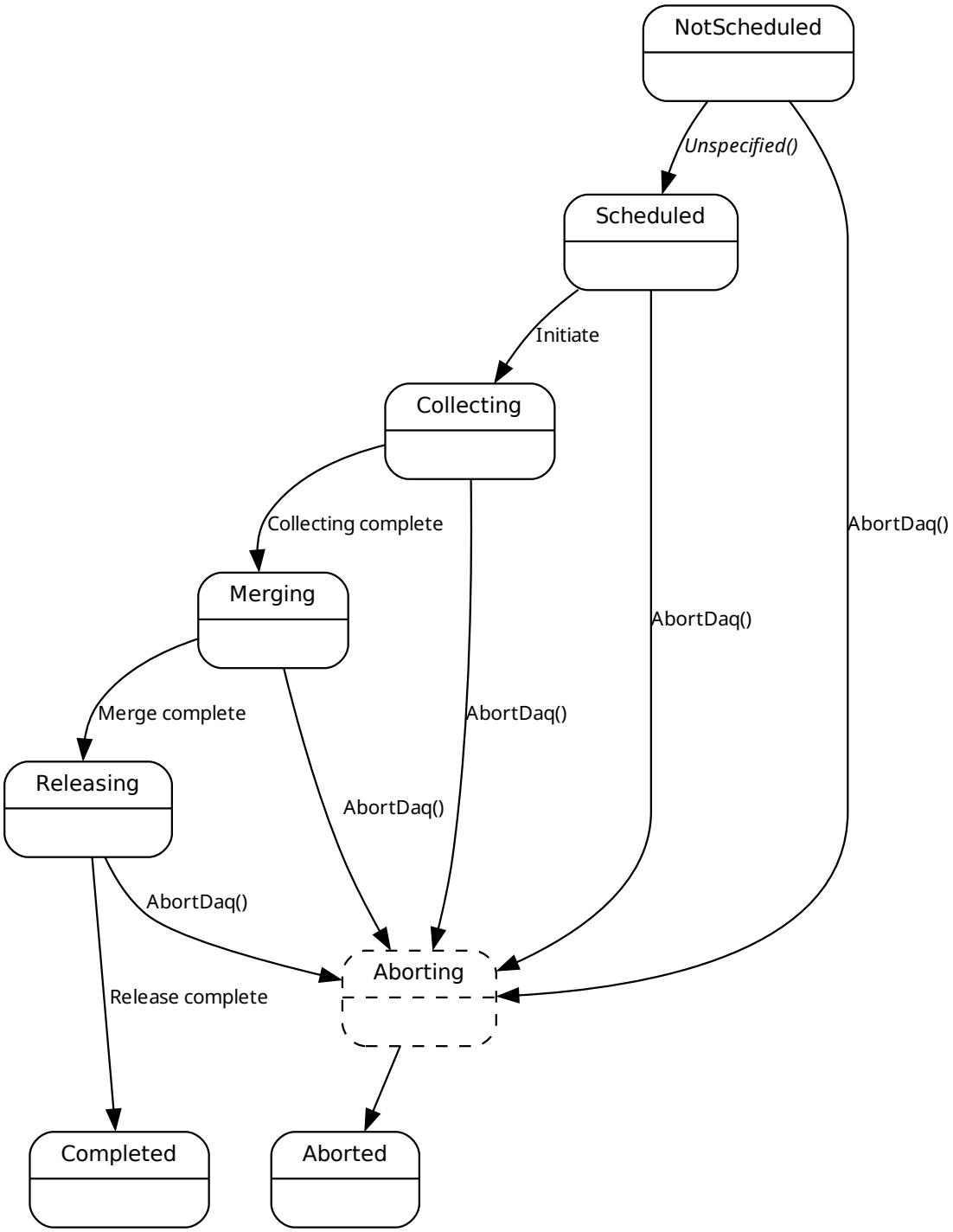


Fig. 9.3: *Data Acquisition* state and transition overview for StateMerging

9.3.4 Collecting

Files are collected from where they were created to the local *daqDpmServer workspace*. At this time there is no optimization implemented for the case the file is available from local file system mount.

9.3.5 Merging

This is the state where the final *Data Product* is created from all the previously acquired data. An overview of that process is provided in *Data Product Creation*.

9.3.6 Releasing

The completed *Data Product* is released to configured receivers (c.f. $\mathrm{receivers}$ in json-schema-startdaqv2specification). If a transfer fails this is not treated as a fatal error and does not prevent the completion of the *Data Acquisition*. Any such failure will result in an alert and the *Data Acquisition* error flag is set in $\mathrm{DaqStatus.error}$.

Note: If *daqDpmServer* is deployed on the same host as *OLAS* and host is empty, *daqDpmServer* will try to create a hard link of the *Data Product* to the json-schema-olas-receiver path. If this fails then a symlink will be created instead - if this also fails a copy will be attempted.

9.3.7 Completed

This is the end of the *Data Acquisition* life-cycle, no activities are performed in this state.

10 Data Acquisition Guide

This guide demonstrates different *Data Acquisition* use-cases with focus on the interaction with *OCM* to act as a guide when e.g. writing Sequencer templates/scripts. The provided examples use the same simulators used in *daqOcmServer* and *daqDpmServer* integration tests. These provide limited simulation capabilities but is useful in this case since they are easy to deploy.

For details on more realistic deployment please refer to section *Deployment* and sections for *daqOcmServer* and *daqDpmServer*.

Note: Although *daqDpmServer* is required to produce the final *Data Product* it is not a software component the end user interacts with directly, rather it is *daqOcmServer* that commands the *daqOcmServer*. As such the guide does not contain any examples of how to interact with *daqDpmServer*, other than how to start and stop it.

10.1 Prerequisites

The guide below assumes the runtime environment has been configured and *daqOcmServer* and optionally *daqDpmServer* has been deployed and is running. This section provide the minimal number of steps to achieve that.

The following standard ICS Framework environment variables are expected to be defined:

$DATAROOT
: Will be used by default by *daqOcmServer* and *daqDpmServer* for their respective workspaces:

  - $DATAROOT/ocm
  - $DATAROOT/dpm

$CFGPATH
: Will be used to resolve configuration paths (see *Config Path*). To use configuration files when building from source add the path $PREFIX/resource to $CFGPATH, where $PREFIX is the installation prefix.

The examples further assume the following environment variables are defined. The endpoint URIs should reflect what is in your configuration, and using the example configuration above it would be:

- OCM_REQUEST_EP=zpb.rr://127.0.0.1:12081 (replaces option --rep in *daqOcmCtl*)
- OCM_PUBLISH_EP=zpb.ps://127.0.0.1:12082 (replaces option --pep in *daqOcmCtl*)
- DCS_REQUEST_EP=zpb.rr://127.0.0.1:12090 (DCS simulator request endpoint)
- DCS_DAQ_EP=${DCS_REQUEST_EP}/rec (DCS service endpoint)
- FCF_REQUEST_EP=zpb.rr://127.0.0.1:12091 (FCF simulator request endpoint)
- FCF_DAQ_EP=${FCF_REQUEST_EP}/daq (FCF service endpoint)

Or as console commands:

```
$ export OCM_REQUEST_EP=zpb.rr://127.0.0.1:12081
$ export OCM_PUBLISH_EP=zpb.ps://127.0.0.1:12082
$ export DCS_REQUEST_EP=zpb.rr://127.0.0.1:12090
$ export DCS_DAQ_EP=$DCS_REQUEST_EP/rec
$ export FCF_REQUEST_EP=zpb.rr://127.0.0.1:12091
$ export FCF_DAQ_EP=$FCF_REQUEST_EP/daq
```

Note: Examples in this guide use the command line client *daqOcmCtl* provided with the ifw-daq project. To facilitate use from scripts *daqOcmCtl* can provide return values in JSON format with the --json option. Alternatively the MAL API daqif.OcmDaqControl may be used directly from e.g. Python scripts.

Important: Many examples in this guide include references to components with names such as "fcf1", "fcf2", or "dcs1" with MAL URIs such as "zpb.rr://insws:12345/daq". These components are assumed to be deployed and operational. This means that the examples must be modified to suit your environment to instead reference your components.

10.2 Lifecycle Control

10.2.1 Startup

For testing purposes the default *daqOcmServer* sample configuration provided by the ifw-daq project may be used (config/daqOcmServer/config.yaml)

The configuration is

```
cfg:
    instrument_id: "UNDEF"
    req_endpoint: "zpb.rr://127.0.0.1:12081/" # IP address and port used to accept requests
    pub_endpoint: "zpb.ps://127.0.0.1:12082/" # IP address and port used to accept requests
    oldb_conn_timeout: 2 # timeout in seconds when connecting to runtime DB
    sm_scxml: "config/daqOcmServer/sm.xml"
    log_properties: "config/daqOcmServer/log.properties"

    # DPM communication configuration
    dpm:
        req_endpoint: "zpb.rr://127.0.0.1:12083/"
        pub_endpoint: "zpb.ps://127.0.0.1:12084/"
```

In which case the following steps can be performed to start *daqOcmServer* and simulators used for

demonstration purposes.

Start the servers using default configuration and export environment variables to simplify *daqOcmCtl* interaction. The configuration must be found as a *Config Path* using $CFGPATH environment variable.

```
$ daqOcmServer -l DEBUG &
$ daqDpmServer -l DEBUG &
```

Additionally we start the simulators we can acquire data from:

```
$ daqSimMetadaqif fcf $FCF_REQUEST_EP &
```

The simulated DCS should be deployed in foreground mode to allow control when recording should stop. This is done by sending a line break using Enter to *stdin* after recording has started.

```
$ daqSimRecif -v dcs $DCS_REQUEST_EP
```

Once started we bring *daqOcmServer* operational with:

```
$ daqOcmCtl std.init
"OK"
$ daqOcmCtl std.enable
"OK"
```

Note: To use a custom configuration create a new configuration file using with the help of sections *Configuration File* and *Configuration File*. Then specify the configuration file when starting *daqOcmServer* and *daqDpmServer* with the --config PATH argument.

10.2.2 Shutdown

To shut *daqOcmServer* down the signal SIGINT (Ctrl-c) or the command Exit() can be sent using e.g. the daqOcmCtl application:

```
$ daqOcmCtl std.exit
```

To shut *daqDpmServer* and the simulators down send signal SIGINT (Ctrl-c).

10.3 Observing Status Changes

To observe the published state changes in *OCM* it is possible to use daqOcmCtl without a command with the --status option, in which case it will subscribe and remain running until stopped with Ctrl-c while printing any received topic samples to stderr. This is an example of the output during a *Data Acquisition*:

```
$ daqOcmCtl --status
no command provided -> will subscribe indefinitely
status: Operational;Active
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateAcquiring, substate=NotStarted, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateAcquiring, substate=Starting, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateAcquiring, substate=Acquiring, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateAcquiring, substate=Stopping, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateAcquiring, substate=Stopped, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateMerging, substate=NotScheduled, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateMerging, substate=Scheduled, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateMerging, substate=Collecting, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateMerging, substate=Merging, error=false, message=[]
daq: id=UNDEF.2023-11-22T10:36:32.078, file_id=UNDEF.2023-11-22T10:36:32.078, \
state=StateMerging, substate=Releasing, error=false, message=[]
status: Operational;Idle
```

Note: The status is published by *OCM* for any change, some of which are not caused by state transitions. This may lead to the same status line being repeated multiple times.

The --status option can be passed when executing a command as well, but often the commands complete before any topic samples are received. The exception to this is the daq.awaitstate which only completes when the specified state is achieved or operation times out.

10.4 Automatic Stop Sequence

Automatic stop is the typical scenario when a *Data Acquisition* includes one or more primary data sources (usually detectors) that are configured with a fixed integration time. When all primary data sources stop (also referred to as *Completed* or *Stopped* in recif) as observed by *OCM*, which monitors all the sources, *OCM* will proceed and stop all metadata sources automatically. This condition can then be awaited on to then trigger other activities such as configuring the instrument for the next *Data Acquisition*.

Note: Multiple sources in each category are provided by space-separating each source, e.g:

```
"fcf1@zpb.rr://insws1:12345/daq fcf2@zpb.rr://insws2:12345/daq"
```

Data Acquisition is started with the daq.start command which takes two[11] space separated lists of sources or daq.startv2 which takes a JSON-encoded specification of the *Data Acquisition* to perform. The second form is necessary if additional control is required.

The following example starts a new *Data Acquisition* with a single primary source named *dcs1* and a single metadata source named *fcf1*. The names are only used to give a friendly name to a possibly anonymous URI and is so far not used for anything but logging activities.

```
$ daqOcmCtl --json daq.start "dcs1@${DCS_DAQ_EP}" "fcf1@${FCF_DAQ_EP}"
{
  "id": "UNDEF.2023-11-22T12:33:53.856",
  "error": false
}
```

Using the request daq.startv2 (StartDaqV2()) the following yields same results. To simplify the usage the JSON specification is read from *stdin* as specified with @- and then provided using a bash *Here Document*:

```
$ daqOcmCtl daq.startv2 @- <<EOF
{
  "sources": [
   {
     "type": "primarySource",
     "sourceName": "dcs1",
     "rrUri": "${DCS_DAQ_EP}"
   },
   {
     "type": "metadataSource",
     "sourceName": "fcf1",
     "rrUri": "${FCF_DAQ_EP}"
```

(continues on next page)

[11] First list specify the primary sources and second list the metadata sources.

(continued from previous page)

```
    }
 ],
 "mergeTarget": {
   "sourceName": "dcs1"
 }
}
EOF

{
  "id": "UNDEF.2023-11-22T12:33:53.856",
  "error": false
}
```

Since the DCS simulator will run until we ask it to stop the *Data Acquisition* will remain in state daqif.DaqState.StateAcquiring and substate daqif.DaqSubState.Acquiring.

We can check the list of active *Data Acquisitions* using daq.getactivelist. We expect a single *Data Acquisition* at this point:

```
$ daqOcmCtl --json daq.getactivelist
[
  {
    "error": false,
    "fileId": "UNDEF.2023-11-22T12:33:53.856",
    "id": "UNDEF.2023-11-22T12:33:53.856",
    "message": "[]",
    "result": "",
    "state": "StateAcquiring",
    "substate": "Acquiring",
    "timestamp": 1700656433.8668084
  }
]
```

Data Acquisition status can be checked with the daq.getstatus command by providing the *Data Acquisition* id from the start command:

```
$ daqOcmCtl --json daq.getstatus UNDEF.2023-11-22T12:33:53.856
{
  "error": false,
  "fileId": "UNDEF.2023-11-22T12:33:53.856",
  "id": "UNDEF.2023-11-22T12:33:53.856",
  "message": "[]",
  "result": "",
  "state": "StateAcquiring",
```

(continues on next page)

(continued from previous page)


```
  "substate": "Acquiring",
  "timestamp": 1700656433.8668084
}
```


When *dcs1* completes *OCM* will issue the equivalent of the StopDaq command to stop all metadata sources. To await the completion of all FITS files the daq.awaitstate command is issued with states daqif.DaqState.StateAcquiring / daqif.DaqSubstate.Stopped and then in the terminal running daqSimRecif simulating DCS hit Enter to simulate that it completed automatically. If performed within the timeout the result would look similar to:


```
$ daqOcmCtl --json daq.awaitstate UNDEF.2023-11-22T12:33:53.856 StateAcquiring Stopped 60
Note: Setting request timeout to 62s due to await timeout exceeding request timeout
{
  "status": {
    "error": false,
    "fileId": "UNDEF.2023-11-22T12:33:53.856",
    "id": "UNDEF.2023-11-22T12:33:53.856",
    "message": "[]",
    "result": "",
    "state": "StateAcquiring",
    "substate": "Stopped",
    "timestamp": 1700657163.9278564
  },
  "timeout": false
}
```


At this point the data sources are finished with their contribution and the merging process is started if *daqOcmServer* can communicate with *daqDpmServer*. Since the simulated files are small it is likely already complete.

Checking the status again reports *Data Acquisition* is in state daqif.DaqState.StateCompleted and where the merged file is located:


```
$ daqOcmCtl --json daq.getstatus UNDEF.2023-11-22T12:33:53.856
{
  "error": false,
  "fileId": "UNDEF.2023-11-22T12:33:53.856",
  "id": "UNDEF.2023-11-22T12:33:53.856",
  "message": "[]",
  "result": "/var/run/dataroot/dpm/result/UNDEF.2023-11-22T12:33:53.856.fits",
  "state": "StateCompleted",
  "substate": "Completed",
  "timestamp": 1700657164.0261965
}
```

10.5 Manual Stop Sequence

Manual stop is used either when there is no fixed integration time on all primary data sources or when no primary sources are used at all (in which case there is nothing to inform *OCM* when to stop except for the user).

In this example data is acquired from *fcf1* only, so user must decide when to stop manually.

```
$ daqOcmCtl --json daq.start "" "fcf1@${FCF_DAQ_EP}"
{
  "id": "TEST.2021-03-09T18:48:05.967",
  "error": false
}
```

Or using daq.startv2

```
$ daqOcmCtl --json daq.startv2 @- <<EOF
{
  "sources": [
    {
      "type": "metadataSource",
      "sourceName": "fcf1",
      "rrUri": "${FCF_DAQ_EP}"
    }
 ]
}
EOF

{
  "id": "UNDEF.2023-11-22T12:56:47.200",
  "error": false
}
```

At a later point the *Data Acquisition* can be stopped using daq.stop by specifying the id returned by daq.start.

```
$ daqOcmCtl --json daq.stop UNDEF.2023-11-22T12:56:47.200
{
  "id": "UNDEF.2023-11-22T12:56:47.200",
  "error": false
}
```

If we check status we see that it is indeed stopped and likely completed if *daqOcmServer* can reach *daqDpmServer*:

```
$ daqOcmCtl --json daq.getstatus UNDEF.2023-11-22T12:56:47.200
```

(continues on next page)

(continued from previous page)


```
{
  "error": false,
  "fileId": "UNDEF.2023-11-22T12:56:47.200",
  "id": "UNDEF.2023-11-22T12:56:47.200",
  "message": "[]",
  "result": "/var/run/dataroot/dpm/result/UNDEF.2023-11-22T12:56:47.200.fits",
  "state": "StateCompleted",
  "substate": "Completed",
  "timestamp": 1700657840.7738013
}
```

11 Observation Coordination Manager

11.1 Introduction

Fig. 11.1: Package and main components overview

The package OCM provides the application components *daqOcmServer* and *daqOcmCtl*.

daqOcmServer implements the following MAL request/reply interfaces:

MAL Interface	URI Path	Description
stdif.StdCmds	/std	Standard ICS interface for state control and supervision.
daqif.OcmDaqControl	/daq	Data Acquisition control interface. This interface is enabled when OCM enters *Operational* state and is disabled when OCM enters *NotOperational*.

OCM can control components implementing the following MAL ICDs:

MAL Interface	Description
metadaqif[12]	Standard interface and assumed interface for Data Acquisition metadata sources.
recif[13]	Standard interface used by ELT detector control software and is the assumed interface for Data Acquisition primary data sources.

Caution:

The standard interface *recif* does not support concurrent recordings or in the following commands *which* recording to operate on; it always apply to the *current* recording:

- recif.RecCmds.RecStop (used with command StopDaq)

[12] https://gitlab.eso.org/ecs/ecs-interfaces/
[13] https://gitlab.eso.org/ecs/ecs-interfaces/

- recif.RecCmds.RecAbort (used with command AbortDaq)
- recif.RecCmds.RecWait (used with command StopDaq)

Since the commands always apply to the *current* recording, it means that if system state consistency deviates for any reason, it may lead to undesirable consequences such as aborting the wrong Data Acquisition.

11.2 Client

The provided command line client *daqOcmCtl* can interact with *daqOcmServer* which is described in the following sections.

11.2.1 Environment Variables

$OCM_REQUEST_EP
: Specifies the default OCM request/reply endpoint, e.g. zpb.rr://127.0.0.1:12345/.

$OCM_PUBLISH_EP
: Specifies the default OCM publish endpoint, e.g. zpb.ps://127.0.0.1:12345/.

$DAQ_LOGS
: Optionally specifies path where to create separate log file for *daqOcmServer*.

  New in version 3.1.0.

11.2.2 Command Line Arguments

Exhaustive command line help is available under the option --help. The following list enumerates a subset of common commands.

Synopsis:

```
daqOcmCtl [options] <command> [options] <command-args>...
```

Standard interface commands:

std.init
: Sends the Init() command.

std.enable
: Sends the Enable() command.

std.disable
: Sends the Disable() command.

std.exit
: Sends the Exit() command.

std.setloglevel <logger> <level>
: Sends the SetLogLevel() command with provided logger and level.

std.getstate
: Sends the GetState() command.

std.getstatus
: Sends the GetStatus() command.

std.getversion
: Sends the GetVersion() command.

Data Acquisition commands:

daq.start [options] <primary-sources> <metadata-sources>
: Sends the StartDaq() command with provided arguments.

daq.startv2 [options] <specification>
: Sends the StartDaqV2() command with provided specification.

  It is possible to read JSON from a file by prefixing path with @, e.g. @start.json.

  To read from stdin the - convention is supported (/dev/stdin works as well) which can be used with bash heredocs, e.g:

```
$ daqOcmCtl --json cmd.startv2 @- <<EOF
   {
    "sources": [
     {
      "type": "primarySource",
      "sourceName": "dcs",
      "rrUri": "zpb://10.127.50.10:4050/RecCmds"
     },
     {
      "type": "metadataSource",
      "sourceName": "tcs",
      "rrUri": "zpb://10.127.50.15:5011/daq"
     }
    ]
   }
EOF

{
  "id": "TEST.2023-02-21T17:26:46.440",
  "error": false
}
```

daq.stop [options] <id>
: Sends the StopDaq() command with provided arguments.

daq.forcestop [options] <id>
: Sends the ForceStopDaq() command with provided arguments.

daq.abort [options] <id>
: Sends the AbortDaq() command with provided arguments.

daq.forceabort [options] <id>
: Sends the ForceAbortDaq() command with provided arguments.

daq.getstatus <id>
: Sends the GetDaqStatus() command to query status of *Data Acquisition* identified by <id>.

daq.awaitstate <id> <state> <substate> <timeout>
: Sends the AwaitDaqState() command with provided arguments.

  <id>
  : *Data Acquisition* identifier.

  <state>
  : *Data Acquisition* state to await.

  <substate>
  : *Data Acquisition* state to await.

  <timeout
  : Time in seconds to wait for state to be reached or unable to be reached anymore.

daq.updatekeywords <id> <keywords>
: Sends the UpdateKeywords() command with provided arguments.

daq.getactivelist
: Sends the GetActiveList() command.

11.3 Server

The main OCM application is *daqOcmServer*, which implements all the *Data Acquisition* control and coordination features. The interface to control *Data Acquisitions* is covered in section *The Data Acquisition Process* whereas the much simpler application state control is described in this section.

Changed in version 2.0.0: *daqOcmServer* interacts with *daqDpmServer* to execute the merge process to create the final *Data Product*. *daqOcmServer* Stores and loads relevant *Data Acquisition* state in its *Workspace* to be able to continue after application restart.

11.3.1 State Machine

The *daqOcmServer* state machine is shown in Fig. 11.2 with states and transitions described below.

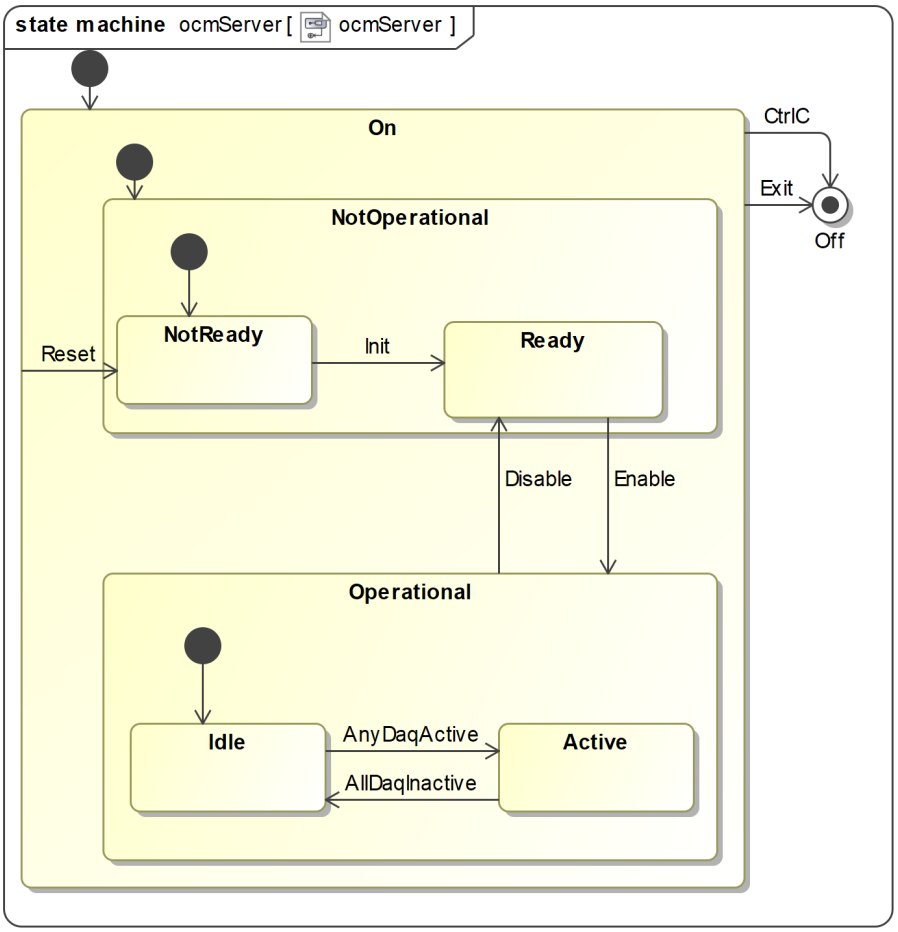


Fig. 11.2: Application statemachine implemented by *daqOcmServer* which satisfies the state machine expected by stdif.

States

On
: Application is running.

Off
: Application is not running.

NotOperational
: Composite state that means that *daqOcmServer* is running, is able to accept StdCmds requests, but is not yet fully operational. For *daqOcmServer* it means in particular that the OcmDaqControl interface is not registered and won't accept any requests.

NotReady

This is the first non-transitional state. Current implementation has already loaded configuration and has registered the stdif.StdCmds interface at this point.

Ready
Has no particular meaning for *daqOcmServer*.

Operational
In the transition to Operational *daqOcmServer* registered the OcmDaqControl interface and is ready to perform *Data Acquisitions*.

Idle
Indicates that there are no active *Data Acquisitions*.

Active
Indicates that there is at least one active *Data Acquisition*.

Note: Active does not mean that *daqOcmServer* is busy and cannot handle additional requests. It simply means that there is at least one *Data Acquisition* is not yet finished.

Since merging is not yet implemented the definition of active is up to the point the *Data Acquisition* is stopped or aborted.

Transitions

Init
Triggered by Init() request.

Enable
Triggered by Enable() request.

Disable
Triggered by Disable() request.

Stop
Triggered by Stop() request.

Note: The behaviour is currently unspecified if this request is issued if *OCM* is in state Active.

AnyDaqActive
Internal event that is created when any *Data Acquisition* becomes active.

AllDaqInactive
Internal event that is created when all *Data Acquisitions* are inactive.

11.3.2 MAL URI Paths

The following tables summarize the request/reply service paths and topic paths for pub/sub.

Table 11.1: Data Acquisition Control URI paths.

URI Path	Root URI Configuration	Description
/std	cfg/req_endpoint	Standard control interface stdif.StdCmds.
/daq	cfg/req_endpoint	*Data Acquisition* control interface daqif.OcmDaqControl.

Table 11.2: Topic URI paths.

Topic Type	URI Path	Root URI Configuration	Description
stdif.Status	/std/status	cfg/pub_endpoint	Standard interface status topic providing information on OCM overall state. Same information is provided with the command GetStatus().
daqif.DaqStatus	/daq/status	cfg/pub_endpoint	*Data Acquisition* status topic daqif.OcmDaqControl. Same information is provided with the command GetDaqStatus().

11.3.3 Command Line Arguments

Command line argument help is available under the option --help.

--proc-name ARG| -n ARG **(string) [default: *ocm*]**
Process instance name.

--config ARG| -c ARG **(string) [default: *config/daqOcmServer/config.yaml*]**
Config Path to application configuration file e.g. --config ocs/ocm.yaml (see *Configuration File* for configuration file content).

--log-level ARG| -l ARG **(enum) [default: *INFO*]**
Log level to use. One of ERROR, WARNING, STATE, EVENT, ACTION, INFO, DEBUG, TRACE.

--db-host ARG| -d ARG **(string) [default: *127.0.0.1:6379*]**
Redis database host address.

11.3.4 Environment Variables

$CFGPATH
: Used to resolve *Config Path* configuration file paths.

$DATAROOT
: Specifies the default root path used as output directory for e.g. OCM FITS files and other state storage. The data root can be overridden by the configuration key cfg/dataroot.

11.3.5 Configuration File

This section describes what the configuration file parameters are and how to set them.

The configuration file is currently based on YAML and should be installed to one of the paths specified in $CFGPATH where it can be loaded using the *Config Path* and the command line argument --config ARG.

If a configuration parameter can be provided via command line, configuration file and environment variable the precedence order (high to low priority) is:

1. Command line value
2. Configuration file value
3. Environment variable value

Enumeration of parameters:

cfg/instrument_id **(string)**
: ESO designated instrument ID. This value is also used as the source for FITS keyword *INSTRUME*.

cfg/dataroot **(string) [default: *$DATAROOT*]**
: Absolute path to a writable directory where OCM will store files persistently. These are mainly FITS files produced as part of a *Data Acquisition*. If directory does not exist OCM will attempt to create it, including parent directories, and set permissions to 0774 (ug+rwx o+r).

cfg/daq/workspace **(string) [default: *{process name}*]**
: Workspace used by *daqOcmServer* to store *Data Acquisition* state persistently and later restore that state when starting up (see section *Workspace* for details). Default value is to use the process name.

  - Absolute paths are used as is (*recommended*).
  - Relative paths are defined relative to cfg/dataroot.

  New in version 2.0.0.

cfg/daq/stale_acquiring_hours **(integer) [default: *14*]**
: Parameter used to control when to archive (discard) stale *Data Acquisitions* from *Workspace* during startup.

Specifically it controls when a *Data Acquisition* in state daqif.State.StateAcquiring is automatically archived when recovered from *Workspace* because it is considered stale (time duration from time of creation to the time it is recovered).

New in version 2.0.0.

cfg/daq/stale_merging_hours **(integer) [default: *48*]**
Parameter used to control when to archive (discard) stale *Data Acquisitions* from *Workspace* during startup.

Specifically it controls when a *Data Acquisition* in state daqif.State.StateMerging is automatically archived when recovered from *Workspace* because it is considered stale (time duration from time of creation to the time it is recovered).

New in version 2.0.0.

cfg/log_properties **(string)**
Config Path to a log4cplus log configuration file. See also logging-configuration for important limitations.

cfg/sm_scxml **(string) [default: *config/daqOcmServer/sm.xml*]**
Config Path to the SCXML model. This should be left to the default which is provided during installation of *daqOcmServer*.

cfg/req_endpoint **(string) [default: *zpb.rr://127.0.0.1:12081/*]**
MAL server request root endpoint on which to accept requests. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

cfg/pub_endpoint **(string) [default: *zpb.ps://127.0.0.1:12082/*]**
MAL server publish root endpoint on which to publish topics from. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

cfg/oldb_uri_prefix **(string) [default: *cii.oldb:/elt/{process-name}*]**
Optional CII URI OLDB prefix that is prepended to all database keys in the form cii.oldb:/{path}. By default the process instance name is used as path element.

Example: "cii.oldb:/elt/instrument-name/ocm"

New in version 2.0.0.

cfg/dictionaries **(List[string]) [default: *[]*]**
List of FITS keyword dictionaries to load as paths relative to $CFGPATH.

Empty list disables keyword validation.

Example: ["dictionary/dit/stddid/primary.did.yaml"]

Warning: CII config service used to load dictionaries prevents use of absolute paths so paths relative to $CFGPATH must be used.

See also:

See *Data Validation* for description of functionality this enables.

New in version 3.1.0.

cfg/oldb_conn_timeout **(integer) [default: *2*]**
Timeout in seconds to use when communicating with the CII OLDB server.

cfg/dpm/req_endpoint **(string)**
daqDpmServer request endpoint without service name.

Example: "zpb.rr://127.0.0.1:12345".

New in version 2.0.0.

cfg/dpm/pub_endpoint **(string)**
daqDpmServer publish endpoint without service name.

Example: "zpb.ps://127.0.0.1:12345".

New in version 2.0.0.

cfg/dpm/timeout_sec **(integer) [default: *5*]**
MAL timeout used when sending requests to *daqDpmServer*.

New in version 2.0.0.

Full example:

```
cfg:
    instrument_id: "TEST"
    dataroot: "/absolute/output/path"
    sm_scxml: "config/daqOcmServer/sm.xml"
    req_endpoint: "zpb.rr://127.0.0.1:12340/"
    pub_endpoint: "zpb.ps://127.0.0.1:12341/"
    db_timeout_sec: 2
    log_properties: "log.properties"

    daq:
        # Relative paths are relative dataroot,
        # absolute paths are absolute.
        workspace: "ocm"
        # Stale DAQ configuration (determines when they are automatically
        # archived at startup)
        stale_acquiring_hours: 18
        stale_merging_hours: 720

    dpm:
        # DPM communication configuration
        req_endpoint: "zpb.rr://127.0.0.1:12350/"
```

(continues on next page)

(continued from previous page)

```
    pub_endpoint: "zpb.ps://127.0.0.1:12351/"
    timeout_sec: 5
```

11.3.6 Workspace

New in version 2.0.0.

The *daqOcmServer* workspace is the designated file system area used to store *Data Acquisition* state information persistently. The workspace location is controlled with the cfg/daq/workspace parameter and will be automatically initialized if directory does not exist. To prevent against accidental misconfiguration *daqOcmServer* will refuse to use the directory if it has unexpected file contents.

Note: When *daqOcmServer* is not running it is safe to delete the complete workspace. Be aware that if there are *Data Acquisitions* in progress this information will be lost.

The information stored in workspace is:

- List of known *Data Acquisitions*.
- For each *Data Acquisition* it stores the status, which contains the same information published as daqif.DaqStatus.
- For each *Data Acquisition* it stores the context, which contains the necessary information to be able to create the *Data Product Specification*. This includes data sources, FITS keywords provided to *daqOcmServer* for example.

When *daqOcmServer* starts up it will load the stored information so it is possible to continue the process. To avoid recovering completely obsolete *Data Acquisitions* there are two *configuration parameters* that are used to discard these, depending on whether the *Data Acquisition* was last known to be in state daqif.DaqState.StateAcquiring or py:attr:*daqif.DaqState.StateMerging*:

- cfg/daq/stale_acquiring_hours
- cfg/daq/stale_merging_hours

Important: As offline changes are not reflected in the persistent state it may happen that the recovered state is inaccurate. This is always a risk and currently *daqOcmServer* does not actively try to correct this.

The structure is as follows:

/
: Workspace root as configured via configuration file, environment variable or command line.

/list.json
: List of *Data Acquisitions*, as an array of *Data Acquisition* identifiers.

/in-progress/
: Root directory containing files related to each *Data Acquisition*.

/in-progress/{id}-status.json
: Contains persistent status for each *Data Acquisition* (where {id} is the *Data Acquisition* identifier).

/in-progress/{id}-context.json
: Contains persistent context for each *Data Acquisition* (where {id} is the *Data Acquisition* identifier).

/archive/
: When *Data Acquisition* is completed (transitions to state daqif.DaqState.StateCompleted) the in-progress files are moved here.

Note: Files in this directory are safe to deleted. An operational procedure is foreseen to specify when this should be done.

The following shows an example of files and directories in the workspace with an in progress *Data Acquisition*.

```
.
├── archive/
└── in-progress/
    ├── TEST.2021-05-18T14:49:03.905-context.json
    └── TEST.2021-05-18T14:49:03.905-status.json
```

11.3.7 Loggers

The following loggers are used (see logging-configuration for how to configure verbosity with log properties file):

daq.ocm
: General application logging.

daq.ocm.manager
: Used by component that manages all controllers and certain other functions.

daq.ocm.manager.awaitstate
: Logs details around the clients waiting for a particular state.

daq.ocm.controller
: Used by the component that controls the *Data Acquisition* lifecycle.

daq.ocm.eventlog
: Used to log events in a more structured manner than normal logs.

11.4 Standard MAL API

Note: *daqOcmServer* hosts the standard commands stdif.StdCmds under URI path /std.

The standard interface is documented in the RAD User Manual and should be consulted for details and only the brief is provided here.

11.4.1 Interfaces

class stdif.StdCmds

Standard command interface.

Stop() → str

Init() → str

Reset() → str

Enable() → str

Disable() → str

GetState() → str

GetStatus() → str

GetVersion() → str

Exit() → str

SetLogLevel(*info:* LogInfo) → str

11.4.2 Data Structures

class stdif.LogInfo

level: str

logger: str

class stdif.Status

status: str

source: str

12 Data Product Manager

12.1 Introduction

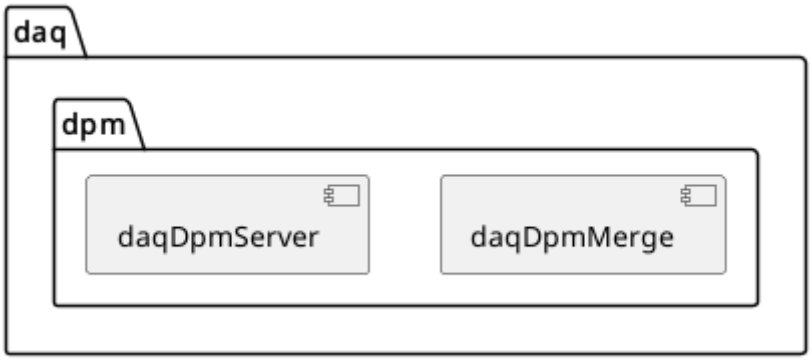


Fig. 12.1: DPM package contents

The package DPM provides the application components *daqDpmServer* and *daqDpmMerge*.

daqDpmServer is the server component that *daqOcmServer* communicates with to delegate the responsibility of creating the *Data Product* for each *Data Acquisition*. This separation is mainly done to allow *daqDpmServer* be used without an fully operational ICS for e.g. daytime merging and to be deployed on the same host as the *OLAS* interface for efficiency.

Key responsibilities:

- Provide interface to *daqOcmServer*.
- Transfer source files to local host.
- Validate source files prior to merging using keyword dictionaries (TBD).
- Execute *daqDpmMerge* to perform the merge using the now local files.
- Deliver result to specified recipients. If nothing is specified the file is delivered to *OLAS*.

daqDpmMerge is, under normal circumstances, executed as a subprocess of *daqOcmServer* and is the component that create the final *Data Product*. This is a standalone command line tool designed to be usable manually, without *daqDpmMerge*.

Key responsibilities:

- Execute the merge process to create final *Data Product*.

daqDpmServer implements the following request/reply interfaces:

MAL Interface	URI Path	Description
daqif.DpmControl	/dpm	Control interface for the server itself and those aspects unrelated to a *Data Acquisition*.
daqif.DpmDaqControl	/daq	Control interface for initiating, controlling and monitoring *Data Product* creation life-cycle from *Data Acquisitions*.

12.2 Server (daqDpmServer)

The main DPM application is *daqDpmServer*, which coordinates the merging of *Data Acquisition* source files to create the final *Data Product*.

Note: Interaction with *daqDpmServer* is mainly reserved for *daqOcmServer*.

12.2.1 State Machine

The *daqDpmServer* does not implement a state machine, when started it becomes operational automatically.

12.2.2 MAL URI Paths

The following tables summarize the request/reply service paths and topic paths for pub/sub.

Table 12.1: daqDpmServer URI paths.

URI Path	Root URI Configuration	Description
/dpm	cfg/req_endpoint	*DPM* control interface daqif.DpmControl.
/daq	cfg/req_endpoint	*DPM Data Acquisition* control interface daqif.DpmDaqControl.

Table 12.2: Topic URI paths.

Topic Type	URI Path	Root URI Configuration	Description
daqif.DaqStatus	/daq/status	cfg/pub_endpoint	Status updates to DAQ is published as change occurs. It is supp.
daqif. StorageStatus	/dpm/ storage	cfg/pub_endpoint	Storage status (importantly available space) is published at intervals in this topic.
daqif. InternalDaqStatus	internal/ daq/status	cfg/pub_endpoint	Internal variant of /daq/status topic used by *daqOcmServer*. New in version 3.1.0.

12.2.3 Command Line Arguments

Command line argument help is available under the option --help.

--proc-name ARG| -n ARG **(string) [default: *dpm*]**
Process instance name.

--config ARG| -c ARG **(string) [default: *config/daqDpmServer/config.yaml*]**
Config Path to application configuration file e.g. --config ocs/ocm.yaml (see *Configuration File* for configuration file content).

--log-level ARG| -l ARG **(enum) [default: *INFO*]**
Log level to use. One of ERROR, WARNING, STATE, EVENT, ACTION, INFO, DEBUG, TRACE.

--workspace ARG **(string) [default: *dpm*]**
Workspace used by *daqDpmServer* to store source files before merging as well as the result after merging is complete (see *daqDpmServer workspace* for details).

- Absolute paths are used as is.
- Relative paths are defined relative to cfg/dataroot.

--rr-uri ARG **(string) [default: *zpb.rr://127.0.0.1:12083/*]**
MAL server request root endpoint on which to accept requests. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

Specifying endpoint as command line argument takes predecence over configuration file parameter cfg/req_endpoint.

--ps-uri ARG **(string) [default: *zpb.ps://127.0.0.1:12084/*]**
MAL publish root endpoint on which to publish topics from. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

Specifying endpoint as command line argument takes predecence over configuration file parameter cfg/pub_endpoint.

--poll-once
Initiates operations once and then runs until there is no more work to do and then exits. Option is provided for interactive use, e.g. with manual error recovery.

12.2.4 Environment Variables

$DATAROOT
If defined it specifies the default value for for cfg/dataroot. If the configuration parameter is defined it takes precedence over $DATAROOT.

$DAQ_LOGS
Optionally specifies path where to create separate log file for *daqDpmServer*.

New in version 3.1.0.

12.2.5 Configuration File

This section describes what the configuration file parameters are and how to set them.

The configuration file is currently based on YAML and should be installed to one of the paths specified in $CFGPATH where it can be loaded using the *Config Path* and the command line argument --config ARG.

If a configuration parameter can be provided via command line, configuration file and environment variable the precedence order (high to low priority) is:

1. Command line value
2. Configuration file value
3. Environment variable value

Enumeration of parameters in the shorthand map/value where value is a map entry in map:

cfg/dataroot **(string) [default: *$DATAROOT*]**
: IFW standard output directory.

cfg/log_properties **(string)**
: *Config Path* to a log4cplus log configuration file. See also logging-configuration for important limitations.

cfg/req_endpoint **(string) [default: *zpb.rr://127.0.0.1:12085/*]**
: MAL server request root endpoint on which to accept requests. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

cfg/pub_endpoint **(string) [default: *zpb.ps://127.0.0.1:12086/*]**
: MAL server publish root endpoint on which to publish topics from. Trailing slashes are optional, e.g. example: "zpb.ps://127.0.0.1:12345/" or "zpb.ps://127.0.0.1:12345".

cfg/daq/workspace **(string) [default: *dpm*]**
: Workspace used by *daqDpmServer* to store source files before merging as well as the result after merging is complete (see *daqDpmServer workspace* for details).

  - Absolute paths are used as is (*recommended*).
  - Relative paths are defined relative to cfg/dataroot.

cfg/limits/daq **(integer) [default: 1]**
: Limits number of concurrent *Data Acquisitions* that *daqDpmServer* will process. Using 0 is infinite.

cfg/limits/merge **(integer) [default: 1]**
: Limits number of concurrent merge processes. Using 0 is infinite.

cfg/limit/net_receive **(integer) [default: 0]**
: Limits number of network receive tranfers. Using 0 is infinite.

cfg/limits/net_send **(integer) [default: 0]**
: Limits number of network send transfers. Using 0 is infinite.

Note: The following parameters are provided but not expected to be modified.

cfg/bin_merge **(string) [default: *daqDpmMerge*]**
: Merge application name.

cfg/bin_rsync **(string) [default: *rsync*]**
: Rsync application name.

Example (partial) configuration:

```
cfg:
  workspace: "/absolute/path/to/workspace"
  req_endpoint: "zpb.rr://127.0.0.1:12085/"
  pub_endpoint: "zpb.ps://127.0.0.1:12086/"
  log_properties: "log.properties"

  # Concurrency limits
  limits:
    daq: 2
    merge: 2
    net_receive: 5
    net_send: 5
```

12.2.6 Workspace

The *daqDpmServer* workspace is the designated file system area used to store both intermediate and final result of *Data Acquisitions*:

- Individual input files from data sources (i.e. FITS files).
- *Data Product Specification* that specifies how inputs are merged together.
- Various internal files used by DPM.
- Final *Data Product* when merged.

The structure is as follows:

/
: Workspace root as configured via configuration file, environment variable or command line.

/queue.json
: Queue of *Data Acquisitions*, as an array of *Data Acquisition* identifiers, that have been scheduled but are not yet completed. The order is significant and are processed in FIFO order.

/result/
: Directory containing *Data Product* results.

For each *Data Acquisition* the *Data Product* result is produced in state Merging by *daqDpmMerge* and is gurauranteed to be available from state Releasing.

The files follow the ICS name which is:

- {fileId}.fits
- {prefix}{fileId}.fits if a prefix has been chosen.

Note: *Data Acquisition* results are moved atomically into this directory and are then immutable. *daqDpmServer* requires read access until *Data Acquisition* reach state StateCompleted.

Like files in /archive/ an operational procedure is foreseen that specifies when files should be deleted. This procedure can e.g. ensure that *Data Product* has been successfully ingested and backed up by *OLAS*.

/in-progress/
: Root directory containing a directory for each *Data Acquisition* that has been queued for merging but is not yet completed. The directories are named after *Data Acquisition* id.

/in-progress/id/
: Contains persistent state for each *Data Acquisition* (where id is the *Data Acquisition* identifier). Also referred to as the Data Acquisition Workspace.

  sources/
  : Subdirectory containing *Data Product* source FITS files. They are renamed to avoid possibility of name collisions but the origin data source is identifiable given the naming pattern: {index}_{data source name}_{origin filename}.

  logs/
  : May contain log files related to the merge operation.

  specification.json
  : *Data Product Specification* specifying how to create *Data Product*. It is written and updated by *daqDpmServer* and read by *daqDpmMerge*.

    The file is produced in state Scheduled after parsing the received *Data Product Specification* for source files.

  sources.json
  : Specifies the source FITS files required to execute the merge. The JSON file specifies the remote origin location as well as the local file path where it will be copied. *daqDpmMerge* will use this as a source lookup.

    The file is produced in state Scheduled after parsing the received *Data Product Specification* for source files.

  status.json
  : Internal merge status maintained by *daqDpmServer*, used to be able to resume an interrupted merge process. The file is written for every state change.

result
: Symlink to result file. The file is produced in state Merging by *daqDpmMerge* and is guaranteed to be available from state Releasing.

/archive/
: Subdirectory for *StateCompleted Data Acquisitions*. Files in this directory are no longer used by *daqDpmServer* and can be removed.

/archive/id/
: Subdirectory for each *Data Acquisition*. When *Data Acquisition* is completed it is moved from /in-progress/id to /archive/id so structure is identical.

The following shows an example of files and directories in the workspace with one completed *Data Acquisition* and two in progress of being merged:

```
.
├── archive/
│   └── TEST.2021-05-18T15:10:02.101/
│       ├── result -> ../../result/TEST.2021-05-18T15:10:02.101.fits
│       ├── sources/
│       ├── logs/
│       ├── sources.json
│       ├── specification.json
│       └── status.json
├── in-progress/
│   ├── TEST.2021-05-18T14:49:03.905/
│   │   ├── sources/
│   │   │   ├── 0_dcs_TEST.2021-05-18T14:49:03.905-dcs.fits
│   │   │   └── 1_fcs_TEST.2021-05-18T14:49:03.905-fcs.fits
│   │   ├── logs/
│   │   ├── sources.json
│   │   ├── specification.json
│   │   └── status.json
│   └── TEST.2021-05-18T15:10:02.101/
│       ├── sources/
│       ├── logs/
│       ├── sources.json
│       ├── specification.json
│       └── status.json
└── result/
    └── TEST.2021-05-18T15:10:02.101.fits
```

12.2.7 Loggers

The following loggers are used (see logging-configuration for how to configure verbosity with log properties file):

daq.dpm
: General application logging.

daq.dpm.scheduler
: Schedules the execution of merges from the queue of *Data Acquisitions*.

daq.dpm.controller
: Logs related to the controller which manages the execution of the merge, including transfer of missing inputs.

daq.dpm.transfer
: Dedicated logger for the FITS file input transfer.

daq.dpm.merger
: Output from the *daqDpmMerge* subprocess.

12.3 Merger (daqDpmMerge)

Standalone application that creates the final Data Product from a specification and is normally executed as a subprocess of *daqDpmServer*.

12.3.1 Command Line Arguments

Synopsis:

daqDpmMerge [options] <specification-file>

Where

<specification-file>
: Data Product Specification file. To read from standard input use -. See *Data Product Specification* for file format details.

Options:

--root DIR
: Root directory DIR from which relative source paths in specification-file will be resolved.

  By default the root directory will be set to:

  1. The same directory that holds specification-file, if it is a regular file.
  2. The current working directory, if specification-file is not a regular file.

--outfile FILE | -o FILE
: FITS output file name, e.g. -o myfits.fits or *–outfile=/path/to/myfits.fits*`. By default the output

name will be derived using the specification-file fileId property: <fileId>.fits.

Relative paths are relative to the root directory.

--dry-run
: Skips operations with visible side-effects. All inputs are opened in read-only mode. Some operations are performed using in-memory FITS file.

  Useful for validating arguments and other inputs.

--json
: Print status messages to standard output in JSON format with one message per line. By default status messages are printed in human readable form.

--logfile FILE | -l FILE
: Specifies log file to create and append to. Relative paths are relative to the root directory.

--help
: Show help and exit.

--version
: Show version information and exit.

12.3.2 Environment Variables

TBD

12.3.3 Exit Codes

Table 12.3: daqDpmMerge exit codes

Code	Description
0	Success
10x	**Problem with spec-file**
100	spec-file not found.
101	Invalid JSON.
102	Invalid schema.
11x	**Problem with source file(s)**
110	Referenced source file not found.
255	Internal error.

12.4 Data Product Specification

For an introductory high-level overview of how data products are created see section *Data Product Creation*.

The Data Product Specification is the primary input to *daqDpmMerge* and describes how the final *Data Product* is created. All source files referenced in the specification must be locally accessible and relative paths are relative to a single root path which is either the directory containing the specification or an explicit root directory passed as an option to *daqDpmMerge*.

Merging can be done by creating a new file or to merge *in-place*. If no FITS file target is specified using the property $\mathrm{target/source}$ described below, then a new empty file will be created automatically into which the sources will be added.

Note: It is expected that *daqDpmMerge* will validate keywords against a set of keyword dictionaries.

12.4.1 Primary HDU keywords

There are two types of sources for primary HDU keywords:

1. Keywords in JSON format.

   These keywords were provided directly to *daqOcmServer* using the *available APIs*. These keywords are defined by the named JSON object FitsKeywordsSource.

2. Keywords copied from source FITS files.

   Which keywords from the source primary HDU to copy to the target primary HDU are derived using the keyword rules specified in the source.

All these keywords will be merged together:

- *Value Keyword*s and *ESO Keyword*s keyword collisions are resolved by keeping the keyword from highest priority source.
- *Commentary Keyword*s are appended in source priority order.

And then by default sorted as specified by DICD [*RD1*](section 3.1.1.3):

1. Value keywords are sorted by source order:

   The individual order of value keywords from the same source (file or list of keywords) is kept to not reorder keywords that belong together. For example it make sense to keep *RA* and *DEC* together:

```
OBJECT  = 'karma_cdfs_8_LP.cat' / Original target.
RA      =        53.087446 / [deg] 03:32:20.9 RA (J2000) pointing
DEC     =         -27.84692 / [deg] -27:50:48.9 DEC (J2000) pointing
```

   Rather than e.g. sorting alphabetically:

```
DEC     =        -27.84692 / [deg] -27:50:48.9 DEC (J2000) pointing
OBJECT  = 'karma_cdfs_8_LP.cat' / Original target.
RA      =         53.087446 / [deg] 03:32:20.9 RA (J2000) pointing
```

2. ESO hierarchical keywords are sorted by:
    1. Category[14] (DPR, OBS, TPL, GEN, TEL, ADA, INS, DET, *any other*)
    2. Keyword name in alphabetic order.

12.4.2 HDU Extensions

The FITS HDU extensions are merged in order of descending priority:

1. Extensions from the *in-place* target, if any.
2. Extensions from other source files.

   The source files in attribute sources are specified in descending priority. Like keywords the relative order between extensions in the same file is kept.

12.4.3 JSON Description

See json-schema-dpspec-intro for detailed specification.

[14] Category in an ESO hierarchical keyword is the first token in the logical keyword name. The category is *TEL* in the logical keyword name *TEL MOON RA*. See also glossary *ESO Keyword*.

13 MAL Interface

This interface is hosted at https://gitlab.eso.org/ifw/ifw-daqif/.

The following sections document the MAL interface daqif [rd-daqif] in a language agnostic manner. For data structures only the data members are documented, not the accessors generated by MAL. The names are not fully qualified as it is different across the supported languages.

daqif are the domain specific interfaces implemented and used by *daqOcmServer* and *daqDpmServer*. *daqOcmServer* implements in addition the *standard interface*.

13.1 OCM Data Acquisition Control

Note: daqOcmServer hosts the *Data Acquisition* control interface daqif.OcmDaqControl under URI path /daq.

---

class daqif.OcmDaqControl

OCM *Data Acquisition* control interface.

StartDaq(*id*, *file_prefix*, *prim_sources*, *meta_sources*, *properties*) → *DaqReply*

Create and start new data acquisition with v1 merge-heuristics. If *id* is not provided (left empty) daqOcmServer will create a unique identifer automatically.

Heuristics

The following heuristics is used to determine how results are merged together. For more control use StartDaqV2().

- If a *single* primary data source produce a *single* FITS file this will selected as the *in-place* merge target. - Otherwise data sources produce multi-extension FITS files with no data in primary HDU. - Data sources have the following relative priority:

  Note: Order determines merge order of keywords and extensions.

  ---

  1. Primary data sources in the order user specified in *prim_sources*.
  2. Metadata sources in the order user specified to *meta_sources*.

- Each data source produce keywords that will be merged to *Data Product* primary HDU returned via *metadaqif* or with FITS file primary HDU.
- There is currently no way to provide keyword rules that select and transforms keywords. All keywords in *user* class will be merged.

Arguments

The format for *prim_sources* and *meta_sources* is a space separated list of sources in the format: <name>@<rr-uri>, e.g. meta-source@zpb.rr://example:1234/daq.

There must be at least 1 data source.

Fail-fast

If any source fails to start the command will fail.

Parameters

- id (str) – Optional unique identifier of data acquisition.
- file_prefix (str) – Optional file name prefix.
- prim_sources (str) – List of primary data sources (e.g. detectors).
- meta_sources (str) – List of metadata sources (e.g. FCF Device Manager or CCS).
- properties (str) – JSON properties (see json-schema-startdaqproperties).

Returns

If *id* was provided that is returned as an acknowledgement, otherwise the *id* generated by daqOcmServer is returned.

Return type

DaqReply

Raises

DaqException – On fatal error.

StartDaqV2(*specification*) → *DaqReply*

New in version 2.1.0.

Create and start new data acquisition from JSON specification.

StartDaqV2() was introduced in to enable the capabilities of *daqDpmMerge* that was not possible to express using StartDaq(). The decision was also made to use JSON to express the specification to allow more flexible extensibility.

See json-schema-startdaqv2 for detailed documentation of the JSON schema.

Note: This version is strictly more capable than StartDaq() and is required when additional control of the outcome is required. Unlike StartDaq() it also supports specifying how to merge FITS files that already exist.

Fail-fast

If any source fails to start the command will fail.

Parameters
specification (str) – Serialized JSON document. See json-schema-startdaqv2specification for details.

Returns
If *id* was provided that is returned as an acknowledgement, otherwise the *id* generated by daqOcmServer is returned.

Return type
DaqReply

Raises
DaqException – On fatal error.

StopDaq(*id*) → *DaqReply*

Stops data acquisition and keep any data acquired.

Fail-Fast

If all data sources fail the command will report error by throwing DaqException. A partially successful execution is reported using normal status reply (which includes information about any partial failure). This is done because it is prioritized to create a partially complete data product over discarding all acquired data.

Parameters
id (str) – Id of data acquisition to stop.

Return type
DaqReply

Raises
DaqException – On fatal error.

ForceStopDaq(*id*) → *DaqReply*

Like StopDaq() the command stops data acquisition and keeps any data acquired. The only difference is that if non-fatal error occurs the *Data Acquisition* is marked as stopped whereas StopDaq() would not.

Fail-slow

The command is resilient to non-fatal errors. If any non-fatal errors occur the error flag in the reply is set but no exception is thrown. If fatal error occur DaqException is thrown.

Warning: Any data source that failed to stop properly will not be able to provide data to the *Data Acquisition* or the final *Data Product*.

Although *daqOcmServer* is always left in a consistent state, any data source that failed to stop may not. Manual intervention may be necessary to restore a problematic data source to a functional state.

Parameters
id (str) – Id of data acquisition to forcibly stop.

Return type
DaqReply

Raises
DaqException – On fatal error.

AbortDaq(*id*) → *DaqReply*

Aborts data acquisition and discards any data acquired.

Fail-fast

If any error occur DaqException is thrown and the DAQ state remains in *Aborting*.

Parameters
id (str) – Id of data acquisition to abort.

Return type
DaqReply

Raises
DaqException – On fatal error.

ForceAbortDaq(*id*) → *DaqReply*

Like AbortDaq() the command aborts data acquisition and discards any data acquired. The only difference is that if non-fatal error occurs the *Data Acquisition* is marked as aborted whereas AbortDaq() would not.

Fail-slow

The command is resilient to non-fatal errors. If any non-fatal errors occur the error flag in the reply is set but no exception is thrown. If fatal error occur *DaqException* is thrown.

Warning: Although daqOcmServer is always left in a consistent state, any data source that failed to abort may not. Manual intervention may be necessary to restore a problematic data source to a functional state.

Parameters
id (str) – Id of data acquisition to forcibly abort.

Return type
DaqReply

Raises
DaqException – On fatal error.

UpdateKeywords(*id: str*, *keywords: str*) → *DaqReply*

Update keywords for specified *Data Acquisition*. For each keyword:

- If it already exist[15], it is updated with the provided value,
- otherwise it is added.

Fail-fast

Command will not make any attempt to be robust against errors. Failures are treated as fatal and DaqException will be thrown.

Parameters

- id (str) – Id of data acquisition to update.
- keywords (str) – JSON encoded set of keywords. Refer to JSON keywords schema for details.

Return type
DaqReply

Raises
DaqException – On fatal error.

AwaitDaqState(*id*, *state*, *substate*, *timeout*) → *AwaitDaqReply*

Replies when *Data Acquisition* reaches the requested state or when this state is impossible to reach. This is mainly used to support sequencing of higher level coordination activities.

The command will reply when condition is fulfilled or times out. When it times out this is indicated in the reply with AwaitDaqReply.timeout set to true.

Condition is fulfilled if *Data Acquisition* either:

1. Enters requested state. In this case the current state in reply is the same as the

requested state.

2. Enters or is already in a state such that there is no valid transition to the requested state. In this case the current state in reply will not be the same as the requested state.

Note: If *Data Acquisition* is archived the request will reply if condition is already fulfilled otherwise exception will be raised.

Parameters

- id (str) – Id of *Data Acquisition* to await.
- state (DaqState) – State of *Data Acquisition* to await.
- substate (DaqSubState) – Substate of *Data Acquisition* to await.
- timeout (float) – Duration in seconds to wait for condition to be fulfilled. Must be > 0.

Returns
Current *Data Acquisition* status and indication if operation timed out.

Return type
AwaitDaqReply

Raises
DaqException – On fatal error (e.g. invalid arguments).

GetStatus(*id*) → *DaqStatus*
Get status of a *Data Acquisition*.

Note: If *Data Acquisition* has completed *daqOcmServer* will try to reply with status from most recently archived *Data Acquisition* with the same id.

Parameters
id (str) – Id of *Data Acquisition* to get status for.

Return type
DaqStatus

Raises
DaqException – On fatal error.

GetActiveList() → List[*DaqStatus*]
Get list of active *Data Acquisitions* (i.e. not in daqif.DaqState.StateCompleted).

Returns
List of active *Data Acquisitions*.

Return type
List[*DaqStatus*]

Raises
DaqException – On fatal error.

13.2 DPM Control

Interface for application control. It has a similar role as stdif but limited to operations supported by *daqDpmServer*.

Note: This interface is reserved for *daqOcmServer* and is not meant to be used by end-users.

daqDpmServer hosts the interface daqif.DpmControl under URI path /dpm.

---

class daqif.DpmControl

DPM control interface for non-DAQ related commands.

QueryStorageStatus() → *StorageStatus*

Queries disk space usage and availability on the file system used by *daqDpmServer workspace*.

Return type
StorageStatus

Raises
RuntimeError – On fatal error.

Exit() → str

Terminates application in a controlled manner.

Note: Reply is sent before application terminates, but there is no guaranteed delivery.

Return type
str

Raises
DaqException – On fatal error.

[15] OCM will compare keyword name as provided, while considering if it's an ESO hiearchical keyword or standard FITS keyword.

13.3 DPM Data Acquisition Control

Interface for *Data Acquisition* control.

Note: This interface is reserved for *daqOcmServer* and is not meant to be used by end-users.

daqDpmServer hosts the interface daqif.DpmDaqControl under URI path /daq.

---

class daqif.DpmDaqControl

DPM control interface for initiating, monitoring and controlling *Data Product* creation life-cycle of *Data Acquisitions*.

QueueDaq(*specification*) → *DaqReply*

Add *Data Acquisition* to queue for merging.

Parameters
specification (str) – JSON encoded string following schema in *Data Product Specification* specifying how to create the *Data Product*.

Return type
DaqReply

Raises
DaqException – On fatal error.

AbortDaq(*id*) → *DaqReply*

Aborts merging.

Fail-fast

If any error occur DaqException is thrown and the DAQ state remains in *Aborting*.

Parameters
id (str) – Id of data acquisition to abort.

Return type
DaqReply

Raises
DaqException – On fatal error.

GetDaqStatus(*id*) → *DaqStatus*

Query *Data Acquisition* status. If no *Data Acquisition* has previously been queued with QueueDaq() or is archived (in a completed state) an exceptional reply is returned.

Parameters
id (str) – Id of data acquisition to get status for.

Return type
DaqStatus

Raises
DaqException – On fatal error.

GetActiveDaqs() → List[*DaqStatus*]

Get list of active *Data Acquisitions* (i.e. not in daqif.DaqState.StateCompleted).

Returns
List of active *Data Acquisitions*.

Return type
List[*DaqStatus*]

Raises
DaqException – On fatal error.

GetInternalDaqStatus(*id*) → *InternalDaqStatus*

Internal version of GetDaqStatus().

New in version 3.1.0.

Parameters
id (str) – Id of data acquisition to get status for.

Return type
InternalDaqStatus

Raises
DaqException – On fatal error.

13.4 Data Structures

This section contains the data structures that are used by the daqif interfaces daqif.OcmDaqControl, daqif.DpmControl and daqif.DpmDaqControl or used as pub/sub topic type.

class daqif.DaqState

Enumeration of top-level *Data Acquisition* states, see *The Data Acquisition Process* for additional details.

StateAcquiring

State that span the initial point it is created in OCM through the phase where data is acquired until all inputs for the Data Product have been created. See also *StateAcquiring*.

StateMerging

State that spans not where a *Data Product* is created from the acquired data. Phase where *Data Product* is created. See also *StateMerging*.

StateCompleted

Data Acquisition is completed.

StateUndefined

Only used for technical reasons. This is not an expected state.

class daqif.DaqSubState

Enumeration of possible *Data Acquisition* sub-states, see *The Data Acquisition Process* for additional details.

Note: Only a subset of the listed states are possible/valid in each DaqState.

NotStarted

Data Acquisition is created but not yet started.

Starting

Transitional state where *Data Acquisition* is being started. Some data sources might already be acquiring data. When all sources have started the *Data Acquisition* transitions to Acquiring.

Acquiring

Data is being acquired.

Stopping

Transitional state to Stopped. Some data sources might still be acquiring data and are about to be stopped. When all data sources have stopped the *Data Acquisition* transitions to Stopped.

Stopped

All data has been acquired.

Aborting

Transitional state where *Data Acquisition* remains while it is in the process of being aborted.

Aborted

If a *Data Acquisition* has errors or otherwise needs to be aborted the *Data Acquisition* also include the Aborted state.

NotScheduled

Before *Data Acquisition* is acknowledged by *DPM* it remains in NotScheduled.

Scheduled

Data Acquisition is acknowledged by *DPM* and is scheduled for merging (i.e. the *Data Acquisition* is in the backlog).

Collecting

Inputs for *Data Product* are being collected to *DPM* host in preparation for merging.

Merging

Data Product is being created by merging input files together.

Releasing

Data Product is being released to specified receivers, which in the normal case is *OLAS*.

Completed

Nominal final state for a *Data Acquisition* (i.e. final state when *Data Acquisition* is not aborted).

Undefined

Only used for technical reasons. This is not an expected state.

class daqif.DaqException

Exception used by OcmDaqControl.

id: str

Data Acquisition identifier.

message: str

Exception message.

class daqif.RuntimeError

Exception used when there is no *Data Acquisition*.

message: str

Exception message.

class daqif.DaqStatus

Contains the *Data Acquisition* status reply.

id: str

Data Acquisition identifier.

fileId: str

A unique *OLAS* compatible file id is assigned by *daqOcmServer* when *Data Acquisition* is started. This is also used as a basis for the *Archive File Name* and recoded as ARCFILE FITS keyword with only the addition of the FITS extension: {fileId}.fits[*RD7*].

See also:

Archive File Name

state: DaqState

Data Acquisition state at the time the reply was sent.

substate: DaqSubState

Data Acquisition sub-state at the time the reply was sent.

timestamp: double

Timestamp of last status update.

error: boolean

A derived property that if set indicates that there is one or more issues with *Data Acquisition*. This is also the main mechanism for communicating asynchronous errors that cannot be communicated as part of a command reply. Attribute message contains the details.

message: str

Contains message related to current status. This is currently mainly used to provide information about active alerts.

result: str

Once *Data Product* has been created successfully in state Merging this property will contain the file path.

To ensure that result is available synchronize to substate Releasing or Completed.

New in version 2.0.0.

class daqif.DaqReply

Basic reply type that also indicate non-fatal errors.

id: str

Data Acquisition identifier.

error: boolean

When True this indicates *Data Acquisition* has one or more issues which can be read from DaqStatus.message.

class daqif.AwaitDaqReply

timeout: boolean

Indicates if the await request timed out before condition was fulfilled.

status: DaqStatus

Data Acquisition status when reply was sent.

class daqif.StorageStatus

Provides storage status information.

capacity: int

Partition size in number of bytes.

free: int

Free space on the file system in number of bytes.

available: int

Free space available to *daqDpmServer*, if non-privileged, in number of bytes. This may be less or equal to free.

class daqif.InternalDaqStatus

Structure used for internal communication between *daqOcmServer* and *daqDpmServer*. It is left undocumented on purpose and should not be used by end-users.

New in version 3.1.0.

See also:

If you are looking for the MAL URI endpoints c.f.:

- *OCM Server*
- *DPM Server*

Python Module Index