EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

Common Pipeline Library
Technical Developers Manual

VLT-MAN-ESO-19500-3349

Issue 0.9

Date 2004-08-18

– DRAFT –

– DISTRIBUTED FOR INTERNAL USE ONLY –

Prepared:	D. J. McKay	2004-08-18	.....
	Name	Date	Signature
Approved:	?. ?????		.....
	Name	Date	Signature
Released:	?. ?????		.....
	Name	Date	Signature

This page was intentionally left blank

Change record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
0.1	08/03/2004	All	Skeleton of sections
0.2	12/05/2004	All	Bulk of initial text inserted
0.3	17/05/2004	All	Updated version with additional text
0.4	21/05/2004	All	Internal release for review
0.5	17/06/2004	All	Document number added
0.6	30/06/2004	All	Changed name from EsoPipe to EsoRex
0.7	13/07/2004	All	Implemented informal review comments
0.8	03/08/2004	All	Added comments from NPFM and FFTW documentation
0.9	18/08/2004	All	Added index and some minor corrections

This page was intentionally left blank

Contents

1 Introduction

Astronomy has come a long way since the days of photographic plates, let alone visual observations. Today, telescopes sport an impressive array of sophisticated instrumentation. Inherent in each of these is a level of sophistication paired with challenging quantities of data output. Everything is handled digitally, and a high degree of automation is the only way to cope with the sheer volume of scientific output.

In response to these trends in astronomical instrumentation, the prevalence of automated data processing software — referred to as *pipelines* — is on the increase. The pipelines themselves comprise atomic data processing components called *recipes*, which perform a given operation, within the overall sequence of data processing that constitutes the pipeline itself.

In response to this and the high-level requirements of the VLT, the European Southern Observatory has furthered this development by creating a coherent and unified environment in which such software can be created.

1.1 Purpose

This document is aimed at the software developer who wishes to produce software to work within the ESO pipeline environment. It describes the terms and nomenclature, provides a comprehensive list of requirements and offers as step-by-step tutorial on generating a prototype pipeline recipe.

The goal of providing a uniform software development environment for data reduction pipelines is to reduce the cost of development and maintenance. This document aims to make the use of this environment as easy as possible, allowing developers to understand all aspects of the software structure and to quickly arrive at the stage where they can concentrate on the algorithms, and not just the infrastructure to incorporate it into the overall software system.

1.2 Scope

The current specifications address aspects related to the Data Flow System which must be considered by a VLT Instrument Consortium in the context of developing and delivering VLT pipeline recipe software. Software written internally within ESO shall also follow the specifications described herein.

1.3 About this document

This document provides a general introduction to the concepts involved in developing an ESO-compliant pipeline recipes. It also serves to act as a reference point to other relevant documentation, as well as describing how the required software may be obtained.

However, this document deals primarily with two main areas. The first is that of the requirements of the ESO pipeline environment. In order to maintain a large number of different pipelines for an array of instruments, it is necessary to place standards on the way in which pipeline software is written, tested and delivered.

Secondly, the document provides a detailed step-by-step description of the development of a pipeline recipe. This is based upon available skeleton software, which may be transmuted into a fully operational system with

relative ease. The description of this process, in addition to careful explanations of the rationale of this system, are described herein.

1.4 Acknowledgements

In June 2001, N. Devillard and R. Palsa first proposed a common software library in order to ease and accelerate the development efforts for the different VLT instrument pipelines. This software library, called *Common Pipeline Library* (CPL), would essentially be built up from already existing code. In particular, the *Eclipse* library (used for ISAAC and NACO pipelines) and concepts of the VIMOS data reduction software would be the main pillars of the CPL software.

In September 2001, M. Peron formed a CPL project team, consisting of N. Devillard and Y. Jung (working for ISAAC, NAOS/CONICA), together with R. Palsa and C. Izzo (working for VIMOS, FORS1/2), as well as P. Ballester and C. Sabet from the VLTI pipeline project. K. Banse served as mediator and chairman. In September 2002, N. Devillard left the CPL project and M. Kiesgen became a member of the team by the end of that year. Finally, L. Lundin, A. Modigliani and D. J. McKay joined the CPL team in the course of the year 2003 . A preliminary version of the CPL was released in May 2002. Building on this basic version, the first official release of the CPL was made available to the public by ESO in December 2003.

The CPL recipe development prototype template was developed by Y. Jung.

1.5 Contacts

If there is a need to contact the CPL team, then the relevant details may be found on the “Contact” page of the CPL website: `http://www.eso.org/cpl/`. See also Section 5.

1.6 Reference Documents

The following documents are referenced in this work.

[1] VLT Data Flow System Operations Model for VLT/VLTI Instrumentation VLT-PLA-ESO-19000-1183
[2] VLT Data Flow System Specifications for Pipeline and Quality Control VLT-SPE-ESO-19600-1233
[3] Data Flow for VLT/VLTI Instrument Deliverables Specification VLT-SPE-ESO-19000-1618/2.0
[4] DFS Pipeline & Quality Control – User Manual VLT-MAN-ESO-19500-1619
[5] ESO DICB – Data Interface Control Document GEN-SPE-ESO-00000-0794
[6] International Standard ISO/IEC 9899: 1999(E) — Programming languages — C
[7] VLT Common Pipeline Library User Manual VLT-MAN-ESO-19500-2720
[8] The ESO Data Interface Definition — `http://www.eso.org/dicb/`
[9] The Common Pipeline Library Reference Manual — `http://www.eso.org/cpl/reference/`
[10] Definition of the Flexible Image Transport System (FITS) — NOST 100-2.0, NASA/Science Office of Standards and Technology, 1999
[11] The Gasgano User’s Manual VLT-PRO-ESO-19000-1932
[12] The CPL website — `http://www.eso.org/cpl/`
[13] The Splint website — `http://www.splint.org/`

As a suitable introduction, readers are encouraged to read the section entitled *Software development with CPL* in the *VLT Common Pipeline Library User Manual* [7].

1.7 Abbreviations and Acronyms

The following general abbreviations are used within this document.

CONICA	COudé Near Infrared Camera Array
CPL	Common Pipeline Library
CPLVS	CPL Validation Software
CVS	Concurrent Version System
DET	DETector
DFO	Data Flow Operations (group)
DFS	Data Flow System
DHS	Data Handling Server
DICB	Data Interface Control Board
DICD	Data Interface Control Document
DIT	Detector Integration Time
DO	Data Organiser
DRS	Data Reduction System
ESO	European Southern Observatory
FDR	Final Design Review
FITS	Flexible Image Transport System
FORS	FOcal Reducer/low dispersion Spectrograph
FTP	File Transfer Protocol
GASGANO	A host application for running CPL-based recipes
GUI	Graphical User Interface
HTML	HyperText Markup Language
IEC	International Electrotechnical Commission
ISAAC	Infrared Spectrometer And Array Camera
ISO	International Organisation for Standardisation
LSS	Long Slit Spectroscopy
LTDL	LibTool Dynamic Loader (software)
MIDAS	Munich Image Data Analysis System
MOS	Multi Object Spectroscopy
NACO	**NA**OS-**CO**NICA
NAOS	Nasmyth Adaptive Optics System
NOST	NASA/Science Office of Standards and Technology
PAC	Preliminary Acceptance Chile
PAE	Preliminary Acceptance Europe
PAF	PArameter File
PDF	Portable Document Format
PDR	Preliminary Design Review
POSIX	Portable Operating System Interface (standard)
PRO	PROduct (keyword indicator)

PSO	Paranal Science Operations (group)
QC	Quality Control
QFITS	(A FITS access library used by CPL)
RB	Reduction Block
RBS	Reduction Block Scheduler
SDK	(CPL recipe) Software Development Kit
SEG	System Engineering Group
SOF	Set Of Frames
TFITS	Table FITS
UT	Unit Telescope
VIC	VLT Instrument Consortium
VIMOS	VIsible Multi-Object Spectrograph
VLT	Very Large Telescope
VLTI	Very Large Telescope Interferometer

1.8 Glossary

The following terms are used repeatedly throughout this document. The descriptions given here are assumed, unless specified otherwise. Terms shown in it italics have their own entries.

CVS — Concurrent Version System. This freely-available software allows developers to track source code and perform version control. It is assumed in this document, and the recipe template has been produced with this package in mind.

Dynamic library — Unlike a static library, a dynamic library does not need to be linked to an executable at link time. Instead, an executable may chose to load a dynamic library during run time. Additionally, providing that the interface is defined and maintained constant, it is not necessary to rebuild the executable or the dynamic library, should the other's internals change.

This concept allows for a very flexible development environment, and because of this, it has been chosen as the method for implementing *pipeline recipes*.

EsoRex — EsoRex is a stand-alone application that may be used to run recipes. Unlike *GASGANO*, it is run as a command-line utility. Because the *recipe* itself is a dynamic library, it is not possible to run it by itself. Thus, EsoRex provides one way of accomplishing this, in addition to automating configuration parameters, and so on. Details about EsoRex are given in Section A.

GASGANO — Gasgano is an explorer-style GUI software tool for managing and viewing data files produced by the ESO VLT. The primary purpose for Gasgano is to provide a user-friendly tool capable of presenting an organised, manageable view over the large data sets generated by the VLT and the other ESO astronomical facilities.

Pipeline — A pipeline is a generic term that refers to a collection of data reduction tasks, that have been automated in some way. The pipeline software should be capable of running without human supervision to accomplish a sequence of processing steps. Within the context of the ESO pipeline environment, these processing steps are referred to as *recipes*.

Plugin — A plugin is a unit of compiled software that may be incorporated into another executable at run time. In the case of the ESO software environment, plugins have been implemented as *dynamic libraries*. Each *recipe* is in turn a plugin.

PRO_ keyword — A "product" keyword. These keywords are, amongst various other things, used to classify the *product*.

Product — An output file from a *recipe*. Typically, products will be either FITS or TFITS files.

Recipe — A recipe is a data reduction operation that has been implemented as an single software unit. Within the context of the ESO pipeline environment, a recipe is implemented as a *plugin*.

SOF — The *Set of Frames* (SOF) is the input specification file that is given to a recipe at run time. It defines the files that are to be processed by the recipe, and provides additional tags that allows the files to be identified (according to data type).

1.9 Conventions for this document

Examples of items to be shown of the command line are indicated using fixed-width fonts. The generic shell prompt is indicated with a percent-sign (%). For example:

```
% mkdir /home/user/
```

References to other documents are indicated by placing the reference number in square brackets ("[]"). The list of the reference documents themselves may be found in Section 1.6.

2 Overview of Pipeline Software

Like all major astronomical facilities, the telescopes operated by ESO require a large amount of automatic data processing in order to put it into a form that it can be easily dealt with by astronomers. This data processing is usually carried out by complex software that implement sophisticated algorithms and deal with vast quantities of data. They also typically run on "remotely" located hardware, unattended for the most part by human operators. As such, the requirement for robustness, automation and precision is of the utmost importance.

Due to the huge scale of ESO's Paranal and La Silla operations, there are many instruments, collecting data at a multitude of different operating modes; from fibre- and slit-spectrographs to imagers to interferometry. Most of these require a software pipeline to handle their initial data processing requirements. Currently, such pipelines are operational for eight instruments, and have been written on the basis of three different software packages leading to considerable overlap and duplication of functionality. With the number and complexity of ESO instruments set to increase, there has been a growing demand to try to consolidate the common functionality of these pipelines.

The development of the pipeline environment has been driven by the operational requirements of the VLT. However it is also intended that the pipeline software is made available to external users.

2.1 The *Common Pipeline Library*

The Data Flow System (DFS) group of ESO has developed the CPL with the aim to increase the functionality available to pipeline developers, to provide a more rigorously-tested set of functions within this available set and, hence, ease the development and maintenance load on the software developers who are supporting the ever increasing family of ESO instrumentation.

The CPL consists of a set of ISO-C libraries, which have been developed to standardise the way VLT instrument pipelines are built, to shorten their development cycle and to ease their maintenance. The CPL was not designed as a general purpose image processing library (although it accomplishes that task too) but, rather, its design reflects two primary ESO requirements. The first of these is to provide a unified interface to the VLT pipeline infrastructure. The second, is to provide a set of software tools, which make possible the rapid construction of astronomical data-reduction tasks. The CPL provides:

- Many useful data types (property lists, images, tables, ...).
- String and file utilities.
- Medium level data access methods (a simple data abstraction layer).
- Image and signal processing capabilities.
- Standard implementations of commonly used data reduction tasks.
- Support for dynamic loading of recipe modules.
- Standardised application interface for pipeline recipes.

For more general information about the CPL, please refer to the *VLT Common Pipeline Library User Manual*.

2.2 The "instrument-to-scientist" path

Every instrument on the VLT (or VLTI) has a data reduction "pipeline", which is managed as part of the DFS. This pipeline is used to pre-process the raw data as extracted from the instrument and deliver it to the the end repository.

2.3 Architecture of a pipeline

The DFS pipeline has two primary components: data management and data processing.

2.3.1 Data management

Data management is independent of the instrument being used and is implemented as part of the DFS infrastructure. Data management may be interactive (for example, an external user who runs GASGANO) or automatic (as part of the operational environment). The data management facility performs the following functions:

- data classification,
- data grouping,
- data association, and
- pipeline recipe selection.

2.3.2 Data processing

Data processing is instrument specific and the data processing algorithms are contained within dynamic libraries called recipes.

Each pipeline recipe accepts as input a list of frames (typically image files and calibration data) to be processed. Additionally, it accepts various configuration parameters to control the data reduction process. The recipe must be built in a way that it may be run by itself for testing and manual data processing.

While it is possible to run recipes in a stand-alone mode, it is typical to string them together, so that the successful completion of one will trigger the execution of the next one, propagating until the data have been completely reduced. This is what is done within the DFS infrastructure as part of the automatic operation of the complete on-line DFS.

Note that the pipeline recipes work in a way so that they can be operated without requiring access to any databases, visualisation or plotting capabilities.

The creation of recipes that fulfils the above criteria is, of course, the subject of subsequent sections of this document.

For more information about the DFS Pipeline concept, please refer to the *Data Flow for VLT/VLTI Instrument Deliverables Specification* [3]. Details of the on-line, off-line and desktop data reduction environments are discussed there, and should serve as a useful introduction to the general overview of the system.

2.4 GASGANO, EsoRex and running recipes

ESO provides two stand-alone host applications for running recipes. As has been explained above, the recipe itself is not a stand-alone executable, but rather a dynamical library — or plugin — that is loaded and executed by a host application.

GASGANO is one of these applications, and it implements a graphical user interface. See the Section 1.8 for a summary of the GASGANO program. There is also a complete user manual available [11].

EsoRex is a command-line driven utility which also may be used to run recipes. Although it may be easily used as an interactive tool, it is probably most useful in the scripting environment, where it may be embedded into data reduction scripts for the automation of processing tasks.

The basic idea of EsoRex, is that it takes various arguments specified on the command line, and uses them to display the recipe environment, to query an individual recipe or the run the recipe itself. These arguments fall into four categories, which are listed here, and will be described thereafter:

- EsoRex command line options
- The recipe
- Recipe options (if a recipe was specified)
- SOF file (if a recipe was specified)

Thus, the usage for the package is as follows:

```
esorex [esorex-options] [recipe [[recipe-options] SOF]]
```

The command line options may be obtained using the command:

```
% esorex --help
```

These options are also listed in Appendix A.3.1.

The recipe name, is that given by the recipe developer. As this document deals with the creation of recipes, the manner in which are recipe is created and provided will be explained later (see Section 4).

Like EsoRex itself, the specified recipe may also accept input options. That is, the recipe may have command line options defined for it. These may be used to control the behaviour of the recipe or to set various input parameters. As the recipe is specified as a command-line argument itself, any switches that come after it in the command-line argument sequence are regarded to be pertinent to the recipe, rather than toe EsoRex.

The input file, also called an *SOF* (Set of Frames) file, contains a list of the input data files, along with classification tags. The description of the format and use of these SOF files is given in Section 3.5.2.

Throughout the remainder of this document, EsoRex will be used to illustrate the operation of pipeline recipes, due to its concise output.

A more complete description of EsoRex is given in Appendix A beginning on page 53. The appendix collates all the common operations that are performed with EsoRex from the point of view of the EsoRex application itself, as opposed the the sporadic examples given in the main text of this document, which are designed to illustrate features of the recipes and their development.

3 Requirements for a New Recipe

This section describes the requirements for writing any new recipe that is to be incorporated into the DFS.

3.1 The recipe concept

The basic concept of the pipeline recipe is to detach the actual processing code from the interface environment used to run it. That is to say, the recipe contains the processing algorithm, and is written to conform to a standard interface (the recipe/plugin interface of CPL). A host application, that wishes to execute the recipe algorithm, conforms to the other side of this recipe/plugin interface, and may then call any CPL recipe at will.

The interface itself defines setup functions, name, calling procedure and expected arguments. Once established, any recipe may be called, and the host application will be able to determine the details by predefined CPL function calls.

A good example of this detachment is given by the programs EsoRex and GASGANO (see Section 1.8 for a brief summary of the two). One is a graphical user interface, and the other is command line driven. Both have a completely different look-and-feel to them, but both are capable of calling the same recipe and achieving the same result.

The recipe developer needs only to provide the CPL plugin/recipe interface, and need not concern him- or herself with the details of graphics, command line parsing, or whatever.

As this document does not concern itself with the details of writing host applications, the remainder of this section will instead concern itself with the details of the development of the recipe itself.

3.2 Standards

3.2.1 Operating system

ESO pipelines shall be able to execute without error on the following platforms:

- HP-UX 11
- Solaris 2.8, and,
- Linux (glibc 2.1 or later),

3.2.2 Language

All pipeline code developed by the consortium shall be ISO/IEC 9899:1999(E) compliant and compile without warning under at least its principal compiler, which is `gcc` version 3.2.

GCC should run with the maximum warning level enabled. This equates to the following compiler flags

```
gcc -ansi -pedantic -Wall
```

Any warnings which cannot be eliminated shall be explicitly commented in the code.

3.2.3 POSIX compliance

ESO pipelines shall be POSIX compliant.

3.2.4 Static code checking

Additionally, ESO recommends the use of static code checking on all delivered source code. For example, the program `splint` may be used to check ISO/IEC 9899:1999(E) and POSIX compliance, in addition to providing comprehensive static code-path checking [13].

3.2.5 Tools

ESO pipelines will make use of the following utilities for the compilation, linking and execution of pipeline software:

- An ISO-C compiler (preferably GNU gcc v3.2 or better)
- GNU autoconf (version 2.57 or better)
- GNU automake (version 1.7.2 or better)
- GNU libtool (version 1.4.3)

Thus, developers of pipeline recipes shall also use these utilities.

EsoRex and GASGANO will be used to test any delivered recipes. Thus, developers shall ensure that any produced recipes test correctly in the environment of these utilities.

3.2.6 Libraries

ESO pipeline software shall use the following libraries, as the implementation for recipe-to-host-software interaction.

- CPL (version 1.1 or later — available from `http://www.eso.org/cpl/`)

ESO pipeline software shall use CPL functions for the following types of operations:

- image manipulation,
- table handling,

- matrix manipulation,
- string operations,
- property list management,
- FITS file interaction, and
- statistical, arithmetic and conversion operations.

The CPL library makes use of the following libraries (all of which are distributed as part of, or in association with, CPL).

- QFITS (version 4.3.7 — from `http://www.eso.org/projects/aot/qfits/`)
- LTDL (GNU libtool (version 1.4.3) — distributed with CPL)
- CEXT (distributed with CPL)

ESO pipeline software should use CPL functions wherever possible for internal operations.

Code written for ESO pipelines may not use any other third-party library without the express permission of ESO.

3.2.7 Coding style/standard

All pipeline software shall conform to the ESO coding standard and ESO style guidelines. In particular, the following requirements exist:

- The developed source files shall contain a prologue that tells what is contained within that file: a description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names.
- Every newly implemented function called by `rrrecipe_reduce` (See Section 3.7.5) should have a header comment block containing a detailed and comprehensive description of the function role. Pseudo-code is recommended (structured summary of the algorithm, using both usual C control structures and natural language), except for very simple algorithms.
- All variables shall be explicitly declared and commented to describe each variable's role.
- Functions shall be explicitly declared and commented to describe the function's return type and the purpose of all arguments.
- A default case shall be defined in every switch structure.

Additionally, ESO makes the following non-compulsory recommendations.

- Although there is no maximum length limit for source files, files with more than about 1000 lines are generally regarded to be cumbersome to deal with. Source files should therefore be generally kept within this limit.
- The percentage of comments should be greater then 30% (for every function, number of lines of comments / total number of non-blank lines).
- Comments should be significant and relevant; they should explain but not repeat what can be already expressed by the objects names. It is not necessary to duplicate in words the meaning of the name of a variable/type/argument (for example: `int i; /* An integer */`).
- The number of function arguments (input/output parameters) should not exceed five. In cases where this is not possible, ESO recommends the use of a data structure to pass the data.
- All variables should be initialised just when declared.
- The `while`ăvariable should be initialised before entering the `while` loop; its initial value should be checked before entering the loop.

3.2.8 Directory structure

Each instrument pipeline shall be organised according to the directory hierarchy provided by the Recipe Template (an instrument pipeline code-template — see Section 4).

The top level directory shall be called as per the short form of the instrument name, which is usually an abbreviation or acronym (e.g. NACO). In addition, the letter `p` shall be appended to the name, to indicate that it is a pipeline development area. As per common UNIX directory naming practice, the entire name shall be in lowercase. For example, `nacop` is the directory for the NACO pipeline software.

Underneath this directory, there shall be all the files as provided by the recipe template.

3.2.9 Pipeline code

Code that is common to the pipeline software (that is, common to all recipes) shall be placed in a sub-directory named according to the instrument (but without the trailing letter `p`). Here, the `_dfs`, `_pfits` and `_utils` files are located. Refer to Section 4.4.3 for the details.

3.2.10 Recipe code

There shall be one `recipename.c` file per recipe that exists in the pipeline. These files shall be located in the directory `recipes` in the pipeline development hierarchy.

Typically there is one module per individual recipe, in addition to modules containing all the code that is common to the instrument recipe collection as a whole. However, in the case where there is a lot of code (more than several thousand lines), then it is permissible to split the recipe into multiple modules. However, `recipe.c` is reserved for the top level functions, and should contain calls to:

- rrrecipe_create()
- rrrecipe_exec()
- rrrecipe_destroy()
- rrrecipe()
- rrrecipe_reduce()
- rrrecipe_save()

Descriptions and examples are given in Section 4.4.8.

Additional modules within the recipe package should be named `recipename_functiongroup.c` and `recipename_functiongroup.h` as appropriate. See Section 4.4.8 for an example, along with a description of how this works in the recipe template.

3.2.11 File naming conventions

File names for source code and directories shall follow the examples set out in Section 4.

3.2.12 Build utilities

The developed pipeline code is expected to make use of the following build packages, in order to promote code portability amongst different UNIX platforms.

- Automake
- Autoconf
- GNU make

The expected versions of these software packages are given in Section 3.2.5.

3.3 Accounts

The software written for an instrument pipeline/recipe shall require no special account or privileges over those of a standard UNIX user account, in order for it to be run in a local directory with a package such as EsoRex (Section A).

3.4 Delivered recipe

The following items describe the expectation regarding the delivered pipeline code. See also the *Data Flow for VLT/VLTI Instrument Deliverables Specification* [3].

3.4.1 Source code

All source code shall be provided.

3.4.2 Distribution format

The code shall be delivered in the form of a tar-archive (created using the `tar` or `gtar` utility or equivalent) and compressed with the `gzip` compression utility. The name of this compressed tar-archive (commonly referred to as a "tarball") shall conform to the format: "*instrument-name*`p-X.Y.Z.tar.gz`", where `X`, `Y` and `Z` are the major, minor and patch version numbers.

3.4.3 Accessing external libraries

Generally, the provision of CPL caters to all the needs of recipe development. Functionality beyond this is most commonly confined to the specific needs of the recipes themselves, and thus is coded directly as part of the recipe source code base.

However, if the recipes developed for an instrument require external libraries, then the following requirements are imposed by ESO.

- Developed instrument recipes shall not use any external libraries, except as a result of agreed negotiation between the developing consortium and ESO.
- Any external library shall be packaged as part of the recipe package. That is, there shall be no external dependency of such a library, and the library shall be included in its entirety within the delivered recipe package.
- All calls to the external library shall be made from so called "wrapper functions" within the recipe module `iiinstrument_utils.c` (or its equivalent). See Sections 4.4.6 and 4.4.7 for further information and examples.
- The library shall be configured as part of the standard "configure" of the package as a whole.
- The library shall be built by the Makefiles that build the rest of the recipes in the instrument package.
- There shall be no maintenance requirements outside that of the recipe environment itself. That is to say, that ultimately the instrument recipe package shall be wholly self-contained.

One foreseen library that may be necessary for some recipes is that which can perform fast Fourier transforms. In this case, ESO permits the use of the FFTW package. An example of the use of the FFTW library is given in Appendix B.

3.4.4 Accessing external executables

Developed instrument recipes shall not use any external executables, except as a result of agreed negotiation between the developing consortium and ESO. While it is technically possible to call external executables from within the recipe environment (such as via `system()` calls, etc.), this often results in complications with process management, which often leads to undesirable side-effects, when run within the production environment. Therefore, ESO provides no support for running external executables from within recipes.

While acceptable solutions do exist, they are outside the scope of this document, and the requirement is thus stated that developers wishing to pursue this line of technical solution are required to contact ESO to negotiate such a solution before proceeding.

3.4.5 Makefiles, installation

The tarball shall contain all necessary build utilities. These should be already in place if the recipe template is used (as per Section 4).

3.4.6 Version documentation

In each release of the pipeline software, the provision of the following files is expected.

- AUTHORS
- BUGS
- COPYING
- ChangeLog
- INSTALL
- NEWS
- README
- TODO

The files, or at least templates/place-holders for them, are provided as part of the recipe template described in Section 4.

3.4.7 Quality assurance and compliance

The exact test cases for the delivery of any given recipe are defined at the time of the *Preliminary Design Review* (PDR). The are assessed at the time of *Preliminary Acceptance Europe* (PAE). The details of these stages, and how they relate the overall delivery process, are defined in the *Data Flow for VLT/VLTI Instruments Deliverables Specification* document [3].

The compliance factors, however, shall include all the requirements within this document, except in the case of explicit negotiation of a concession between the consortium and ESO.

3.5 Recipe Input

Recipe input shall consist of files and passed parameters. They consist of configuration files and SOF files, which are parsed by the host application (for example, EsoRex) and are provided to the recipe via the plugin interface. They also consist of data files themselves, which shall be specified by the plugin interface, although it is up to the recipe code itself to open and parse the files themselves.

All three types of files are listed here.

3.5.1 Configuration files

Configuration files may be used to feed information to recipes. These files contain the parameters and values that can also be specified via the command line. They are typically used by EsoRex, and EsoRex is capable of generating configuration files, which may be subsequently edited to generate the correct behaviour (see Appendix A). Although these are not parsed by the recipe directly, the specification of such files is listed here.

Configuration files shall conform to the same type as those used by packages such as EsoRex. The format is defined as follows:

- Configuration files shall be plain text (ASCII)
- Any line comprising solely whitespace (space, tab, etc.) shall be ignored.
- Any line, whose first non-whitespace character is a hash symbol (#) shall be treated as a comment.
- Non-comment, non-blank lines shall be formed with a keyword, equals-sign and value. These lines are henceforth referred to as “configuration lines”
- A configuration line shall be insensitive to whitespace
- When whitespace is critical within a specified value, the entire value shall be enclosed in double quotes (").
- configuration lines shall be regarded as case-sensitive

Keywords in configuration files shall conform to the following naming convention: *instrument.recipe.parameter*, where each part is the name of that item, as appropriate. For example, for the NACO pipeline, the image jitter recipe has a “rejection border” parameter. It would be named in the configuration file as:

```
naco.naco_img_jitter.rej_bord
```

3.5.2 SOF files

SOF (Sets of Frames) files are used to specify the input files to a recipe. They are not used directly by the recipe, rather are used by the host application to obtain input information which in turn may be provided to the recipe through the plugin interface.

Each line of SOF the file contains three fields, separated by whitespace. The first field is the fully-specified filename. The second field is the DO (Data Organiser) category. The third field is either `RAW` (a raw input frame) or `CALIB` (a calibration frame). The second and third fields are instrument specific.

`RAW` implies the raw files (in the true sense of the word) from the telescope. This includes science frames, flat fields, bias frames, etc..

`CALIB` files are those which will be used for calibrating the instrument. These have already been processed in some way (for example: master bias, etc.).

Note that not all data files used as recipe inputs are always `RAW`; one can (and does) use `CALIB` files as recipe inputs.

When developing a recipe, it is acceptable to use any category name as the recipe developer sees fit. ESO recommends that categories be named using only uppercase letters, as this is typical within the DFS.

The categories themselves are defined in the `iiinstrument_dfs.h` file. However, when the recipe is delivered by the pipeline consortium to ESO, these definitions will be changed to match the output generated by the DO. As a result, the category names shall be abstracted in a single location, and the symbolic name used throughout the rest of the code, as required.

See Section 4.4.5 for an example, as well as details about how these fields are incorporated into the recipe operation. An example SOF file is also given in Section A.4.11.

3.5.3 Input data files

The recipe itself may read in data files directly. The full path of the data file, the DO category and file data category are specified through the plugin interface as a frameset (often this information comes originally from the SOF file — see Section 3.5.2, above). The plugin interface and frameset concepts are a part of the CPL functions for handling recipes and passing data to them. See the CPL documentation for details [7,9].)

Input data files are typically in FITS format, and shall be parsed using the CPL FITS handling routines.

3.5.4 Naming and location

As the SOF file provides the full path name, there are no specific requirements for the naming of input files, other than that they are all specified through the SOF file interface system. The SOF file is specified to the host application (for example, EsoRex), so there are no further requirements there.

3.6 Output data (products)

Data files must be FITS format and are written to the local directory by the recipe. However, they may subsequently be renamed by host software.

The list of created files shall be returned as a CPL *frameset*. This frameset should indicate which items are *products*. This is done using by setting the group parameter of an individual frame to `CPL_FRAME_GROUP_PRODUCT`.

3.6.1 Format

All output files shall be in FITS format [10].

3.6.2 Specifying output data using 'framesets'

The CPL recipe interface will accept a list of output products in the form of a frameset. The frameset is the recipe's way of telling the host application what the output file (products) of the recipe are. Each frame of the frameset provides for the specification of the filename, as well as a number of other fields. At the time of output, the recipe shall complete all fields for each generated product.

filename This is the local filename. No path information shall be included.

type This shall be set to either IMAGE, TABLE or MATRIX to indicate the type of data contained within the file.

group This shall be set to CALIB, in the case of produced calibration frames, or PRODUCT, in the case of output scientific data. The other possibility is RAW, but this value shall never be used for recipe products.

level This shall indicate the status of the output product in the context of the entire pipeline. It shall be set to TEMPORARY, INTERMEDIATE or FINAL, as appropriate.

tag This shall be the category for the product. As there are no conventions as to what these values are called, that shall be defined in a central location within the code (see Section 4.4.5) so that the value can be easily changed across the entire suite of recipes for a given instrument.

See the CPL Reference Manual documentation on frames for more information regarding frame keyword specification [9]. In particular, the manual specifies the exact symbolic name that shall be used in the code itself (e.g. IMAGE becomes `CPL_FRAME_TYPE_IMAGE`).

See also Section 4.4.5.

3.6.3 Naming

Each output file shall be named in a unique way for each individual execution of the recipe. Filenames shall consist of a *stem* and *suffix* separated by a single full-stop ("`.`").

The following filetype-suffix associations shall be made.

Suffix	Format	Description
`fits`	FITS	Flexible Image Transport System. Used for image storage.
`tfits`	FITS	As above, but for storing *only* tables.
`paf`	PAF	Parameter File. (Deprecated)

See Reference [10] for details of the FITS format.

It is the responsibility of host applications — for example, EsoRex — to rename files as appropriate.

3.6.4 Location

All output files shall be written into the current working directory. It is the responsibility of host applications — for example, EsoRex — to rename, copy or move files as appropriate.

3.6.5 Log files

Log files are handled automatically by the host application, provided that that developers use the available CPL messaging calls for all message output. These calls are selected according to the severity level of the message. The choices, in decreasing order of severity, are:

- `cpl_msg_error()`
- `cpl_msg_warning()`
- `cpl_msg_info()`
- `cpl_msg_debug()`

In choosing the appropriate message level, it should be assumed that normal, successful operation, by an interactive user will be done at “info” level. The “debug” level is used for the maximum level of messaging, but should be used with the intended audience being the end-user of the recipe, not necessarily the developer. The “warning” level shall be used for any anomalous behaviour in the recipe or recipe environment. The “error” level should be reserved for messages which imply that human inspection or intervention is required in otherwise unattended operation.

The logfile itself is written during the operation of the recipe to a file `.logfile` in the current working directory. Host applications (such as EsoRex) will move/copy this file as appropriate.

Log file and terminal output may then be controlled at run-time by the host application.

The standard library functions (for example `printf()`) should never be used by recipe code.

3.7 Documentation

The following outlines the documentation requirements.

3.7.1 Intrinsic

Code shall be written in a clear and concise manner, without using complicated or obscure syntax or constructions. Comments shall be provided regularly throughout the code, and particular sections which require complexity shall be clearly documented.

3.7.2 Block comments

All code shall include in-source commenting. This includes a header block comment containing the copyright information. It also includes the header block comment for each function. Examples of both of these are shown in Section C.

3.7.3 Automatically generated documentation

The comments written in header blocks shall make use of the *Doxygen* keywords to permit the automatic generation of documentation.

3.7.4 Written documentation

In addition to the source code itself, the recipe developer shall provide written documentation which describes the algorithms and methods used in the delivered code.

This documentation shall be part of the *Data Reduction Library Specification* and *Data Reduction Library Design*. The details of these documents are specified in the *Data Flow for VLT/VLTI Instruments Deliverables Specification* [3]

3.7.5 Online documentation

There is presently no requirement to generate on-line documentation for any recipe. However, by providing Doxygen tags in function header blocks, it should always be possible to generate it. Doxygen tags are already present in the recipe template.

3.8 Test requirements

The recipe developer shall provide validation procedures and any supporting test software to satisfy the requirement for Validation and Testing, as described in the *Data Reduction Library Design* (which, in turn, is specified in the *Data Flow for VLT/VLTI Instruments Deliverables Specification* [3]).

3.8.1 Compilation Tests

A `tests` directory in the delivered software (also provided in the recipe template) shall be used for the provision of post-compilation testing. It shall be used to verify that the compilation was successful and that the recipe

or recipe functions may be called without failure. The tests still execute the code — it is not a static checking facility like, for example, `splint` — but they do not require any additional data files.

This stage is used to catch any major build failures prior to the need to obtain and unpack potentially many gigabytes of test data.

3.8.2 Unit Tests

Every recipe data reduction function (Section 4.4.15) shall have its own unit test suite, which comprises one or more unit test cases.

A unit test case (Section 3.8.3) is a C function which calls the recipe data reduction functions with a given set of parameters.

There shall be at least one unit test case for every instrument mode, and there can be as many test cases as necessary, depending on how complex the function is or how many modes and conditions are to be covered. Regardless, the following unit test patterns shall be implemented for the `rrrecipe_reduce()` functions (or equivalent, as described in Section 4.4.15), along with any function called by it, that was developed by the consortium:

- A simple pass unit test pattern (black box testing): the related unit test case gives normal inputs to the `rrrecipe_reduce` function so that the latter generates an expected and repeatable result.
- A simple fail unit test pattern (black box testing): it exercises an error, i.e. tells that, given the same condition as the unit test, the code of the `rrrecipe_reduce` function will correctly trap some high-level expected errors (for example, “input file not found”).
- The code-path pattern (white box testing): priority is not to set up the conditions to test for pass/fail, but rather to set up conditions that test the code paths.ăThe results are then compared to the expected output for the given code path.
- Parameter-range pattern (white box testing): test every `rrrecipe_reduce` function using a range of conditions as wide as possible, in particular exercise the limit values of the various parameters.
- Performance patterns: such test cases will try to answer to following questions: how efficiently does the code under test perform its function?ăHow fast?ăHow much memory does it use? Does it trade off data insertion for data retrieval effectively?ăDoes it free up resources correctly?ă

The unit tests source code should evolve together with the code of the `rrrecipe.c` module (Section 4.4.8). The coding of the unit test cases should be done in parallel with the implementation of the `rrrecipe_reduce` module.

The source code of the unit tests shall follow the same coding rules as the actual recipe modules and functions.

3.8.3 Test cases

Each recipe shall have at least one test case per instrument mode that it supports. This test case shall comprise:

- a set of reference data,
- a list of required parameters in order to process this data, and
- a list of the expected results for that test.

Test cases shall be extended to cover the following “milestones” within the project development.

1. Consortia delivery before commissioning
2. Consortia delivery after commissioning
3. ESO version before beginning of operations
4. ESO version during operations including test cases indicated by: DFO/PSO,SEG,pipe developer.

Tests shall be presented in a way that permits their execution and documentation by SEG testers.

Tests shall be provided as a separate package (tar-archive file) to that of the actual instrument recipe(s) package, due to the fact that the tests are not required for established operation, and that the quantity of data associated with these tests may be very large.

3.8.4 Test software

Recipe developers are expected to provide a unitary test for each recipe data-reduction function. They are also expected to provide test data and expected results for standard operation of the recipe. The exact nature of the tests, including a verification matrix, shall be provided for the Preliminary Acceptance Europe (PAE) and Preliminary Acceptance Chile (PAC)

3.8.5 Additional testing tools

See Section 3.2.4.

3.9 Delivery and compliance testing

The final stage of the delivery of the recipe shall be carried out by means of formal compliance testing. This completion of the recipe development programme may be divided into four main components. Refer to the Data Flow for VLT/VLTI Instrument Deliverables Specification document for more information regarding the formal testing sequence within the complete development programme [3].

3.9.1 Completeness of delivery

This is to check for the inclusion of all required components in the delivered recipe. This means all build scripts, read-me files, source code, etc.. as indicated in the code requirements.

The provision of the the required documentation shall also be confirmed.

Additionally, the existence of a separate package containing the test data and test software shall be confirmed at this point.

3.9.2 Check delivery compliance

At this stage, the conformance of the delivered recipe is tested. This includes compliance to ESO coding standards, compliance to the FDR design documentation and the correct building of the software.

Post compilation tests (see Section 3.8.1) are checked at this stage.

3.9.3 Check delivery integrity

This stage involves the application of the testing with the provided test data. The exact tests will be defined as part of the FDR. The integrity tests, in terms of software, shall be provided along with the test data as a separate package to that of the recipe code itself. This is to save copying large quantities of data to the production-line environment unnecessarily.

3.9.4 Completion of the FDR test plan

The FDR will provide a test plan for each recipe within the delivered pipeline software. This is the final check before the delivery is regarded as complete.

4 How to Develop a New Recipe — a step-by-step approach

Conforming to a standard structure is tricky, due to the unfamiliarity of the system to any new developer. It is believed that the best way to instruct software engineers who are new to a pipeline/recipe product is through a commented example, which results in an operational prototype, ready for use in the development of a production system.

Adopting this approach, ESO makes available a *recipe template*. The recipe template is a tar-file containing the software development directory structure, pre-generated `configure` and `makefile`s and prototype source code. It will compile in its own right and serve as an example of the recipe code structure that is expected. However, it also serves as a template, into which recipe developers can simply insert their own algorithms directly.

By providing this infrastructure, it allows developers to concentrate on the important aspects of their work — namely, developing the instrument data reduction algorithms.

This section describes the recipe template, and goes through the components one by one. In addition to documenting the individual components, it also serves as a complete tutorial, and it's completion will result in a working recipe, albeit primitive, which may be run using GASGANO or EsoRex.

4.1 Getting and installing the required packages

The recipe template is part of the CPL Recipe Software Development Kit (hereafter referred to as the SDK). The SDK shall be made available from the CPL website `http://www.eso.org/cpl/`. The SDK contains all the necessary components for running recipes, including the GNU tools, the CPL and QFITS libraries, as well as the recipe template itself. It does not contain a compiler, although this can be expected on almost every system. See Section 4.1.1 below for the compiler requirements.

To install the recipe template unpack the archive and read the instructions in the top-level `README` file. While this will describe how to go about installing the entire SDK, this document will focus solely on installing the recipe template itself, and all subsequent sections should be read bearing that in mind.

4.1.1 Requirements

The recipe template requires the same libraries as are stipulated for general recipe development. These are listed in Section 3.2.6.

4.1.2 Obtaining the necessary software

For installing the necessary software, the following packages are required:

- An ISO-C compiler (preferably GNU `gcc` v3.2 or better)
- GNU autoconf (version 2.57 or better)

- GNU automake (version 1.7.2 or better, ensure autoconf is already installed before attempting installation)
- GNU libtool (version 1.4.3 — provided with the distributions of CPL, EsoRex, etc., ensure automake is already installed before attempting installation)
- QFITS (version 4.3.5 — from `http://www.eso.org/projects/aot/qfits/`)
- CPL (version 1.1 or later — available from `http://www.eso.org/cpl/`)
- The Recipe Template — not publicly available at the moment. See Section 4.2.
- EsoRex (optional) — not publicly available at the moment.

In the case of packages that are not publicly available, please contact the CPL team at ESO directly (see Section 1.5). In all cases, the software comes as a “gzipped”, tar-file.

4.1.3 Recommended installation location and procedure

The installation procedure for QFITS and CPL is detailed in the CPL Users Manual [7]. The default is that QFITS will be installed in `/opt/qfits` and that CPL and EsoRex will be installed in `/usr/local`. However, this may be easily changed by the developer to suit the local environment.

For the installation of the relevant GNU tools, see the documentation that comes with those packages.

4.2 Obtaining the recipe template package

The recipe template package is part of the CPL Recipe Software Development Kit (SDK) archive, which is available as a compressed “tar” archive. The recipe template contains the complete skeleton, and it will compile and link “as is”.

4.2.1 FTP/web sites

The latest version of the recipe template will be announced and made available via the main CPL website, `http://www.eso.org/cpl/` [12]. From the main page, follow the link to the “Downloads” page and, if available, the recipe template archive package will be listed with the other items for download. Of course, links to the *Common Pipeline Library* itself, as well as QFITS may also be found here, although these are included in the SDK itself, so there is generally no need to download these as well.

Once downloaded, unpack the archive in a directory tree, where write permission is given. The archive will create its own directory, which contains the hierarchy of files needed to build the recipe template.

The `README` file in the distribution should be checked for details of the installation procedure as well as any notes particular to that version of the code. However, the instructions given in the Section 4.3 provide a more detailed description of the installation procedure and what may be expected at the various stages, allowing problems with any given installation to be quickly identified.

4.3 Building and testing the unmodified recipe template

Having obtained the recipe template tarfile, the first stage is to unpack and build it. Uncompress and untar the file, to create the recipe template source directory, `iiinstrumentp`.

4.3.1 Building the recipe template

Be sure to check the `README` file for detailed instructions on the installation procedure. Generic instructions on the use of the `configure` script can be found in the `INSTALL` file.

However, most installations will conform to the following procedure. To build the recipe template, first change into the `iiinstrumentp` directory and run the `configure` script.

```
% cd iiinstrumentp
% ./bootstrap  # <-- Optional - see below!
% ./configure --with-qfits=/home/user/qfits \
             --with-cpl=/home/user/cpl \
             --with-cext=/home/user/cext \
             --prefix=/home/user/recipes \
             --enable-maintainer-mode
```

The `./bootstrap` is only used when the recipe is being built from the CVS archive (that is, for ESO development staff). If the recipe template is being built from a tar-file, then this step in unnecessary.

The `--prefix` directory is the location where the installed recipe plugin will be placed. This should be a directory other than the recipe template source tree. It should also be separate from other library directories (to avoid potential conflicts in library handling). Typically, a system will have a dedicated recipe repository, where all plugin libraries can be stored.

Note that the additional `configure` parameter, `--enable-maintainer-mode`, is required if the developer intends to package the software for distribution. While not necessary for normal development within the CVS context, it is generally expected that the source code will be delivered or distributed as a tar-file archive. In order to build the tar-file (that is, use the `make dist` options), then the “maintainer mode” needs to be enabled. This topic is described further in Section 4.7.

The QFITS and CPL directories are those where those packages were installed. However, the CEXT package is likely to be found in the same directory as the CPL, as it is an integral part of that system. (Only in the case of a modified (and unusual) CPL installation will it be located elsewhere.)

Note that if, for some reason, it is necessary to re-run the `configure` script, it is essential that the `config.h` header file is removed first. This is because the `configure` script only checks for the existence of this file when determining whether or not it needs to be created. Therefore, any changed configuration parameters will not be created, should the file already exist.

Once the configuration has completed, the code should be built and then installed. This is accomplished simply, using the following commands.

```
% make
```

```
% make install
```

Problems at this stage are most likely caused by the inability of the build procedure to find the relevant libraries. Double check the paths specified when running the `configure` script (see above).

4.3.2 Testing the recipe template

With the recipe template built and installed, it should be possible to test its operation using EsoRex or GASGANO. For the purposes of this document, only the EsoRex case shall be given. Detailed instructions on the use of EsoRex are described in Appendix A. However, for the purposes of the testing here, it is assumed that a suitable configuration file has been setup, which provides the correct path to the directory where the recipes are installed. Alternatively, the recipe directory can be specified using the command line option, `--recipe-dir`. This directory is the same directory as the one specified using the `--prefix` option, when the `configure` script was run during the installation of the recipe template.

When using EsoRex to detect for the existence of the recipe, the `-recipes` option is used. Here is an example, showing the anticipated output.

```
% ./esorex --recipe-dir=/home/user/recipes --recipes

  esorex -- ESO Pipeline Tool, version 0.9.5 (beta release)
   Test release only. Names/functions are likely to change.

List of Available Recipes :

  rrrecipe              : Short description of rrrecipe

%
```

Here, `rrrecipe` is the name of the recipe. The text `Short description of rrrecipe` is the synopsis of what the recipe does. These will be changed when the recipe template is converted into a proper development recipe.

```
% ./esorex --recipe-dir=/home/user/recipes --help rrrecipe

  esorex -- ESO Pipeline Tool, version 0.9.5 (beta release)
   Test release only. Names/functions are likely to change.

Recipe: rrrecipe -- Short description of rrrecipe

Usage: esorex [esorex-options] rrrecipe [rrrecipe-options] sof

Options:

  --stropt              : the string option []
```

```
  --boolopt             : a flag [TRUE]

For help on the options of esorex itself, please use the command
'esorex --help' (that is, without specifying any recipe name). For
more information about the recipe, one can also use the command
'esorex --man-page rrrecipe'.
```

4.4 Modifying the recipe template

Once the recipe template has been tested *unmodified*, the conversion to the real recipe should commence. By the end of this process, there will be an operational recipe, which will be correctly formatted and ready to accept the inclusion of the data reduction algorithms.

While this document simply goes ahead and modifies the installed recipe template, it may be wise, when starting, to make a copy of the entire recipe template tree before starting the next step. That way, there will be a known working version of the recipe template available for easy reference during the code modification process.

4.4.1 Example names

In all subsequent examples, the following names will be assumed.

xxxins — The instrument name.

xxxrec — The recipe name.

zzzkey — An example (FITS) keyword name.

4.4.2 Renaming the files and functions

The first step is to rename all directories, files, and all functions within the files, changing key words into the appropriate names. Within the recipe template, there are several key words that are unique and thus may be easily found and replaced. They are as follows:

Key word	Description	Example
`iiinstrument`	The name of the instrument	*naco*
`rrrecipe`	The name of the recipe (including the instrument, mode and recipe)	*naco_img_twflat*

Note the number of `i`s and `r`s at the beginning of `iiinstrument` and `rrrecipe`, respectively. Also note that the recipe name includes the name of the instrument, as well as the instrument mode and the actual recipe operation. See Sections 3.2.8, 3.2.9 and 3.2.10 for details on the naming scheme.

Note also that the 'search-and-replace' operation must be case-sensitive. That is, if `iiinstrument` is changed to `naco`, then `Iiinstrument` must be changed to `Naco` and `IIINSTRUMENT` must be changed to `NACO`.

The ‘search-and-replace’ applies to function and variable names within the files, as well as the files and directories themselves.

The recipe name itself doesn’t need to be a mammoth description. In the above example, `naco_img_twflat`, the `twflat` refers to“twilight flat-field correction”.

And remember to leave the ‘p’ at the end of the main instrument directory name. For example, the development directory for the NACO pipeline would be called `nacop`. Of course, if a strict ‘search-and-replace’ is performed, this will be handled automatically.

4.4.3 Configuring the pipeline functions

Once the directories, files, functions and variables have all been renamed, the next step is to configure the two main components of the pipeline development system: the pipeline functions common within the instrument and the recipe functions themselves.

To start, the pipeline functions shall be considered.

Within the `iiinstrumentp` directory (which will have been renamed by this stage), there will be a second directory `iiinstrument` (no trailing ‘p’). This directory contains all the common functions for this particular pipeline. In particular, there are three modules which are provided and which give examples of functions that nearly every pipeline will use. These modules are:

`iiinstrument_pfits.c` — Section 4.4.4

`iiinstrument_dfs.c` — Section 4.4.5

`iiinstrument_utils.c` — Section 4.4.6

Details about adding common functions beyond those in these three files are set out in Section 4.4.7.

4.4.4 `iiinstrument_pfits`

The `iiinstrument_pfits.c` module contains only one type of function — an accessor for each keyword that may be accessed by any of the recipes. The format of the function name of these accessors is always as `iiinstrument_pfits_get_`<*keywordname*>, where <*keywordname*> is the name used within the recipes (it does not necessarily need to be this value in the FITS file, as this very function abstracts it). The function prototype is as always in the following form:

```
char * iiinstrument_pfits_get_keywordname(char * filename);
```

The returned string is the value associated with the given keyword name. For example:

```
char * iiinstrument_pfits_get_dit(char * filename)
{
    char    *   val ;
```

```
    if (qfits_is_paf_file(filename)) {
        val = qfits_paf_query(filename, "DET.DIT") ;
    } else {
        val = qfits_query_hdr(filename, "DET.DIT") ;
    }
    return qfits_pretty_string(val);
}
```

In this case, the `iiinstrument_pfits_get_dit()` function returns the value associated with the keyword "`DET.DIT`" — the detector integration time. The argument to the function is the name of the FITS or PAF file which is being interrogated. Note that the function will accommodate either format of file.

In addition to giving this flexibility, the provision of such an accessor function also greatly assists in the maintenance of backwards compatibility code. A common example is the case where FITS keywords are changed during the instrument lifetime.

It is also useful for getting a single piece of information that may exist in one of several keywords. This may be something such as checking a "mode" keyword first and then, depending on the result, checking one of several "sub-mode" keywords.

While the `iiinstrument_pfits` uses these accessor functions, it would be prudent to discuss the alternative — *property lists*. These are a feature provided by CPL. While there is no denying their utility, the problem is that the property lists are designed for a fixed structure, and do not handle the exceptions easily. Thus, they are less suited to the "backwards compatibility" scenario suggested above. For that reason alone, their use is strongly recommended. And, of course, it is easy to create new accessors; simply copy an existing one.

The examples supplied in the `iiinstrument_pfits` show the handling of keywords that often occur within VLT instruments. (`ARCFILE`, the archive file, and `DET.DIT`, the detector integration time.)

4.4.5 `iiinstrument_dfs`

One of the requirements of the pipelines is that they provide the mandatory FITS keywords, as defined by the DICB standard [5], in all products. In particular, all products must contain the `PRO_CATG` keywords. These are written into the FITS headers and are read by the ESO data management tools, in addition to other header keywords. With this information, ESO data management tools may define their own category keywords, which are used to classify the data during the operation of the overall pipeline.

While the external pipeline developers do not need to concern themselves with the details of the DO operation, they do need to take into account two important aspects. The first is the recognition of the DO category (which is written into the Set of Frames (SOF) file, and is used to identify the files), and the second is the the `PRO_CATG` keywords, which must be written by the recipe into the products.

The value associated with the `PRO_CATG` keyword indicates the *category* of product and, as such, the values associated with this keyword will be different for each different type of product. Examples include: `MASTER_FLAT`, `MASTER_BIAS`, `BAD_PIXEL_MAP`, `MOS_SCIENCE_REDUCED`, etc.. These values are text strings. Typically, the keywords will be decided in consultation between ESO and the consortium developing the recipe. See Section 3.6.2 for details.

So, in `iiinstrument_dfs.h` every possible `PRO_CATG` value that could be used is defined. E.g.

```
#define RRRECIPE_QC  "RRRECIPE_QC_PRO_CATG"
```

The file `iiinstrument_dfs.h` also contains, the function prototypes, etc. for `iiinstrument_dfs.c`, as is to be expected. The purpose of this is to gather all the keywords in a single place, thus making it easy to change them, in order to match the specification of the keyword by the VLT or to make it possible to easily implement changes to keywords, should the requirements or standards change in the future.

In `iiinstrument_dfs.c`, the DO category classifications are defined. This is done using functions. For each DO category, *xxx*, we need a function, `int iiinstrument_is_xxx(char * tag)`, where `xxx` is a name that is strongly associated with the frame category name that should be recognised with this function.

The functions return 1 in the case that the DO category is matched, 0 in the case that the DO category is not matched, and -1 in the case of an error.

For example:

```
int iiinstrument_is_badpix(char * tag)
{
    /* Test entries */
    if (tag == NULL) return -1 ;

    if (!strcmp(tag, "IIINSTRUMENT_BPM_DO_CATG")) return 1 ;
    return 0 ;
}
```

The SOF files contain these classifications. Typically an SOF file will look something like this:

```
a.fits     JITTER_OBJ
b.fits     JITTER_SKY
c.fits     JITTER_SKY
d.fits     JITTER_OBJ
flat.fits  MASTER_FLAT
```

Alternatively, it may have the additional RAW/CALIB classifications, such as:

```
a.fits     JITTER_OBJ   RAW
b.fits     JITTER_SKY   RAW
c.fits     JITTER_SKY   RAW
d.fits     JITTER_OBJ   RAW
flat.fits  MASTER_FLAT  CALIB
```

See Section 3.5.2 for additional information.

Note that it is the recipe's responsibility to check that all input files exist. It should not be left to the host application to ensure that all SOF classification tags are provided. Some recipes may also need to cope with un-tagged input files (for example, raw files with hitherto no classification).

In addition to these aspects, there are three other functions within the `iiinstrument_dfs.c` module.

`iiinstrument_compare_tags()` — is used for input frame comparison and should not generally be changed. It is used for classifying (referred to as "labelising") the input frames, and the function provided should cover all cases. That is, there is no need to modify this function.

`iiinstrument_add_pro()` — writes the `PRO_` keywords into a property list. Essentially, it modifies the property list before creating the products. In amongst the arguments there is one argument for each `PRO_` keyword type.

`iiinstrument_add_inputfiles()` — writes the list of files that have been used into the HISTORY keywords within the fits file. Apart from changing the name, this function should not be altered.

4.4.6 `iiinstrument_utils`

The `iiinstrument_utils.c` file contains code to accomplish two utility operations. The first is the provision of the license. This is typically the same for all the recipes of the pipeline and is best placed in the common code area. Simply replace the license in the recipe template with that applicable to the recipe being developed.

The second is the `iiinstrument_frameset_to_iset` function, which is provided as a standard utility function, and does not require any change.

The `iiinstrument_utils.c` is also a good module in which to place other common functions (see Section 4.4.7).

4.4.7 Additional functions

Should any other additional functions be required, and which are common to several recipes within the pipeline development area, then the `iiinstrumentp/iiinstrument` directory is the appropriate place to put them. That is, any other reusable utility functions that are generic to the instrument should go here. While there are no strict naming conventions in force, it would be good to group such functions into a single module, or at least modules with a related name base.

In the case of a single function, it is acceptable to put it in the `iiinstrument_utils.c` file (described in Section 4.4.6). However, it is not unreasonable for many functions to be necessary, in which case the use of separate modules is expected. These modules should be named `instrument_yyy.c`, where `yyy` is an appropriate indication of the content of the file.

4.4.8 Writing the recipe

Once the common functions have been written or modified, the recipe functions themselves can be written.

Within the `iiinstrumentp` directory (although it will have been renamed, of course), there will be a directory `recipes`. This directory contains all the recipe modules.

Typically, each recipe will have its own module. However, in the case of particularly large recipes, it is possible to split a given recipe module into multiple files. However, the main recipe module shall retain the name

`rrrecipe.c` (appropriately changed according to the name of the recipe of course). The additional files may be named according to the recipe name, with an appended functional identifier (separated by an underscore). For example, the recipe may be named *xxxrec*, in which case, the recipe module will be `xxxrec.c` or, perhaps something like `xxxrec.c` in combination with two other files, such as `xxxrec_model.c` and `xxxrec_astrometry.c`, to give an example. Note that the recipe template expects these as separate source/object files, rather than collating them into a library. This has been done simply to keep the build simple.

Remember that `xxxrec.c` is reserved for the top level functions, and should contain at least:

- `cpl_plugin_get_info()`,
- `rrrecipe_create()`,
- `rrrecipe_exec()`,
- `rrrecipe_destroy()`, and
- `rrrecipe()`.

The implementation of these functions is described in the section entitled *Software development with the CPL* in the *VLT Common Pipeline User Manual* [7]. See also Section 3.2.10 in this document for more details. A completely listing of the recipe template source code for `rrrecipe.c` is given in Appendix C

In the `xxxrec.c` file, there are the usual function prototype definitions. All functions within the module should be forward declared. Do not rely on implicit or non-prototyped functions. The exception to this rule is the function `cpl_plugin_get_info()`, as it is already defined in the `cpl_plugininfo.h` header file.

4.4.9 Global variables

The next item to consider is the `rrrecipe_config` structure. This contains the global variables (within the scope of the module). These are stored in a structure for data sharing. The member variables store the command line input and any computed QC parameters.

Then there is another global variable `rrrecipe_man` which contains a character array containing the description of the recipe. This description is used in generating the 'man page' documentation (a feature of the EsoRex utility — see Section A). The `rrrecipe_man` variable is incorporated into the recipe as an argument to the `cpl_plugin_get_info()` function.

4.4.10 `cpl_plugin_get_info()`

The first function as such is the exception to the rule on function prototypes within the recipe module. The `cpl_plugin_get_info()` function defines the recipe interface, and is used by calling packages (e.g. EsoRex) in order to determine information about the recipe as a plugin, in order to dynamically load and execute it [7].

Within `cpl_plugin_get_info()`, there is the call to the core setup function: `cpl_plugin_init()`. The arguments to this function should be altered as per the `cpl_plugin_init()` example in the documentation ([7]). In particular, the last three arguments are the names of the functions that create, execute and destroy

the plugin environment at run time. Although these names do not need to be modified (assuming that the global search and replace has changed the `rrrecipe_` part to the name of the recipe, they are being pointed out, in order to trace the path of control through the code. Additionally, the information presented here complements the documentation given in the CPL User and Reference Manuals [7,9].

The function prototype for cpl_plugin_init() is as follows:

```
void cpl_plugin_init (
   cpl_plugin * self,
   unsigned int api,
   unsigned long version,
   unsigned long type,
   const char * name,
   const char * synopsis,
   const char * description,
   const char * author,
   const char * email,
   const char * copyright,
   cpl_plugin_func create,
   cpl_plugin_func execute,
   cpl_plugin_func destroy )
```

As there are quite a number of arguments in this function, and that there definition is crucial to the interface between the recipe and the host application, they shall be discussed here individually.

self The plugin to initialise.

api The plugin interface version number. This defines the interface implementation, and is provided as a way of implementing future upgrades to CPL without changing the interface. For any recipes, this should use the symbolic constant `CPL_PLUGIN_API`.

version The pipeline's version number. Because of the way the recipe template is organsied, this is generally implemented as a single version number for all recipes. The number itself is implemented as an `unsigned long`. The units are the 'patch level', the hundreds are the 'minor version' number and the ten-thousands are the 'major version' number. Thus, 30127 would be used to represent v3.1.27. Note that with this system, the number of minor versions and patch levels is limited to one hundred each. Within the recipe this is implemented using the symbolic constant `IIINSTRUMENT_BINARY_VERSION`, which is generated automatically from the configure script. See Section 4.7.1.

type The plugin interface type. In this case, a recipe is being developed, so the CPL symbolic constant `CPL_PLUGIN_TYPE_RECIPE` should always be used.

name The plugin's unique name. See Section 4.4.2.

synopsis The plugin's short description (purpose, synopsis ...). This should typically be 30 to 50 characters long.

description The plugin's detailed description. This may be any length. However, bear in mind that static literal strings have a limited length under ISO-C, and so this may need to be generated dynamically, by some other means.

author The plugin's author's name.

email The plugin's author's e-mail address.

copyright The plugin's copyright and licensing information. Like the 'description', above, remember the ISO-C limitations on statically declared literals.

create The function used to create the plugin. In the recipe template, this is the function `rrrecipe_create`, which is described in Section 4.4.11.

execute The function used to execute the plugin. In the recipe template, this is the function `rrrecipe_exec`, which is described in Section 4.4.12.

destroy The function used to destroy the plugin. In the recipe template, this is the function `rrrecipe_destroy`, which is described in Section 4.4.13.

These last three functions need to be discussed individually. Note that there is no need to make them publicly available because they are exported by the plugin interface itself and they are only called through this interface.

4.4.11 `rrrecipe_create()`

In the first instance, `rrrecipe_create()` must convert the plugin which has been passed to it, from a pointer to the type `cpl_plugin`, to a pointer to `cpl_recipe`, to get access to the additional members that the `cpl_recipe` class provides. This cast operation is safe since the plugin has been explicitly instantiated as a `cpl_recipe` in the `cpl_plugin_get_info` function that was called initially. However, the developer does not need to concern himself or herself with this detail, and it suffices to leave the cast intact as:

```
cpl_recipe * recipe = (cpl_recipe *)plugin ;
```

Following this, is the main purpose of the `rrrecipe_create()` function: establishing the configuration parameters. These parameters may be used to control functions within the recipe itself, and will be automatically detected, displayed and parsed by any calling program (for example, EsoRex).

The first step is to declare and initialise the list:

```
cpl_parameter * p ;
recipe->parameters = cpl_parlist_new() ;
```

After this, each option that the recipe will support is added using these three calls, for each parameter:

- `cpl_parameter_value_new()`

- `cpl_parameter_set_alias()`
- `cpl_parlist_append()`

See the CPL Reference Manual [9] for more information about these function calls.

Note that this code doesn't actually get the options as such, it merely creates their entries, which then allows host programs (such as EsoRex) to know their name, type, description and any default options.

If the host program changes or configures any of these options, it is handled automatically, and the recipe may simply query the parameter when it needs a given value, knowing that it will have been already configured.

In the recipe template, two examples are given.

4.4.12 `rrrecipe_exec()`

Other than the naming, this function should not be altered. It simply recasts the argument from a pointer to `cpl_plugin` into a pointer to `cpl_recipe`. It then calls the `rrrecipe()` function, passing the segregated `cpl_parlist` and `cpl_frameset` pointer arguments.

The main purpose here is to abstract the 'plugin' concept. That is, from `rrrecipe()` onwards, the plugin idea as such is hidden, and the developer deals only with a frameset and parameter list.

4.4.13 `rrrecipe_destroy()`

This function simply deletes the parameter list associated with the recipe plugin. There is no need to modify this code.

4.4.14 `rrrecipe()`

The `rrrecipe()` function is called with two arguments. The first is a CPL parameter list which contains the parameters supported by the recipe. These were defined in `rrrecipe_create()` (Section 4.4.11). The second argument is the list of frames that have been given to the recipe for processing.

The first stage of operations within the `rrrecipe()` function is to actually parse the input parameters and to put the values into a global structure for ease of use. This structure is defined at the start of this module and is described in Section 4.4.9.

The next block of code groups the frames according to type. This is a typical operation for recipes. The code fragment is reproduced here.

```
/* Retrieve calibration data */
/* Labelise the input frames according to their tags */
if ((labels = cpl_frameset_labelise(framelist, iiinstrument_compare_tags,
                &nlabels)) == NULL) {
    cpl_msg_error(fctid, "Cannot labelise the input frames") ;
```

```
    return -1 ;
}
if (nlabels == 1) {
    /* Only one label - all images are raw frames */
    cur_frame = cpl_frameset_get_frame(framelist, 0) ;
    tag = (char*)cpl_frame_get_tag(cur_frame) ;
    if (rrrecipe_is_raw(tag)) {
        rawframes = cpl_frameset_copy(framelist) ;
    }
} else {
    /* For each label */
    for (i=0 ; i<nlabels ; i++) {
        cur_set = cpl_frameset_extract(framelist, labels, i) ;
        cur_frame = cpl_frameset_get_frame(cur_set, 0) ;
        tag = (char*)cpl_frame_get_tag(cur_frame) ;
        if (rrrecipe_is_raw(tag) == 1) {
            /* Raw frames */
            rawframes = cpl_frameset_copy(cur_set) ;
        } else if (iiinstrument_is_flat(tag)) {
            /* Flatfield calibration file */
            flat = cpl_strdup(cpl_frame_get_filename(cur_frame)) ;
        } else if (iiinstrument_is_badpix(tag)) {
            /* Bad pixels calibration file */
            badpix = cpl_strdup(cpl_frame_get_filename(cur_frame)) ;
        }
        cpl_frameset_delete(cur_set) ;
    }
}
cpl_free(labels) ;
```

This code groups the input frames according to type. This, of course, will depend on the type of data that is expected, and so the details may change from one recipe to another. For each given data type that is expected, there will be an `if-else` clause to detect it.

The developer should copy the examples given and use the `instrument_is_xxxx()` functions to detect the existence of the different types (see Section 4.4.5). Any unused type will be ignored, however a warning should be given.

This section of the code also notes if there is only a single type of input frame (`if (nlabels == 1)`) in order to support the case where every frame has been given without any tag. In this case we'd like to assume that all frames are raw frames (this is checked in the example).

The next stage is to check that the mandatory frames have been found. In this case, it is done by checking the `rawframes` frameset variable against `NULL`.

Then the preparation for the actual data reduction begins, by setting the appropriate messaging level. This uses the `cpl_msg_indent_more()` and `cpl_msg_indent_less()` functions. The recipe reduction

function(`rrrecipe_reduce()`) is then called.

In this case, all the data reduction is contained within a single function (`rrrecipe_reduce()`). However, it is conceivable that multiple functions could be be implemented for a multi-stage recipe.

Note the code that cleans up in both the case of error, and in the event of successful completion.

Once the reduction functions have been completed, the products are then saved. Note that during the course of the execution of the recipe, should an products be produced, then the framelist needs to be updated in addition to producing the product itself.

4.4.15 `rrrecipe_reduce()`

The `rrrecipe_reduce()` function is where the actual recipe algorithm is implemented. The completion of this function is subject to the design of the developer, and is therefore not discussed here.

Of course, the `rrrecipe_reduce()` function contains the sequence of data reduction steps, and not necessarily the full implementation of all the code to accomplish it. This may be contained in other functions and modules (such as those described in Sections 4.4.6 and 4.4.7).

4.4.16 `rrrecipe_save()`

The `rrrecipe_save()` function provides a template for several important features. Firstly, it saves the FITS files, using the `plist` feature to generate the appropriate FITS header. See the detailed documentation on `plist` calls for more details [7,9]. In particular, the `iiinstrument_add_pro()` function and the `iiinstrument_add_inputfiles()` function are called. These functions have already been defined in the module `iiinstrument_dfs.c` — see Section 4.4.5.

The next step is to add each of the products to the output frameset. This step is crucial. The frameset contains a list of all the output products and, as such, provides the means by which the external host application can discern the output files.

The last step is to write the PAF file. This file provides a summary of the recipe parameters, and may additionally also log the QC parameters. It is important that this file is also added to the frameset, which defines the output products of the recipe.

As a final point, please note that the developer is free to change the prototypes of `rrrecipe_reduce()` and `rrrecipe_save()` according to his or her needs.

4.5 Implementing the recipe itself

The recipe itself — that is, the core algorithm — is expected to be implemented within the `rrrecipe_reduce()` function, and the functions that it calls (for example, the common instrument functions written within the `iiinstrument_utils.c` module; see Section 4.4.6).

While the exact specification of this function is left to the recipe developer, ESO recommends the function interface that is provided in the recipe template. In particular, the use of the `rrrecipe_config` structure

(defined at the top of the `rrrecipe.c` module) is used as global storage for all parameters that are determined during the course of the recipe, and that are later used by the `rrrecipe_save()` function to store them into the output products. By using this construction, the function definition need not be changed should any additional parameters be added or deleted. (An alternate solution would be to pass them all as arguments into `rrrecipe_reduce()` and `rrrecipe_save()`, although clearly this would be somewhat cumbersome).

The recipe template itself gives image creation as an example, although this too is not fixed, and different recipes could generate alternative products. For example, these could be tables or even simply text-based data files containing QC parameters.

The `rrrecipe_reduce()` function is recipe dependent, and therefore there are no specific requirements, over and above the general coding standards and policies of the recipe as a whole (see Section 3.2) and the scientific requirements of the instrument data reduction package definition.

4.6 Testing

See Section 3.8 for details on the test requirements of recipes.

Within the recipe template, there is a `tests` directory. This has been provided as a location for the inclusion of unit tests, and has been done as a convenience to the recipe developer. While ESO recommends that units tests are placed here, and that they are subsequently included in the delivered recipe, they are by no means mandatory.

These tests are not part of the formal testing process for the recipes and are independent of the required tests described in Section 3.8.

If the `tests` directory is used, note that the main `Makefile` provides a build label for any tests included there. Thus, it is possible to use the sequence:

```
% make
% make install
% make test
```

to build, install and execute the tests.

4.7 Packaging and delivery

The final point regarding the preparation of an instrument recipe is that of distribution and delivery of the source code.

As mentioned in Section 4.3.1, the `--enable-maintainer-mode` is required as a parameter for the packaging of the system. What this does, is to enable additional “builds” within the generated Makefile. In particular, the `dist` build is used to package the system for delivery.

4.7.1 Version number

The version number is controlled from the `configure.ac` script in the top-level directory of the recipe template. The relevant lines looks something like this:

```
AC_INIT([IIINSTRUMENT Instrument Pipeline], [0.0.1], [hkruger@paulaner.de],
[iiinstrument])
```

The second bracketed field (which in the provided example is `[0.0.1]`) is the version number, split into components: major, minor and patch. On running the configure script, these are translated into the symbolic constants `IIINSTRUMENT_MAJOR_VERSION`, etc., although it should be noted that only the composite version — `IIINSTRUMENT_BINARY_VERSION` is used directly in the code to indicate the version number of all the recipes for the plugin-interface. Note that the recipes within a given instrument pipeline distribution all share the same version number. This is because recipes operate together, rather than in isolation. Thus, when one changes it is typical to upgrade the entire package. Unified version numbers assist this.

Developers are encouraged to be generous in the use of version number increments.

4.7.2 Tagging

As the system is maintained within a CVS environment, it is possible to "tag" the current version of the source code at any given time. This is accomplished with the command:

```
% cvs tag iiinstrumentp-X_Y_Z .
```

where `X`, `Y` and `Z` are the major, minor and patch version numbers respectively. Note that the tag must use underscores ("_") between the version numbers, but a hyphen ("`-`") between the tag-prefix and the version. This is because full-stops are not handled directly by the cvs tagging command. However, the `X_Y_Z` will become `X.Y.Z` in the filename of the tar-archive (see below).

The use of major and minor version numbers is described in the Data Flow for VLT/VLTI Instrument Deliverables Specification document [3]. The patch number shall be used for all variations in the code within a single major/minor entity. Every change to the code, no matter how slight, shall increment the patch number (or major/minor version, if appropriate).

4.7.3 Distribution

To package the code ready for deliver or distribution, the following command should be used from the main `iiinstrumentp` directory:

```
% make dist
```

This will produce a compressed archive file named `iiinstrumentp-X.Y.Z.tar.gz`, where `X`, `Y` and `XZ` are, respectively, the major, minor and patch version numbers as indicated above. The compression is that of `gzip`, and the archive format is `tar` ('UNIX' tape archive).

5 Support

Of course, it is the interests of any external developers who are writing recipe software for ESO, and ESO's own staff, that the development of recipes is supported wherever feasible. Thus, ESO will endeavour to provide answers to questions arising from the interpretation of this document.

5.1 Contacting the ESO/CPL team

In addition to any assigned ESO contact staff, the Common Pipeline Library development team may be contacted in case of query. The details of how to contact these staff may be found on the CPL website [12]:

```
http://www.eso.org/cpl/
```

Click the link to the "Contacts" page, and the details may be found there.

5.2 Documentation

Documentation exists for the CPL as well as other aspects of recipe development. Please refer to the to the list of reference documents provided in Section 1.6.

5.2.1 Printed documentation

PDF file versions of the *VLT Common Pipeline Library User Manual* [7] and *The Common Pipeline Library Reference Manual* [9] are available from the CPL website [12]. From the main page, follow the link to the "Downloads" page and click on the links to the documentation.

5.2.2 Online documentation

Additionally, the *The Common Pipeline Library Reference Manual* [9], is available on-line. Due to the extensive nature of the CPL itself, it may be more pragmatic to use the linked version on the CPL website, rather than downloading and printing the entire manual.

A EsoRex

This Appendix describes the utility EsoRex.

A.1 Introduction to EsoRex

EsoRex is a tool to run recipes created using the CPL. This section describes the operation of EsoRex, along with a description of a number of common operations that the user may wish to perform, along with the expected command sequence.

A.2 Obtaining and installing EsoRex

EsoRex may be obtained from the CPL website (`http://www.eso.org/cpl/`. Download the compressed 'tar' archive and unpack it in a suitable source directory. In the created directory (`esorex-X.Y.Z/`) there will be a `README` file containing specific instructions regarding the build and installation of EsoRex, as well as an `INSTALL` file which provides more general information about the tools used to build the package.

A.3 EsoRex options

The following options are available with EsoRex. They are specified on the command line after the `esorex` command. Arguments to any of the options follow an appended equals sign. For example:

```
% esorex --recipe-dir=/home/user/recipes
```

This command causes esorex to register the directory in which recipes may be found. As no other options or recipes are specified, the program will exit thereafter without further action.

Note that the recipe directory should contain only recipes, and that other dynamic libraries may cause EsoRex to fail, should they be found. As the `--recipe-dir` does a recursive search through the directory tree, then this should also be considered when setting up the directory hierarchy.

A.3.1 List of available options

`--help` This causes EsoRex to display a help message and then exit without further action. The help message contains a list of all the available options, as well as the current (i.e. default) values. If a recipe name is also given, then help will be given for it as well.

`--version` Display version number of the program and then exit.

`--man-page` Display a manual page for the specified recipe, and then exit. Note that this option only applies to recipes, and that it does nothing for esorex by itself. See also the `--help` option.

`--params` List the input parameters and their current settings (whether from the command line or a configuration file) for the esorex application. Parameters are labelled using the parameter's alias. If a recipe is also specified, then the list of its parameters will also be generated in the same way. This option may also be used to determine any default values.

`--recipes` Display a list of all available recipes (that are available in the directory tree specified with `--recipe-dir`).

`--config` Configuration file to be used for EsoRex.

`--recipe-config` Configuration file for any selected recipe.

`--create-config` Creates a default configuration file (called `esorex.rc`) for EsoRex and puts the resulting file into the `.esorex` directory in the home area of the user. If a recipe is specified then the configuration file will be created for the recipe instead (called `recipename.rc`). Note that an existing file will not be overwritten unless the `--output-overwrite` option is set. And, unless the `--cleanup` option is enabled, a backup file will be copied to `filename.rc.bak` in the same directory.

`--recipe-dir` Directory containing recipe libraries. Note that esorex will recursively search not only the specified directory, but all sub-directories below it as well. Multiple directory heads may be specified, by separating the starting paths with colons (:).
For example, `--recipe-dir=/usr/lib:/usr/local/lib`.

`--output-dir` The directory to where the product files should be written.

`--output-prefix` Prefix applied to any output file. For example, specifying `pre` would translate the filename `filename.fits` to `pre_0000.fits`. See also the `--suppress-prefix` option.

`--output-overwrite` When TRUE, any existing product files in the specified output directory will be overwritten. If FALSE, then EsoRex will not destroy any pre-existing files. Note that if the current directory is used as the output directory (i.e. `--output-dir=.`), then temporary files may overwrite pre-existing files anyway.

`--suppress-prefix` When TRUE, the original name of the output product is maintained. If FALSE, then the name of the output file is changed to the *prefix_number* format. The prefix can be altered using the `--output-prefix` option.

`--link-dir` The directory in which a symbolic link to each of the product files should be written. The enable/disable switch to control whether the link is actually made is the `--suppress-link` option.

`--suppress-link` When TRUE, no symbolic link is created to the output product. However, if FALSE, then a symbolic link is created in the directory specified with the option `--link-dir` for each product that is created by the recipe.

`--log-file` Filename of logfile.

`--log-dir` This specifies the directory where the logfile shall be placed.

`--log-level` Controls the severity level of messages that will be printed to the logfile. Possible options include `debug`, `info`, `warning`, `error` and `off`. The extremes are `debug`, where all messages generated by the recipe are displayed and `off`, where all messages are suppressed.

`--msg-level` Controls the severity level of messages that will be printed to the terminal. In the same manner as the `--log-level` option, the available values include `debug`, `info`, `warning`, `error` and `off`.

`--time` Measure and show the recipe's execution time.

A.4 Using EsoRex

The following sections describe some of the common operations that may be performed using EsoRex.

A.4.1 Getting help on EsoRex

The simplest way to obtain help on EsoRex is via its own in-built help system. To do this, simply append the `--help` flag to the `esorex` command.

```
% esorex --help
```

This lists all the command-line options for the EsoRex application itself.

A.4.2 Using an EsoRex configuration file

EsoRex has quite a lot of input parameters, so it is often much more convenient to use a configuration file. It is even possible for EsoRex to create a configuration file for you (see below).

The configuration file contains the EsoRex options, less the `--` switch, but prefixed with `esorex.caller.`. Blank lines are ignored and lines beginning with # are treated as comments.

Here is an example configuration file.

```
# Example EsoRex configuration file
#
esorex.caller.recipe-dir=/home/username/EsoRex/Plugins
esorex.caller.log-dir=.
esorex.caller.log-file=esorex.log
esorex.caller.output-dir=.
esorex.caller.output-prefix=out_
```

Then, to use this configuration file, use the `--config` command switch. E.g.

```
% esorex --config=myesorex.rc
```

Note that the configuration file created here, contains the *essential* configuration options for normal EsoRex use. This configuration file will be assumed in subsequent examples.

Note also, that EsoRex will also search for a configuration file called `esorex.rc` in the `$HOME/.esorex` directory. Any values in this file will be used by default, however any value specified in an explicit configuration file (using the `--config` option), or directly on the command line will overwrite these values.

A.4.3 Using EsoRex to generate a configuration file

Rather than coding a configuration file by hand, it is possible for EsoRex to create its own configuration file, using sensible defaults, and over-riding them with any additional ones that you provide on the command line. To do this, use:

```
% esorex --create-config
```

This will create a file `.esorex/esorex.rc` in your home directory. If the directory `.esorex` doesn't exist, it will be created for you. EsoRex will fill in most of the parameters in the configuration file for itself, but if there are any for which sensible defaults can not be determined, then EsoRex will put them in blank but commented out, and write a warning message to the screen.

If an configuration file already exists, then the old one will be copied to `.esorex/esorex.rc.bak`. This feature can be disabled by using the `--cleanup` option.

A.4.4 Displaying EsoRex settings

```
% esorex --config=myesorex.rc --params
```

This will show the caller (i.e. EsoRex) parameters. However, unlike the `--help` which shows the parameters as command line options, with their description, `--params` will display the variable and its current value, whether that is from the command line or the configuration file.

A.4.5 Listing all available recipes

```
% ./esorex --config=myesorex.rc --recipes
```

This will search for recipes in the recipe directory (which is specified in either the configuration file, or using the `--recipe-dir` command line switch). The directory search is *recursive*. That is, EsoRex will look not only in the recipe directory itself, but in all sub-directories.

Warning: If non-recipe dynamic libraries exist in the recipe directory tree, they may (depending on contents) cause EsoRex to crash. This is due to a limitation in the `ltdl` library, and is a known problem. There is no immediate solution, other than to ensure that the recipe directory tree does not include any non-identifiable dynamic libraries, other than genuine recipes. This is on the list of items to be addressed, and the developers will evaluate and test new versions of `ltdl` in anticipation of a version that fixes this problem.

A.4.6 Getting help on a specific recipe

```
% ./esorex --config=myesorex.rc --help myrecipe
```

This will display information about the recipe `myrecipe`, including a list of the recipes options.

A.4.7 Displaying recipe parameters

```
% esorex --config=myesorex.rc --params myrecipe
```

This will show the caller (i.e. EsoRex) parameters, followed by the recipe parameters. It will show the value that would be used were the recipe actually executed. These values will be the latest specified from either the configuration file or the command line. If the parameter was not specified in either of these locations, the default will be displayed.

A.4.8 Setting recipe parameters

To specify parameters for a recipe, put the options *after* the recipe name. E.g.

```
% ./esorex --config=myesorex.rc myrecipe --myoption
```

Of course, one can use the `--params` option to ensure that the values are correctly read. E.g.

```
% ./esorex --config=myesorex.rc --params recipeone --op=subtract
```

A.4.9 Using a recipe configuration file

Like EsoRex itself, it is possible to also specify a configuration file for the recipe options. Note that this is still an EsoRex command, and so should go *before* the recipe name (only actual recipe options should go after the recipe name). Thus:

```
% ./esorex --config=myesorex.rc --params --recipe-config=myrecipe.rc \
myrecipe
```

would display the parameters that would be used, given the recipe configuration file `myrecipe.rc`.

The recipe parameters must be specified in the configuration file as their full names. Here is an example configuration file:

```
#
# EsoRex configuration file for VIMOS 'vmspcaldisp' CPL recipe.
#

vimos.Parameters.bias.removing.method=Zmaster
vimos.Parameters.extraction.fuzz=5
vimos.Parameters.lamp.frames.validate=true
vimos.Parameters.ids.refine=false
vimos.Parameters.extraction.window=-1
```

```
vimos.Parameters.flat.apply=false
vimos.Parameters.badpixel.clean=false
vimos.Parameters.cosmics.clean=false
vimos.Parameters.BLA=Input
vimos.Parameters.slit.model=false
vimos.Parameters.slit.order=0
```

Note that like the `esorex.rc` configuration file, EsoRex will also search `$HOME/.esorex` for any recipe configuration files and use them as a default. It is also possible to use the `--create-config` option when a recipe name is also specified, and EsoRex will create a configuration file for that recipe. This can then be altered with a text editor.

A.4.10 Actually running a recipe

Without any `--params` or `--help` options, EsoRex will attempt to execute any specified recipe.

```
%  ./esorex --config=myesorex.rc --recipe-config=myrecipe.rc \
myrecipe data.sof
```

See the section on Sets of frames (below) for more details on how to actually specify the data.

A.4.11 Sets of frames

A `sof` file contains a list of the input data. This data is specified in an `sof` file (which is just a text file), where each input file is specified with its associated *classification* and *category*. The format of each line in the `sof` file is as follows:

```
full-path-to-file  classification  category
```

An example file, might look like this:

```
/home/user/data/mos/VIMOS.2003-12-26T01:05:06.233.fits  MOS_SCIENCE    RAW
/home/user/data/mos/VIMOS.2003-12-26T01:26:00.251.fits  MOS_SCIENCE    RAW
/home/user/data/mos/VIMOS.2003-12-26T01:47:04.050.fits  MOS_SCIENCE    RAW
/home/user/data/cal/master_bias4.fits                   MASTER_BIAS    CALIB
/home/user/data/cal/grs_LR_red.3.tfits                  GRISM_TABLE    CALIB
/home/user/gasgano/extract_table2.fits                  EXTRACT_TABLE  CALIB
/home/user/data/cal/badpixel.3.tfits                    CCD_TABLE      CALIB
```

B Incorporating FFTW into Recipes

This Appendix FFTW describes how to use within recipes.

B.1 FFTW

It is recognised that one of the most efficient FFT libraries that is currently availabe is FFTW. At the time of writing, FFTW was at v3.0.1. It is available from `http://www.fftw.org/` and is released under the GNU Public License. It is recommended over GSL for serious FFT work.

FFTW is an optimised, general-purpose discrete Fourier transform library. Its main feature is the ability to generate a "plan" prior to the execution of the transform. This means that multiply executed FFTs may have the optimal method chosen for them, as the FFTW library abstracts many different individual algorithms. These include 1-, 2-, 3- and N-dimensional transforms, complex data, real-data, even- or odd-symetric real data and discreet Hartley transforms.

B.2 Data types

Superficially, it would appear that the use of FFTW is prohibited by the different types that are used. For example, while the CPL may use a 2-dimensional "`cpl_image *`" type, FFTW uses an unwrapped linear array of type "`fftw_complex *`".

B.3 Row/column order

Note that FITS images are row-major order, but that the function `cpl_image_get_nx()` returns the number of columns, not the number of rows, thus reflecting the cartesian convention. In the same way, the function `cpl_image_get_ny()` returns the number of rows. Thus, when calling `fftw_plan_dft_2d()`, the first two arguments will be in the order that the first argument indicates the axis that changes slowest — `ny`, then `nx` follows.

B.4 Example

The following example shows some code fragments to indicate possible ways of converting and transforming CPL data types using FFTW.

```
#include <stdio.h>
#include <stdlib.h>
#include "fftw3.h"
#include "cpl_init.h"
#include "cpl_image.h"

int main() {
```

```
cpl_image *img_in;          /* Input image */
cpl_image *img_re;          /* Output image */
cpl_image *img_im;          /* Output image */
fftw_plan p;                /* FFT plan (for the FFTW library) */
fftw_complex *in;           /* Input data array */
fftw_complex *ri;           /* Output data array (real-imaginary) */
double *data_in;            /* Unwrapped input data */
double *data_re;            /* Unwrapped output data */
double *data_im;            /* Unwrapped output data */
int nx;                     /* Number of x-elements in data array */
int ny;                     /* Number of y-elements in data array */
int i;                      /* Loop counter for indexing the data */


/* Setup the CPL */
cpl_init();

/* Create a test image */
img_in = cpl_image_gen_gaussian_double (16, 16, 7.7, 7.6, 10.0, 2.0, 1.5);

/* Extract the image data and parameters */
data_in = cpl_image_get_data_double (img_in);  /* Data */
nx = cpl_image_get_nx(img_in);                 /* Size of X-axis */
ny = cpl_image_get_ny(img_in);                 /* Size of Y-axis */

/* Establish the input/output data arrays and the FFT plan */
in = fftw_malloc(sizeof(fftw_complex) * nx * ny);
ri = fftw_malloc(sizeof(fftw_complex) * nx * ny);
p = fftw_plan_dft_2d(ny, nx, in, ri, FFTW_FORWARD, FFTW_ESTIMATE);

/* Pack the CPL image data into the FFTW input array */
for(i=0;i<nx*ny;i++) { in[i][0] = *(data_in+i); in[i][1] = 0.0; }


/* Actually perform the FFT */
fftw_execute(p);


/* Put the real output from the FFT into a CPL image */
data_re = malloc(nx*ny*sizeof(double));
for(i=0;i<nx*ny;i++) { *(data_re+i) = ri[i][0]; }
img_re = cpl_image_new_double(nx, ny, data_re, NULL);

/* Put the imaginary output from the FFT into a CPL image */
data_im = malloc(nx*ny*sizeof(double));
```

```
  for(i=0;i<nx*ny;i++) { *(data_im+i) = ri[i][1]; }
  img_im = cpl_image_new_double(nx, ny, data_im, NULL);


  /* Cleanup FFTW resources */
  fftw_destroy_plan(p);
  fftw_free(in);
  fftw_free(ri);

  /* Do any further processing of the CPL images. Here we simply save them */
  cpl_image_save(img_in, "in.fits", BPP_DEFAULT, NULL);
  cpl_image_save(img_re, "re.fits", BPP_DEFAULT, NULL);
  cpl_image_save(img_im, "im.fits", BPP_DEFAULT, NULL);


  /* Deallocate the CPL images/data */
  cpl_image_delete_but_data(img_re);
  cpl_image_delete_but_data(img_im);
  free(data_re);
  free(data_im);

  /* Deallocate the original image */
  cpl_image_delete(img_in);

  /* Terminate the program */
  return (EXIT_SUCCESS);

} /* End of main() */
```

C Source code for `rrrecipe.c`

```
/* $Id: rrrecipe.c,v 1.1.1.1 2004/03/15 08:23:07 yjung Exp $
 *
 * This file is part of the IIINSTRUMENT Pipeline
 * Copyright (C) 2002,2003 European Southern Observatory
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/*
 * $Author: yjung $
 * $Date: 2004/03/15 08:23:07 $
 * $Revision: 1.1.1.1 $
 * $Name:  $
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

/*-----------------------------------------------------------------------------
                                Includes
 -----------------------------------------------------------------------------*/

#include <math.h>
#include <qfits.h>
#include <cpl_memory.h>
#include <cpl_recipe.h>
#include <cpl_plugininfo.h>
#include <cpl_parlist.h>
#include <cpl_frameset.h>
#include <cpl_frame.h>
#include <cpl_plist.h>
#include <cpl_image_gen.h>

#include "iiinstrument_utils.h"
#include "iiinstrument_pfits.h"
#include "iiinstrument_dfs.h"
```

```
/*-----------------------------------------------------------------------------
                            Functions prototypes
 ----------------------------------------------------------------------------*/

static int rrrecipe_create(cpl_plugin *) ;
static int rrrecipe_exec(cpl_plugin *) ;
static int rrrecipe_destroy(cpl_plugin *) ;
static int rrrecipe(cpl_parlist *, cpl_frameset *) ;

static cpl_image * rrrecipe_reduce(cpl_frameset *, char *, char *) ;
static int rrrecipe_save(cpl_image *, cpl_frameset *) ;

/*-----------------------------------------------------------------------------
                            Static variables
 ----------------------------------------------------------------------------*/

static struct {
    /* Inputs */
    int         bool_option ;
    char        str_option[512] ;

    /* Outputs */
    double      qc_param ;
} rrrecipe_config ;

static char rrrecipe_man[] =
"This is the desciption of the rrrecipe recipe\n"
"\n";

/*-----------------------------------------------------------------------------
                                Functions code
 ----------------------------------------------------------------------------*/

/*----------------------------------------------------------------------------*/
/**
  @brief    Build the list of available plugins, for this module.
  @param    list    the plugin list
  @return   0 if everything is ok

  This function is exported.
 */
/*----------------------------------------------------------------------------*/
int cpl_plugin_get_info(cpl_pluginlist * list)
{
    cpl_recipe  *   recipe = cpl_calloc(1, sizeof(*recipe)) ;
    cpl_plugin  *   plugin = &recipe->interface ;

    cpl_plugin_init(plugin,
                    CPL_PLUGIN_API,
                    IIINSTRUMENT_BINARY_VERSION,
```

```
                    CPL_PLUGIN_TYPE_RECIPE,
                    "rrrecipe",
                    "Short description of rrrecipe",
                    rrrecipe_man,
                    "Hans Kruger",
                    "hkruger@paulaner.de",
                    iiinstrument_get_license(),
                    rrrecipe_create,
                    rrrecipe_exec,
                    rrrecipe_destroy) ;

    cpl_pluginlist_append(list, plugin) ;

    return 0;
}

/*----------------------------------------------------------------------------*/
/**
  @brief    Setup the recipe options
  @param    plugin  the plugin
  @return   0 if everything is ok

  Create the recipe instance and make it available to the application using the
  interface.
 */
/*----------------------------------------------------------------------------*/
static int rrrecipe_create(cpl_plugin * plugin)
{
    cpl_recipe * recipe = (cpl_recipe *)plugin ;
    cpl_parameter * p ;

    /* Create the parameters list in the cpl_recipe object */
    recipe->parameters = cpl_parlist_new() ;

    /* Fill the parameters list */
    /* --stropt */
    p = cpl_parameter_value_new("iiinstrument.rrrecipe.str_option",
            CPL_TYPE_STRING, "the string option", "iiinstrument.rrrecipe",NULL);
    cpl_parameter_set_alias(p, "stropt", NULL, NULL) ;
    cpl_parlist_append(recipe->parameters, p) ;

    /* --boolopt */
    p = cpl_parameter_value_new("iiinstrument.rrrecipe.bool_option",
            CPL_TYPE_BOOL, "a flag", "iiinstrument.rrrecipe", TRUE) ;
    cpl_parameter_set_alias(p, "boolopt", NULL, NULL) ;
    cpl_parlist_append(recipe->parameters, p) ;

    /* Return */
    return 0;
}
```

```
/*----------------------------------------------------------------------------*/
/**
  @brief    Execute the plugin instance given by the interface
  @param    plugin  the plugin
  @return   0 if everything is ok
 */
/*----------------------------------------------------------------------------*/
static int rrrecipe_exec(cpl_plugin * plugin)
{
    cpl_recipe * recipe = (cpl_recipe *)plugin ;
    return rrrecipe(recipe->parameters, recipe->frames) ;
}

/*----------------------------------------------------------------------------*/
/**
  @brief    Destroy what has been created by the 'create' function
  @param    plugin  the plugin
  @return   0 if everything is ok
 */
/*----------------------------------------------------------------------------*/
static int rrrecipe_destroy(cpl_plugin * plugin)
{
    cpl_recipe  *   recipe = (cpl_recipe *)plugin ;
    cpl_parlist_delete(recipe->parameters) ;
    return 0 ;
}

/*----------------------------------------------------------------------------*/
/**
  @brief    Get the command line options and execute the data reduction
  @param    parlist     the parameters list
  @param    framelist   the frames list
  @return   0 if everything is ok
 */
/*----------------------------------------------------------------------------*/
static int rrrecipe(
        cpl_parlist     *   parlist,
        cpl_frameset    *   framelist)
{
    const char      *   fctid = "rrrecipe" ;
    cpl_parameter   *   par ;
    char            *   flat ;
    char            *   badpix ;
    int             *   labels ;
    int                 nlabels ;
    cpl_frameset    *   rawframes ;
    cpl_frameset    *   cur_set ;
    cpl_frame       *   cur_frame ;
    char            *   tag ;
    cpl_image       *   prod_ima ;
    int                 i ;
```

```
/* Initialise */
par = NULL ;
badpix = NULL ;
flat = NULL ;
rawframes = NULL ;
rrrecipe_config.qc_param = -1.0 ;

/* Retrieve input parameters */
/* --stropt */
par = cpl_parlist_find(parlist, "iiinstrument.rrrecipe.str_option") ;
strcpy(rrrecipe_config.str_option, cpl_parameter_get_string(par)) ;

/* --boolopt */
par = cpl_parlist_find(parlist, "iiinstrument.rrrecipe.bool_option") ;
rrrecipe_config.bool_option = cpl_parameter_get_bool(par) ;

/* Retrieve calibration data */
/* Labelise the input frames according to their tags */
if ((labels = cpl_frameset_labelise(framelist, iiinstrument_compare_tags,
                &nlabels)) == NULL) {
    cpl_msg_error(fctid, "Cannot labelise the input frames") ;
    return -1 ;
}
if (nlabels == 1) {
    /* Only one label - all images are raw frames */
    cur_frame = cpl_frameset_get_frame(framelist, 0) ;
    tag = (char*)cpl_frame_get_tag(cur_frame) ;
    if (rrrecipe_is_raw(tag)) {
        rawframes = cpl_frameset_copy(framelist) ;
    }
} else {
    /* For each label */
    for (i=0 ; i<nlabels ; i++) {
        cur_set = cpl_frameset_extract(framelist, labels, i) ;
        cur_frame = cpl_frameset_get_frame(cur_set, 0) ;
        tag = (char*)cpl_frame_get_tag(cur_frame) ;
        if (rrrecipe_is_raw(tag) == 1) {
            /* Raw frames */
            rawframes = cpl_frameset_copy(cur_set) ;
        } else if (iiinstrument_is_flat(tag)) {
            /* Flatfield calibration file */
            flat = cpl_strdup(cpl_frame_get_filename(cur_frame)) ;
        } else if (iiinstrument_is_badpix(tag)) {
            /* Bad pixels calibration file */
            badpix = cpl_strdup(cpl_frame_get_filename(cur_frame)) ;
        }
        cpl_frameset_delete(cur_set) ;
    }
}
cpl_free(labels) ;
```

```
    /* Test if the raw frames have been found */
    if (rawframes == NULL) {
        cpl_msg_error(fctid, "Cannot find raw frames in the input list") ;
        if (badpix) cpl_free(badpix) ;
        if (flat) cpl_free(flat) ;
        return -1 ;
    }

    /* Apply the reduction */
    cpl_msg_info(fctid, "Apply the data reduction") ;
    cpl_msg_indent_more() ;
    if ((prod_ima = rrrecipe_reduce(rawframes, flat, badpix)) == NULL) {
        cpl_msg_error(fctid, "Cannot reduce the data") ;
        cpl_frameset_delete(rawframes) ;
        if (badpix) cpl_free(badpix) ;
        if (flat) cpl_free(flat) ;
        cpl_msg_indent_less() ;
        return -1 ;
    }
    cpl_frameset_delete(rawframes) ;
    if (badpix) cpl_free(badpix) ;
    if (flat) cpl_free(flat) ;
    cpl_msg_indent_less() ;

    /* Save the products */
    cpl_msg_info(fctid, "Save the products") ;
    cpl_msg_indent_more() ;
    if (rrrecipe_save(prod_ima, framelist) == -1) {
        cpl_msg_error(fctid, "Cannot save the products") ;
        cpl_image_delete(prod_ima) ;
        cpl_msg_indent_less() ;
        return -1 ;
    }
    cpl_msg_indent_less() ;

    /* Return */
    cpl_image_delete(prod_ima) ;
    return 0 ;
}

/*----------------------------------------------------------------------------*/
/**
  @brief    The recipe data reduction part is implemented here
  @param    raw         the raw frames
  @param    flat        the flat field or NULL
  @param    bpm         the bad pixels map or NULL
  @return   the produced image or NULL in error case
 */
/*----------------------------------------------------------------------------*/
static cpl_image * rrrecipe_reduce(
```

```
        cpl_frameset    *   raw,
        char            *   flat,
        char            *   bpm)
{
    cpl_frame       *   frame ;
    char            *   fname ;
    char            *   sval ;
    cpl_image       *   prod_ima ;

    /* The QC parameter is the DIT retrieved from the header */
    frame = cpl_frameset_get_frame(raw, 0) ;
    fname = (char*)cpl_frame_get_filename(frame) ;
    sval = iiinstrument_pfits_get_dit(fname) ;
    rrrecipe_config.qc_param = (double)atof(sval) ;

    /* The image is a generated one */
    prod_ima = cpl_image_gen_test(100, 100) ;

    /* Return */
    return prod_ima ;
}

/*----------------------------------------------------------------------------*/
/**
  @brief    Save the rrrecipe products on disk
  @param    ima         the image produced
  @param    set         the input frame set
  @return   0 if everything is ok, -1 otherwise
 */
/*----------------------------------------------------------------------------*/
static int rrrecipe_save(
        cpl_image       *   ima,
        cpl_frameset    *   set)
{
    const char      *   fctid = "rrrecipe_save" ;
    char            **  fnames ;
    int                 nfnames ;
    char                name_o[512] ;
    char                cval[512] ;
    cpl_plist       *   plist ;
    cpl_frame       *   product_frame ;
    FILE            *   paf ;
    char            *   sval ;

    /* Get the files names */
    if ((fnames = cpl_frameset_to_filenames(set, &nfnames)) == NULL) {
        cpl_msg_error(fctid, "getting file names") ;
        return -1 ;
    }

    /**********************/
```

```
/* Write the FITS file */
/***********************/
/* Set the file name */
sprintf(name_o, "rrrecipe.fits") ;
cpl_msg_info(fctid, "Writing %s" , name_o) ;

/* Get FITS header from reference file */
plist = cpl_plist_new() ;
if (cpl_plist_load(plist, fnames[0], 0)) {
    cpl_msg_error(fctid, "getting header from reference frame");
    cpl_free(fnames) ;
    cpl_plist_delete(plist) ;
    return -1 ;
}

/* Correct the header */
cpl_plist_set_int(plist, "NAXIS", 2) ;
cpl_plist_set_int(plist, "NAXIS1", cpl_image_get_nx(ima)) ;
cpl_plist_set_int(plist, "NAXIS2", cpl_image_get_ny(ima)) ;
cpl_plist_erase(plist, "NAXIS3") ;

/* Add DataFlow keywords */
iiinstrument_add_pro(plist, fnames[0], "REDUCED", NULL, RRRECIPE_IMA,
        "Ok", "rrrecipe", fnames, NULL, nfnames, NULL, NULL, 0) ;

/* Save list of input files as HISTORY in the header */
iiinstrument_add_inputfiles(plist, fnames, nfnames) ;

/* Add QC params as history keyword */
sprintf(cval, "QC.QCPARAM = %g", rrrecipe_config.qc_param) ;
cpl_plist_append_string(plist, "HISTORY", cval) ;

/* Clean the useless keywords */
cpl_plist_erase_regexp(plist, "ESO DET VOLT*");
cpl_plist_erase_regexp(plist, "ESO DET IRACE");

/* Save the file */
if (cpl_image_save(ima, name_o, BPP_DEFAULT, plist)!=CPL_ERROR_NONE) {
    cpl_msg_error(fctid, "Cannot save the product");
    cpl_plist_delete(plist) ;
    return -1 ;
}
cpl_plist_delete(plist) ;

/* Log the saved file in the input frameset */
product_frame = cpl_frame_new() ;
cpl_frame_set_filename(product_frame, name_o) ;
cpl_frame_set_tag(product_frame, RRRECIPE_IMA) ;
cpl_frame_set_type(product_frame, CPL_FRAME_TYPE_IMAGE);
cpl_frame_set_group(product_frame, CPL_FRAME_GROUP_PRODUCT);
cpl_frame_set_level(product_frame, CPL_FRAME_LEVEL_FINAL);
```

```
    cpl_frameset_insert(set, product_frame);

    /**********************/
    /* Write the PAF file */
    /**********************/
    /* Set the file name */
    sprintf(name_o, "rrrecipe.paf") ;
    cpl_msg_info(fctid, "Writing %s" , name_o) ;

    /* Create the default PAF header */
    if ((paf = qfits_paf_print_header(name_o,
                    "IIINSTRUMENT/rrrecipe",
                    "QC file",
                    qfits_get_login_name(),
                    qfits_get_datetime_iso8601())) == NULL) {
        cpl_msg_error(fctid, "cannot open file [%s] for output", name_o) ;
        cpl_free(fnames) ;
        return -1 ;
    }

    /* Add ARCFILE */
    if ((sval = iiinstrument_pfits_get_arcfile(fnames[0])) != NULL)
        fprintf(paf, "ARCFILE         \"%s\" ;#\n", sval) ;

    /* QC.QCPARAM */
    fprintf(paf, "QC.QCPARAM       \"%.4f\"\n", rrrecipe_config.qc_param) ;

    fprintf(paf, "\n");
    fclose(paf) ;

    /* Log the saved file in the input frameset */
    product_frame = cpl_frame_new() ;
    cpl_frame_set_filename(product_frame, name_o) ;
    cpl_frame_set_tag(product_frame, RRRECIPE_QC) ;
    cpl_frame_set_type(product_frame, CPL_FRAME_TYPE_PAF);
    cpl_frame_set_group(product_frame, CPL_FRAME_GROUP_PRODUCT);
    cpl_frame_set_level(product_frame, CPL_FRAME_LEVEL_FINAL);
    cpl_frameset_insert(set, product_frame);

    /* Return */
    return 0 ;
}

/* End of file */
```

Index

— End of document —