RPW Operation Centre

RPW Calibration Software Interface Control Document

ROC-PRO-PIP-ICD-00037-LES
Iss.01, Rev.01

Prepared by:	**Function:**	**Signature:**	**Date**
Manuel Duarte Xavier Bonnin	RPW Ground Segment software engineer RPW Ground Segment Deputy Project Manager		19/03/2017
Verified by:	**Function:**	**Signature:**	**Date**
Xavier Bonnin	RPW Ground Segment Deputy Project Manager		Dd/mm/yyyy
Approved by:	**Function:**	**Signature:**	**Date**
Yvonne de Conchy	RPW Ground Segment Project Manager		Dd/mm/yyyy
For application:	**Function:**	**Signature:**	**Date**
Name	Team Member #4		Dd/mm/yyyy

CLASSIFICATION **PUBLIC** ☒ **RESTRICTED** ☐

l'Observatoire de Paris LESIA
Laboratoire d'Études Spatiales et d'Instrumentation en Astrophysique

CNRS-Observatoire de PARIS
Section de MEUDON – LESIA
5, place Jules Janssen
92195 Meudon Cedex – France

Change Record

Issue	Rev.	Date	Authors	Modifications
0	0	08/07/2015	Manuel Duarte	First draft (inherits from the RCS ICD for ROC-SGSE, ROC-TST-GSE-ICD-00023)
1	0	09/11/2016	Xavier Bonnin	First release
1	1	19/03/2017	Xavier Bonnin	- RCS CDF outputs path is now passed via the CLI specific input parameters, and not returned by the software in the stdout. - "outputs.release" attribute removed from the descriptor file - Add "outputs.template" attribute in the descriptor file - Rename "Exceptions and errors" section to "Exceptions handling" and update the content

Acronym List

Acronym	Definition
BP	Basic Parameters
CDF	Common Data Format
CLI	Command Line Interface
CSV	Comma Separated Values
ICD	Interface Control Document
ID	Identifier
I/F	Interface
I/O	Input/Output
JSON	JavaScript Object Notation

ROC	RPW Operation Centre
RPW	Radio and Plasma Waves instrument
SCM	Search Coil Magnetometer
SGSE	Software Ground Support Equipment
SVN	SubVersioN
S/W	Software
TDS	Time Domain Sampler
LFR	Low Frequency Receiver
THR	Thermal Noise and High Frequency Receivers
WF	Waveform
XML	eXtended Markup Language

Table of Contents

List of figures

1 GENERAL

1.1 Scope of the Document

The RPW Calibration Software Interface Control Document (RCS ICD) describes the interface to be implemented in the RPW Calibration Software (RCS), in order to be autonomously run by the ROC Operations and Data Pipeline (RODP) [RD1].

The RCS definition covers all of the programs that produce RPW calibrated science data at both analyser and sensor levels.

It is the responsibility of the teams in charge to deliver to the ROC a RCS fully compliant with this ICD. The way the software (S/W) must be delivered to the ROC is described in the "ROC Engineering Guidelines for External Users" (REGU) document [RD4]. The REGU should be read in complement to the present document.

1.2 Applicable Documents

This document responds to the requirements of the documents listed in the following table:

Mark	Reference/Iss/Rev	Title of the document	Authors	Date
AD1				
AD2				
AD3				
AD4				

1.3 Reference Documents

This document is based on the documents listed in the following table:

Mark	Reference/Iss/Rev	Title of the document	Authors	Date
RD1	ROC-GEN-SYS-SDD-00036-LES/1/0	ROC Software System Design Document	X.Bonnin, S.Lion	02/12/2016
RD2	ROC-TST-GSE-SPC-00004-LES/1/0	ROC SGSE Software Design Document	X.Bonnin	14/02/2016
RD3	ROC-TST-GSE-ICD-00023-LES/2/3	ROC-SGSE Calibration Software Interface Control Document	Manuel Duarte	10/05/2016
RD4	ROC-GEN-SYS-NTT-00019-LES/1/2	ROC Engineering Guidelines for External Users	X.Bonnin	15/12/2016
RD5	ROC-PRO-DAT-NTT-00006-LES/1/0	RPW Data Products	X.Bonnin	15/02/2016
RD6	ROC-GEN-SYS-NTT-00008-LES/1/2	ROC engineering guidelines	X.Bonnin	15/12/2016
RD7	ECMA-404.pdf/1st edition	The JSON Data interchange format	ECMA	10/2013
RD8	ROC-TST-SFT-SUM-00027-LES/1/1	ROC-SGSE_Validation_Tool_User_Manual	M.Duarte	06/05/2016

2 CONSIDERATIONS & OBJECTIVES

2.1 RPW Calibration Software (RCS) implementation philosophy

During the Solar Orbiter mission, the ROC will be in charge to produce and deliver to ESA calibrated science data for the RPW instrument. The task will be regularly performed in an automated way by the RODP; the main RPW data processing pipeline hosted on the ROC server at LESIA (Meudon, France). Nevertheless, it is the role of the RCS to calibrate the instrument science data and to generate the resulting output data files. Such functionality requires RCS to be integrated into the RODP.

Since the programming language chosen by teams in charge can differ from a RCS to another, the integration must be done in a standard way via a dedicated common interface in order to: (i) minimize the human intervention, (ii) ensure the traceability of S/W and its data products, (iii) mitigate the possible points of failure in the RODP.

2.2 RCS deliverables

RCS will be supplied to the ROC by teams in charge. It is strongly recommended to reduce as much as possible the number of S/W to be delivered by each team in order to optimize the delivery, installation and maintenance processes.

Single S/W by team, but producing several RPW data products, should be the optimal solution. Additionally, a given data product should be as much as possible generated by a given class or function in the RCS.

2.3 RCS data products

The RCS data products will be RPW calibrated science data sets, at processing levels higher than L1 and saved in Common Data Format (CDF) files [RD5]. Especially, the ROC does not expect that the RODP handles other RCS data products than these data sets. In consequence, RCS should not be used to generate derived science data products, such as summary plots. If it is the case, the RCS shall include an option to trigger the production of derived products.

The full list of datasets to be produced by the RODP, as well as the data processing levels to be applied is reported in [RD5].

2.4 Objectives related to this document

In order for the teams to be able to produce and configure their RCS to be run at the ROC, it is necessary to define in detail the functionalities of the interface, and more particularly:

- How the RCS shall be called in a standard way
- How the RODP will identify and call the RCS
- How the RODP will identify and provide to RCS the required input data
- How and where the RCS will store the generated data products
- How the RODP pipeline will check that RCS run without error, and the resulting data products were correctly written.

This will be the purpose of the next sections. Section 3 describes the interface to be implemented on the RCS (i.e., teams side). Section 4 presents the mechanisms to run the RCS on the RODP side. Section 5 gives conventions concerning the RCS output files.

3 DESCRIPTION OF THE INTERFACE BETWEEN RODP AND RCS: RCS SIDE

This section presents the interface that must be found in the RCS, in order to be autonomously run by the RODP. This interface is composed of two main elements:

- A command line interface (CLI), that allows the RODP to call the RCS in a standard way
- A descriptor file (also called "descriptor"), that provides information about the RCS to the RODP. This information must help the pipeline to identify the RCS and build the CLI calling sequence.

3.1 Description of the RCS command line interface (CLI)

The RCS command line interface (CLI) is detailed in this section, namely: the general convention, the calling sequence formalism and the exception handling. A full example of such a CLI call is given in the appendix 6.1.

3.1.1 CLI executable convention

The CLI shall comply the following convention:

- The ROC shall be able to run the RCS by calling an executable - binary or script – file only.
- There must be a single one executable file per S/W.
- A team in charge of a RCS shall ensure that the executable file can be launched on the ROC server environment: a Linux Debian Operating System (OS) with Bourne-Again Shell (BASH), as a primary shell.
- The name of the executable file shall contain alphanumeric characters only. Only the "_" underscore character must be used as a delimiter to separate two fields in the name.

According to the convention above, every RCS CLI calling sequence shall start with:

```
$EXECUTABLE
```

Where `EXECUTABLE` is here the name of the executable.

Note that the interface does not work in the case where the executable is split in several words (for example, IDL batch commands usually work running "`idl -e program -args [...]`" calling sequence). In such situation, the actual RCS launch command must be wrapped into a BASH script. This script will be seen as the main RCS executable from the RODP point of view, and must be thus compliant with this ICD.

3.1.2 CLI input parameters

The CLI input parameters to be used to call the RCS executable, are described in the next section.

3.1.2.1 RCS function name

Each RCS shall contain one or more functions (or classes), which are used to create output data files. Each function has its own set of inputs/outputs that must be known by the RODP to

call it correctly. This will be the purpose of the S/W descriptor file, as explained in the section 3.2.

In order to allow the RODP to launch a given function of the S/W from the CLI, the executable shall provide the name of this function as a first argument.

It means that every calling sequence of a RCS executable shall start as followed:

```
$EXECUTABLE func1
```

Where `EXECUTABLE` is here the name of the executable, and `func1` is the name of the S/W function to be run.

Note that this syntax is not mandatory if input keywords are called, as explained in the section 3.1.2.4.

3.1.2.2 Common input parameters

In addition to the name of the function as a primary argument, the RCS executable calling sequence shall accept the following input parameters:

`--log [/path/to/logdir]`: the absolute path **`/path/to/logdir`** to the directory where the log file(s) will be saved.

`--config [/path/to/configfile]`: if required, the absolute path **`/path/to/configfile`** to the S/W configuration file.

The common input parameters shall take only one argument.

3.1.2.3 Specific input parameters

The specific input parameters shall provide the path to the RCS input/output RPW data files in CDF format.

The specific input parameters shall be called in the CLI following the convention above:

- The specific input parameter calling sequence shall be formed by a *"--flag value*" pair, where "*flag*" shall be a lowercase string, containing only alphanumeric characters and the underscore '_' as a separator, and preceded by the double hyphen prefix '--'. The "*value*" is the parameter value to be passed to the S/W function.
- The "*flag*" name shall be unique for a given S/W function calling sequence.

A typical RCS call, mixing some common and specific parameters of a given `func1` function, looks like:

```
$EXECUTABLE func1 --input_1 /path/to/input1 --input_2 /path/to/input2 \
    --output_1 /path/to/output1 --log /path/to/log
```

The order of the input parameters is not important, except the function name, which must be always the first argument to be called.

The way the RODP automatically identifies a function and its inputs/outputs from information in the descriptor, and how it builds the corresponding calling sequence is explained in the section 4.2.

Note that additional inputs such as the path to the master CDF or calibration table files, shall not be passed as input parameters to the CLI, but in a configuration file to be loaded by the S/W it-self, using the dedicated `--config` parameter.

3.1.2.4 Input keywords

The RCS executable calling sequence shall support at least the following input keywords:

`--identification`: return information about the S/W. The format and content of the returned stream shall correspond to the `"identification"` JSON object in the S/W descriptor file. (See the section 3.2 for more details about the descriptor file.)

`--version`: return information about the current S/W release. The content of the returned stream shall comply the `"release.version"` attribute value in the S/W descriptor file.

`--help`: display a help message.

For instance, to get information about the current version of the S/W:

```
$EXECUTABLE --version
```

Where `EXECUTABLE` is name of the executable file.

Note:

- Input keywords do not take argument.
- Calling the input keyword does not require to provide the function name `func1` as a first argument.

3.1.3 RCS exception handling

The RCS shall include a mechanism to handle exceptions that can occur during the execution. The exceptions to be notified to the RODP, and the expected actions to be performed by the RCS are explained in this section.

3.1.3.1 RCS exception level definition

Table below gives the definition of the exception levels to be implemented in the RCS.

Exception level	Definition
Medium	It is a "warning" exception level. It alerts about a potential issue, but does not necessarily require stopping the execution.
High	It is an "error" exception level, which requires aborting the execution.

Table 1. RCS exception levels.

3.1.3.2 Identified exceptions

A RCS shall raise an exception if it encounters one of the following events. In each case, the RCS shall return a message giving the detailed cause of the exception.

Event	Exception level	Comment
Input(s) data file(s) cannot be found or read correctly	High	
Input(s) file(s) inconsistency	High	It concerns the case where the input file(s) are not the ones expected. (In this case the RCS should include checking of the input file meta-data)

Additional file(s) cannot be found or read correctly	High	It can concern config. file, master CDF and/or calibration table file(s)
Output(s) file(s) cannot be written correctly	High	e.g., bad write permissions, output directory not found, etc.
Log file cannot be written correctly	High	e.g., bad write permissions, output directory not found, etc.
S/W execution environment not defined	High	e.g., environment variables required by the RCS are not defined correctly
Output data file(s) not named correctly (TBC)	Medium	The RCS shall check that name of the output files
Inconsistencies of global attributes in the master CDF for current output dataset (TBC)	Medium	The RCS shall check that the global attributes in the master CDF are consistent

Table 2. RCS exception list.

3.1.3.3 Output error code convention

If a "high" level exception is raised, the RCS shall exit with an error code 1. The error information (i.e., code, type of error and message) shall be written in the standard error (*stderr*). It will be then caught by the RODP and stored into the ROC database for investigations. An ERROR level event shall be also reported into the log file by the RCS before exiting (see section 5.2.1 for the definition of log levels).

If the error code is 0, the RODP will consider that the S/W execution has succeeded. Any other code value will stop the job in the ROC pipeline and emit an error.

3.2 RCS descriptor file specification

The RCS descriptor file is the second key element of the interface with the CLI. It contains information about the RCS that helps the RODP to automatically identify:

- The S/W it-self (e.g., name, identifier, description)
- The release (e.g., version number, release date, author)
- The execution environment (e.g., setup script, executable, configuration, etc.)
- A detailed description of the output files produced by the S/W (name, identifier, description, release, etc.)
- The list of calling sequences for each S/W function and the corresponding inputs and outputs

The information will be used, among others, by the RODP to automatically build the calling sequence for a given RCS function, and to monitor the outputs creation. It results that the RODP will not be able to run a RCS delivered without, or with a badly formatted, descriptor file.

The content of the descriptor is checked and inserted into the RODP database during the registration step (see section 4.1.3).

An example of such a descriptor file can be found in the appendix 6.1.

3.2.1 General convention

The descriptor files shall be delivered with the RCS. They shall follow the convention below:

- The descriptor file shall be written in the JSON format [RD7]
- It shall be placed in the S/W main directory
- It shall be named "***roc_sw_descriptor.json***"
- There must be only one descriptor file per S/W and per release
- All of the paths defined in the S/W descriptor file shall be relative to the S/W main directory.

3.2.2 Descriptor file structure

The structure of the descriptor file is detailed in the following sections.

3.2.2.1 Identification

Each S/W will be identified in the pipeline by the attributes provided in the `"identification"` JSON object:

- `"project"`**:** name of the project. It shall be "ROC".
- `"name"`**:** full name of the RCS, to be human readable.
- `"identifier"`**:** a name used as a unique reference by the RODP to identify the S/W. This identifier (ID) shall contain alphanumerical uppercase characters only. The hyphen character shall be used a separator if required (e.g., "THR-CALBAR"). This ID shall be assigned in agreement with the ROC team to avoid duplicated names.
- `"description"`**:** Short description/purpose of the S/W.

3.2.2.2 Release

The `"release"` object shall inform about the current S/W release. It shall contain the following attributes:

- `"version"`**:** Current version of the S/W in the format following the ROC convention [RD4].
- `"date"`**:** Date of the release of the S/W in the format 'YYYY-MM-DD', where 'YYYY', 'MM' and 'DD' are respectively the year, month and date of the release.
- `"author"`: Name of the person, team or entity responsible of the release
- `"contact"`: contact of the author (e.g., email or address)
- `"institute"`: Name of the institute that delivers the release
- `"modification"`: a string containing the list of S/W modifications in the current release.

In addition, the `"release"` object can provide the following optional attributes:

- `"reference"`: name of a file as reference for the documentation, used as an indication for the ROC team (e.g., user manual).
- `"url"`: indication for an online resource.

3.2.2.3 Environment

The attributes defined in the object `"environment"` are used for setting the RCS environment variables, and reset it to the initial state. It shall contain:

- `"executable"`: Relative path to the S/W executable file to be called by the RODP, using the CLI. It corresponds to the **`EXECUTABLE`** in the calling sequence example given in the section 3.1.2.3.

In addition, the `"environment"` object can contain the following optional attributes:

- `"configuration"`: Relative path to a possible configuration file that it is required by the S/W to run correctly. Note that if the **`--config`** common input parameter is provided in the CLI, it supersedes the value given in this attribute.
- `"activation"`: Relative path to the directory containing script(s) used to set up the environment, or to be run prior to the S/W execution. The script(s) shall work in the BASH shell and shall run without input argument.
- `"deactivation"`: Relative path to script(s) used to reset the environment to its initial state. The script(s) shall work in the BASH shell and shall run without input argument.

3.2.2.4 Modes

The `"modes"` object contains the list of S/W functions that can be run by the pipeline, when it calls the executable file.

For each S/W function, its name, its purpose and the list of input/output datasets to be read/saved shall be supplied. It allows the pipeline to build the S/W function input calls automatically, and to control if the expected output data files are correctly saved.

Each function listed in the `"modes"` object shall contain the following attributes and JSON arrays:

- `"name"`: The name of the function. It corresponds to the **`func1`** in the calling sequence example given in the section 3.1.2.3
- `"purpose"`: A short description of the purpose of the function
- `"inputs"`: A JSON array containing the list of the input RPW data file(s) in CDF format required by the function
- `"outputs"`: A JSON array containing the list of the output RPW data file(s) in CDF format produced by the function.

The purpose and the content of the `"inputs"` and `"outputs"` objects are detailed in the two next sections.

3.2.2.4.1 Inputs

The RODP requires information in order to identify the input data files of a given S/W function, and to build the corresponding CLI calling sequence. Especially, in order to give the right file path as input argument value, it needs to associate each CLI specific input parameter of the function (e.g., `--input_1`), with the corresponding expected RPW dataset.

This is the aim of the `"inputs"` array. It contains a list of JSON objects that give, for each related specific input parameter of the function, the ID of the RCS input RPW dataset as referenced in the RODP database.

Each specific input parameter object shall contain the mandatory attribute:

- `"identifier"`: The RPW dataset ID associated to the input data file, as referenced in the RODP database.

Note:

- With this mechanism, it is not needed to provide the full path to the input data files to the RODP. The attribute above is sufficient for the pipeline to identify a given RPW

dataset, and to build the path to the input data file from the information in the database. This path can be then passed to the CLI.

- **The teams are in charge to define the list of inputs for their S/W in the descriptor file. This work must be done in collaboration with the other teams and the ROC. Especially, the ROC shall supply as soon as possible information required to develop and maintain the RCS and their descriptor file, including an up-to-date list of the RPW datasets ID and versions used by the RCS.**

According to the example of CLI calling sequence in the section 3.1.2.3, the `"inputs"` array will be structured as followed:

```
"inputs":[
"input_1":{
                    "identifier":"RPW_DATASET_ID1”
                },
"input_2":{
                    "identifier":"RPW_DATASET_ID2"
                }
        ],
```

Where `"input_1"` and `"input_2"` objects corresponds respectively to the specific input parameter keywords `--input_1` and `--input_2`, and where `"RPW_DATASET_ID1”` and `"RPW_DATASET_ID2”` are the corresponding RPW dataset IDs, which allow the RODP to pass the input data file paths `/path/to/input1` and `/path/to/input2` in the CLI.

3.2.2.4.2 Outputs

In the same way, the RODP requires information in order to identify in the CLI specific input parameters (e.g., `--output_1`) related to the RCS outputs. It will help the RODP to build the right output file paths and names to be passed to the CLI for a given function. Additionally, it will also allow RODP to check that these output files are correctly saved at the end of the RCS execution.

As for `"inputs`, the `"outputs"` array gives the list of specific input parameters information as JSON objects, but for the RCS output data files. Each JSON objects includes the ID of the RPW dataset, as referenced in the RODP database, but also descriptive information related to this dataset. This information will be inserted/updated in the RODP database, each time the descriptor is loaded (i.e., during the registration step, see section 4.1.3).

Each JSON object listed in `"outputs"` shall contain the following mandatory attributes:

- `"identifier"`: The RPW dataset ID associated to the RCS output CDF, as referenced in the RODP. This ID shall be assigned in agreement with ROC, and following the convention defined in [RD6].
- `"name"`: a more human-readable name for the dataset, not necessarily unique.
- `"description"`: a short description of the dataset.
- `"level"`: the processing level of the dataset. The allowed values are “L1R”, "L2", "L2R", "L2S", "L3", "L4" and "AUX".

In addition, the `"outputs"` object can contain the following optional attributes:

- `"template"`: file name to reference a data schema used for the output generation, as an indication for the ROC team. It might be provided in the case of master CDF

- "reference": Any reference (paper, note, Web site) giving a description of the output.

Note that the RODP will perform an automated validation of the S/W descriptor file at each new release, in order to check that the file content is consistent with the ROC database information. In particular, it will verify that the datasets declared in the "modes" object are all defined in the descriptor and across S/W.

4 DESCRIPTION OF THE INTERFACE BETWEEN RODP AND RCS: RODP SIDE

This section presents the interface, but from the RODP point of view. Especially, the basic principle of the RCS automated deployment and execution mechanisms, to be operated by the pipeline, are described in details.

4.1 RCS automated deployment mechanism

The RCS installation and registration mechanisms inside the RODP are detailed.

When a new release of RCS is delivered by a team, the ROC will first install and run a copy of the S/W on its ROC development server for testing. If the testing phase has successfully ended, then the RCS can be deployed on the prod. server.

4.1.1 RCS release delivery process

The delivery shall be done by teams in charge, following the procedure described in [RD4].

There is no specific automated process realized by the RODP concerning the RCS delivery.

4.1.2 RCS installation process

The RCS installation on the ROC servers shall be performed by the ROC using dedicated scripts. The way the scripts will be run and what they will do are TBD.

Note that the installation step shall not include any compilation or configuration process. It requires the teams shall ensure that their S/W are ready to be used on the ROC servers before delivery. Especially, it means that the S/W directory shall contain all of the files required to run the program in the ROC servers, including the executable file, when the ROC team will deploy it.

Nevertheless, if RCS require compilations to work, the teams in charge shall also deliver all of the source files and libraries required for the compilation, as well as any installation script.

4.1.3 RCS registration process

Once the installation process is ended, the RODP shall launch the RCS registration. This step consists to read information in the descriptor file, check its integrity and its consistency, then insert the content into the RODP database. The RODP shall stop the registration if the descriptor file is not valid, or if the content inside is inconsistent with the database content.

The registration is required before running RCS into the RODP.

4.2 RCS automated execution mechanism

4.2.1 Principle

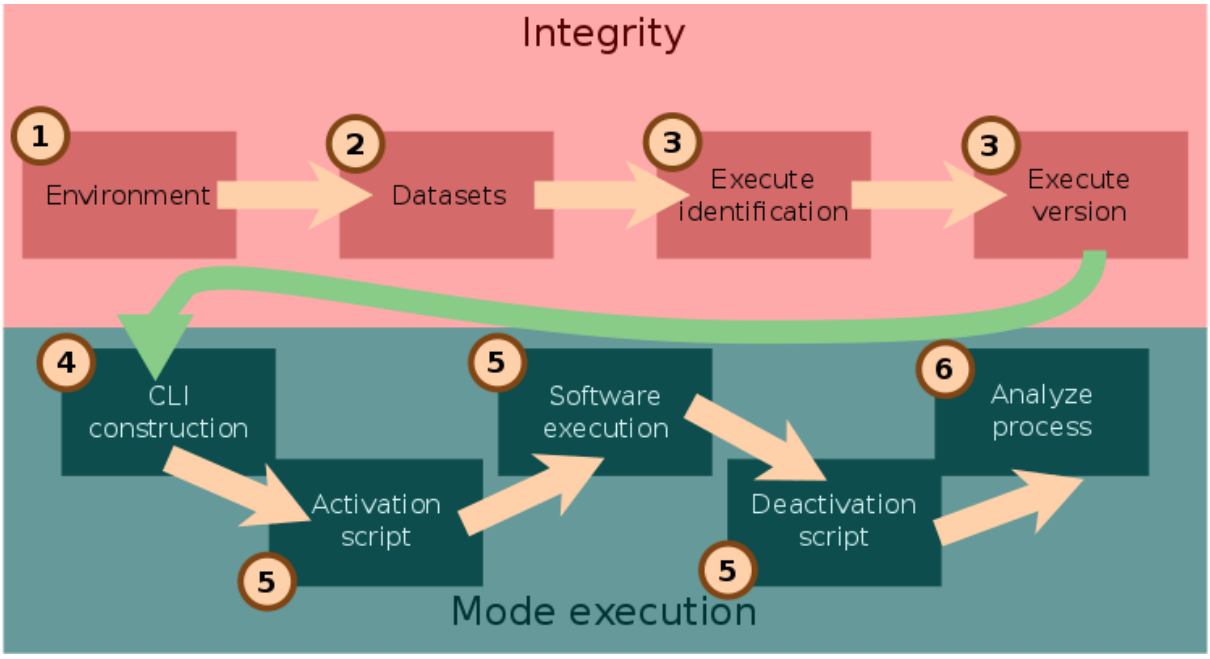

Figure 1. RCS execution workflow.

Figure 1 shows the tasks performed by the RODP just before, during and just after the execution of a given RCS. There are three main phases: S/W initialization, execution and post-verification.

These three phases can be divided into six main steps:

1. Checking the integrity of the environment defined by the S/W. The validity of the executable, activation/deactivation scripts, etc., is checked at this stage to be able to use the S/W for following steps.
2. Checking required I/O RPW datasets have correctly been registered into the ROC database.
3. The S/W identification is performed by the RODP (i.e., using the dedicated **`--identification`** keyword), in order to check the consistency with the S/W information registered in the database. Then, the same operation is done to check the S/W version (using the dedicated **`--version`** keyword).
4. Building the CLI calling sequence with the appropriate input parameters, depending of the RCS function.
5. Activation script *sourcing*, S/W execution with the CLI and deactivation script *sourcing*. Specific environment configuration of the S/W can be done through the activation and deactivation script. The latter permits to let the environment in a cleaned state, or make some cache clean up for example.
6. At the end of the execution, the output of the S/W is analysed to check that the expected data has been produced correctly.

If an error is encountered in any of these steps, the RODP raises an exception and the error is stored into the RODP database for investigations.

Details on each step are given in the following sections.

4.2.2 RCS automated validation

The validation of a RCS by the RODP is done in two steps:

- At the registration: the consistency between the required input datasets and the existence of their definition in the RODP database will be done. If the pipeline doesn't find the definition in the RODP database of an input dataset, or an inconsistency in the inter-dependency between the registered RCS is detected, the execution will be aborted.
- At the execution: the S/W will be called with the `--identification` keyword before running a function, in order to check that the information returned by the executable is as expected (i.e, as the information read from the *roc_sw_descriptor.json* and stored in the RODP database). In the case of a mismatch, the execution is stopped; otherwise the S/W is executed as explained in the following section.

4.2.3 Environment setup

The environment setup is only necessary for S/W requiring specific environment variables to work. The RODP will set the environment automatically before starting the related S/W execution.

A script, defined in the `"environment.activation"` attribute of the descriptor file, will be launched by the pipeline, without any argument. The script will be loaded (i.e., "sourced") before starting the S/W, and the *stdout* of the program redirected to */dev/null*.

For example in BASH:

```
$ source activate.sh 1>/dev/null
```

Where `activate.sh` is here the script path as read in `"environment.activation"`.

If the S/W ends without error, the corresponding deactivation scripts will be launched (if present), sourcing the script defined in the `"environment.deactivation"`field of the descriptor file.

It must be noticed that the RODP will raise an error if the files defined in the `"environment"` object are not found or not executable.

If the scripts edit the value of already existing and/or system environment variables, such as $LD_LIBRARY_PATH or $PATH, the teams in charge shall ensure that they do not overwrite the variable values, since they can be used by the pipeline or other S/W.

To avoid overwriting in BASH enter:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/one/more/path"
```

4.2.4 RCS automated execution

4.2.4.1 RODP calling sequence

The RODP module in charge of calling RCS must provide commands for automated launches of a given S/W function, and with specific input files. Used to run the S/W on a specific file with specific conditions to reproduce an isolated problem.

A typical use case for the command is:

```
$pop cawa run CAL-SOFT calibration1 –output-1 /path/to/output1 \
    --log /path/to/log --inputs --input-1 /path/to/input1 --input_2
/path/to/input2
```

Where `pop` is the executable of the RODP and `cawa` is the module in charge of RCS calls.

If the S/W is already registered into the RODP database with the name `CAL-SOFT`, the latter will run the S/W function, named `calibration1` in the example above, only if defined in the descriptor file. Passing argument for outputs and log paths as if it was the RODP while running, and all inputs and other flags following the `--inputs` flag are transmitted directly to the software executable.

Inputs arguments that must be given to the S/W executable must follow the `--inputs` flag, which must be placed after all other arguments and flags.

The `--inputs` flag is specific to the CaWa CLI, and must be only used by the pipeline in its CLI. The RCS CLI is not expected to include this flag.

4.2.4.2 Example

Figure 2 illustrates the execution schema of two functions “calibration_X” and “calibration_Y” of a given RCS by the RODP. These two functions are respectively in charge of producing “Output X-a/b” and “Output Y-a/b” data files from the input data files “Input X-a/b” and “Input Y-a/b”.

The pipeline’s choice to call these functions is made as a first step during the “Calibration mode selection” using the information provided inside the `"modes"` JSON object of the S/W descriptor file. Especially, it will allow the RODP to build the list of the input/output file paths to be passed/returned to/by the S/W, from the information saved in the RODP database.

The path of the log file and possibly the configuration file are provided by the RODP with the corresponding flags, respectively **`--log`** and **`--config`**.

If the S/W ends correctly, the pipeline will check that the outputs have been saved as expected, and then will insert the outputs meta-data (e.g., path, creation date, etc.) into the RODP database.

If an exception occurs during the execution, the S/W shall transmit all the relevant information through the *stderr*, as explained in the section 3.1.3.

Teams shall be aware that the pipeline will not call the S/W at all in the case where, at least one of the input files defined in the `"inputs"` of the descriptor is not found. It should be taken into account in the S/W architecture design, especially when defining the number of available modes.

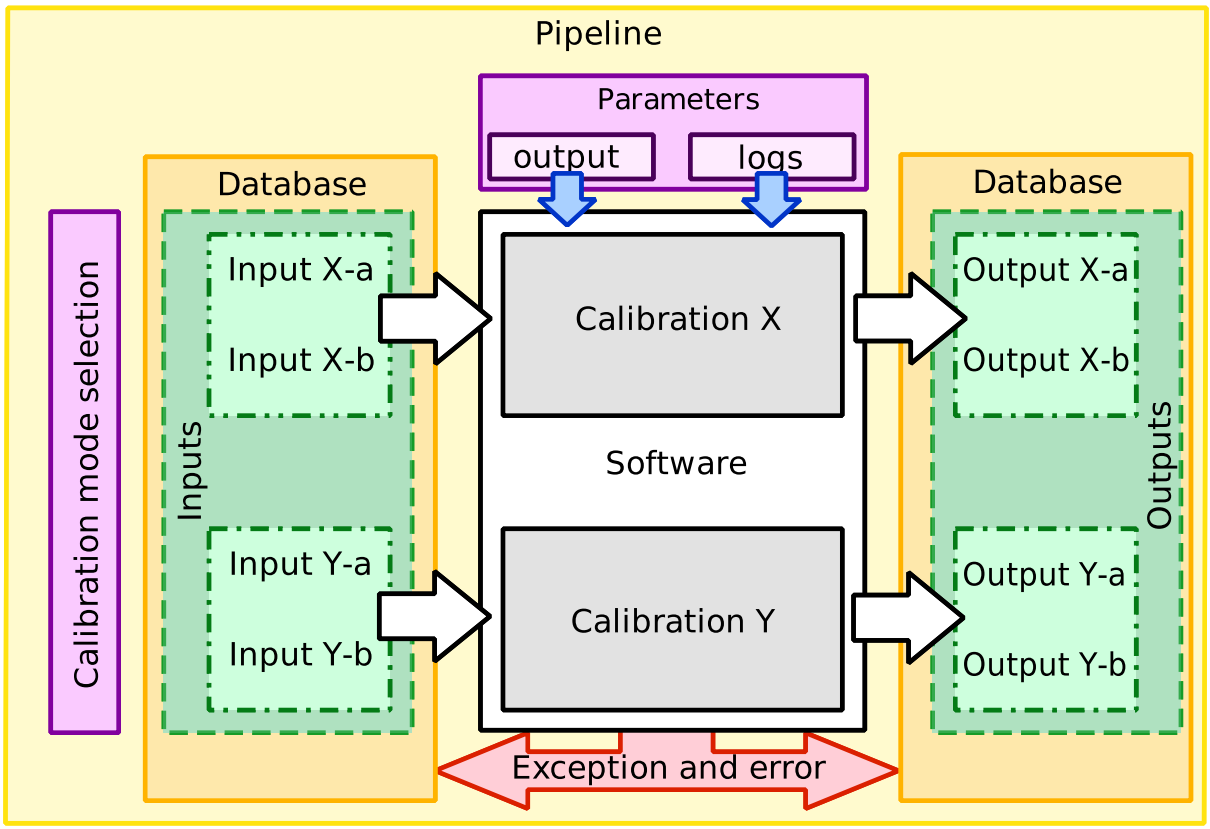

Figure 2. Detail of the execution of a RCS.

4.3 RCS testing mechanism

4.3.1 End-to-end test

The ROC team shall be able to easily check the RCS performances just after S/W deployment on the ROC server. To achieve this goal, each S/W shall be delivered with a set of input data files and their corresponding expected output files.

To perform the verification test, the ROC will run S/W using this input data set, and will compare the resulting data files with the expected outputs.

The details of how this end-to-end test will be done and the testing data set will be delivered to ROC are TBD.

4.3.2 Interface compliance test

If the S/W is not yet registered and ROC developers want to check the compliance with the RODP interface, i.e., present ICD, the following command can be launched:

```
$ pop cawa test /path/to/sw/root/directory calibration1 -output-1
/path/to/output1/                                                \
    --log /path/to/log --inputs --input-1 /path/to/input1 --input_2
/path/to/input2
```

Which will do exactly the same thing as the command showed in the section 4.2.4.1, but using only the S/W at the root directory provided as first argument of the `test` sub-command.

Inputs arguments that must be given to the S/W executable must follow the `--inputs` flag, which must be placed after all other arguments and flags.

The `--inputs` flag is specific to the CaWa CLI, and must be only used by the pipeline in its CLI. The RCS CLI is not expected to include this flag.

Teams that want to test the compliance with the RODP interface can use the dedicated tool described in [RD8].

5 RCS OUTPUTS CONVENTIONS

This section list conventions concerning the expected RCS outputs.

5.1 Science data convention

The RCS science data products shall comply the convention described in [RD5].

5.2 RCS log file convention

This section describes the convention concerning the log file, to be generated by the RCS during its execution at the ROC site.

5.2.1 General convention

Teams in charge shall follow the following log file related general convention:

- There shall be one log file per S/W.
- The S/W it-self shall create the log file. If an existing log file with the same name is found in the log directory, the S/W shall not create a new log file, but shall append new entries in this file.

Note that the ROC might need at some points to move an existing log file from the log directory.

5.2.2 File naming convention

The RCS log file shall follow the file naming convention:

[*rcs_name*].log

Where [*rcs_name*] is the name of the RCS in lower case characters, with only hyphens "-" or underscores "_" as separators (e.g., thr_calbar.log).

It must be noticed that, in practice, this naming convention is only applicable to the ROC team, since the log file is an input argument of the RCS.

5.2.3 Format and content

The log file shall be an ASCII format file. It shall contain enough information about the execution of the RCS, in order to allow the ROC and the teams in charge to monitor the S/W behaviour and to diagnose unexpected events.

Each line of the file shall start with the following prefix:

[*YYYY:MM:DD*] [*hh:mm:ss*] -- [*log_level*] – [*message*]

Where [*YYYY:MM:DD*] and [*hh:mm:ss*] are the date (*YYYY*=4-digits year, *MM*=2-digits month and *DD*=2-digits day of month) and time (*hh*=2-digits hours, *mm*=2-digits minutes and *ss*=2-digits seconds) of the log event. [*log_level*] is the level of severity of the log event (see table in the next section) and [*message*] is a one line description of the log event.

5.2.1 Log event severity levels

Table below gives the list of event severity levels to be used in the RCS log file.

Severity level	Definition	Actions to be performed by the RCS
DEBUG	Debug event (only used in debug mode)	No specific action
INFO	Normal event (e.g., RCS start/end times, routine tasks information, etc.)	No specific action
WARNING	Event that requires attention, but does not compromise the software execution or data production	No specific action
ERROR	Event that requires special attention, and compromises the data production or the software execution (e.g., unexpected values in the data, not input data file found, env. variable not well defined, etc.)	Stop the software execution and exit with an exception (error code 1)

Table 3. RCS log event severity levels.

6 APPENDICES

6.1 Example of a RPW S/W descriptor file

Here is an example of a RPW descriptor file for the THR Calibration software (THR_CALBAR) RCS.

```
{
    "identification": {
        "project": "ROC",
        "name": "THR-CALBAR",
        "identifier": "ROC-THR-CALBAR",
        "description": "The RPW TNR-HFR CALibration softwARe (THR-CALBAR)
produces RPW TNR-HFR calibrated science data"
    },
    "release": {
        "version": "1.1.0",
        "date": "2015-07-29",
        "author": "Antonio Vecchio",
        "contact": "antonio.vecchio@obspm.fr",
        "institute": "LESIA",
        "modification": "Update tnr_l2r_calibration function",
    },
    "environment": {
        "activation": "scripts/setup_thr-calbar_env.sh",
        "deactivation": "scripts/unset_thr-calbar_env.sh"
       "executable": "bin/thr-calbar-api",
        "configuration": "config/thr-calbar_configuration.json"

    }
    "modes": [
        {
            "name": "tnr_l2r_cal",
            "purpose": "Produce calibrated science TNR data file at analys
er level (L2r)",
            "inputs": {
                "input_l1_tnr": {
                    "identifier": "RPW_TNR_L1-SURV"
                }
            },
            "outputs": {
                "output_l2r_tnr": {
                    "identifier": "RPW_TNR_L2R-SURV",
                    "name": "RPW TNR L2R survey data",
                    "description": "RPW TNR L2R science data in survey for
RODP",
                    "level": "L2R",
                }
            }
        },
        {
            "name": "hfr_l2r_calibration",
            "purpose": "Produce calibrated science HFR data file at analys
er level (L2R)",
            "inputs": {
                "input_l1_hfr": {
                    "identifier": "RPW_HFR_L1-SURV"
                }
            },
```

```
            "outputs": {
                "output_l2r_hfr": {
                    "identifier": "RPW_HFR_L2R-SURV",
                    "name": "RPW HFR L2R survey data ",
                    "description": "RPW TNR L2R science data in survey for
RODP",
                    "level": "L2R"
                }
            }
        }
    ]
}
```

6.2 Examples of a CLI calling sequence

With the following configuration, the pipeline will call the S/W for the calibration mode `tnr_l2r_cal` as (assuming that the environment as been setup previously and that the path to the S/W directory is contained in the `$SW_ROOT` variable):

```
$$SW_ROOT/bin/thr-calbar-api tnr_l2r_cal --input_l1_tnr /path/to/input_cdf
\
    --output_l2r_tnr /path/to/output_cdf --log /path/to/log/directory \
    --config $SW_ROOT/config/configuration.file
```

If the S/W ends without error, then the RODP will search for a file at `/path/to/output_cdf`.

If an error occurs, the S/W must stop with an error code greater than 0 and a relevant message in *stderr*. If the error can be caught by the RODP, it will store not necessarily useful information.

For the identification command, the S/W will return the following JSON message in the *stdout*:

```
$ $SW_ROOT/bin/thr-calbar-api --identification
{
    "project":"ROC",
    "name":"CALBAR",
    "identifier":"RPW-TEST-CALBAR"
}
```

, and for the version command:

```
$ $SW_ROOT/bin/thr-calbar-api --version
{
    "version":"1.1.0"
}
```

6.3 ROC S/W descriptor file validation

A JSON format file, as for XML, can be validated against a schema. A JSON schema is following the specifications from http://json-schema.org/draft-04/schema. Dedicated documentation can be found at https://spacetelescope.github.io/understanding-json-schema.

The JSON schema for the ROC S/W descriptor file is described below. It will be used by the ROC to automatically validate a S/W descriptor file after delivery.

```
{
    "$schema":"http://json-schema.org/draft-04/schema#",
    "definitions":{
        "input":{
            "type":"object",
            "patternProperties":{
                "^[A-Za-z][\\w-]+$":{"$ref": "#definitions/input_dataset"}
            },
            "additionalProperties":false
        },
        "output":{
            "type":"object",
            "patternProperties":{
                "^[A-Za-z][\\w-]+$":{"$ref":
"#definitions/output_dataset"}
            },
            "minProperties":1,
            "additionalProperties":false
        },
        "mode":{
            "type": "object",
            "properties": {
                "name": {
                    "type": "string",
                    "pattern": "^[A-Za-z][\\w-]+$"
                },
                "purpose": {"type": "string"},
                "inputs": {"$ref": "#definitions/input"},
                "outputs": {"$ref": "#definitions/output"}
            },
            "required": ["name", "purpose", "inputs", "outputs"],
            "additionalProperties": false
        },
        "release": {
            "type": "object",
            "properties": {
                "author": {"type": "string"},
                "date": {
                    "type": "string",
                    "format": "date-time"
                },
                "version": {
                    "type": "string",
                    "pattern": "^(\\d+\\.)?(\\d+\\.)?(\\d+)$"
                },
                "file": {"type": "string"},
                "institute": {"type": "string"},
                "reference": {"type": "string"},
                "url": {
                    "type": "string",
                    "format": "uri"
```

```
                },
                "contact": {
                    "type": "string",
                    "format": "email"
                },
                "modification": {"type": "string"}
            },
            "required": ["author", "date", "version"],
            "additionalProperties": false
        },
        "output_dataset": {
            "type": "object",
            "properties": {
                "identifier": {
                    "type": "string",
                    "pattern": "^[\\w-]+$"
                },
                "name": {"type": "string"},
                "level": {"enum": ["L0", "L1", "L2R", "L2S", "L2", "L3",
"AUX", "LL01", "LL02", "HK"]},
                "description": {"type": "string"},
            },
            "required": ["identifier", "name", "level", "description"],
            "additionalProperties": false
        },
        "input_dataset": {
            "type": "object",
            "properties": {
                "identifier": {
                    "type": "string",
                    "pattern": "^[\\w-]+$"
                },
            },
            "required": ["identifier"],
            "additionalProperties": false
        }
    },
    "type": "object",
    "properties": {
        "identification": {
            "type": "object",
            "properties": {
                "project": {"type": "string"},
                "name": {"type": "string"},
                "identifier": {"type": "string"},
                "description": {"type": "string"}
            },
            "required": ["project", "name", "identifier"],
            "additionalProperties": false
        },
        "release": {"$ref": "#definitions/release"},
        "environment": {
```

```
            "type": "object",
            "properties": {
                "activation": {"type": "string"},
                "deactivation": {"type": "string"}
            },
            "additionalProperties": false
        },
        "executable": {"type": "string"},
        "configuration": {"type": "string"},
        "modes": {
            "type": "array",
            "items": {"$ref": "#definitions/mode"}
        }
    },
    "additionalProperties": false,
    "required": ["identification", "release", "environment", "executable",
"modes"]
}
```

7 LIST OF TBC/TBD/TBWS

TBC/TBD/TBW			
Reference/Page/Location	Description	Type	Status

8 DISTRIBUTION LIST

LISTS See Contents lists in "Baghera Web": Project's informations / Project's actors / RPW_actors.xls and tab with the name of the list or NAMES below		Tech_LESIA
		Tech_MEB
		Tech_RPW
		[Lead-]CoIs
		Science-CoIs

INTERNAL

LESIA CNRS		

LESIA CNRS		

EXTERNAL (To modify if necessary)

CNES		C. FIACHETTI
		C. LAFFAYE
		R.LLORCA-CEJUDO
		E.LOURME
		M-O. MARCHE
		E.GUILHEM
		J.PANH
		B.PONTET
IRFU		L. BYLANDER
		C.CULLY
		A.ERIKSSON
		SE.JANSSON
		A.VAIVADS
LPC2E		P. FERGEAU
		G. JANNET
		T.DUDOK de WIT
		M. KRETZSCHMAR
		V. KRASNOSSELSKIKH
SSL		S.BALE

AsI/CSRC		J.BRINEK
		P.HELLINGER
		D.HERCIK
		P.TRAVNICEK
IAP		J.BASE
		J. CHUM
		I. KOLMASOVA
		O.SANTOLIK
		J. SOUCEK
		L.UHLIR
IWF		G.LAKY
		T.OSWALD
		H. OTTACHER
		H. RUCKER
		M.SAMPL
		M. STELLER
LPP		T.CHUST
		A. JEANDET
		P.LEROY
		M.MORLOT