Pipeline Development with CPL

Ralf Palsa

Outline

- The Common Pipeline Library
- System Requirements
- Interface to Data Reduction Pipelines
- Developing with CPL
- Roadmap
- Questions
- Coding Standards

The Common Pipeline Library

- **What it is:**
  - The CPL is an SDK organized into a hierarchical set of ISO/IEC 9899:1999(E) compliant libraries, released under GPL2+.
  - Basic data types, low level services and operations (***cplcore***)
  - Application and user interface related types and utilities (***cplui***)
  - Standard implementations of astronomical data reduction tasks (***cpldrs***) → ***HDRL***
  - Data Flow System related utilities (***cpldfs***)
- **What it is not:**
  - It is not a complete data reduction environment or a general purpose image processing library!

The Common Pipeline Library

- **Its purpose is to:**
  - standardize the development of VLT/ELT instrument pipelines (the way they are built, and the way they are developed),
  - have a better idea on what is delivered to ESO as a final product,
  - provide a basic, well tested set of extensible software components to pipeline developers in order to
    - allow a rapid implementation of data reduction tasks,
    - shorten the pipeline development cycle,
    - ease pipeline maintenance over their lifetime (10+ yrs),
  - promote software re-use across pipelines.

The Common Pipeline Library

- **Getting CPL:**
  - http://www.eso.org/sci/software/cpl
  - HTML API Reference documentation is included in the distribution
  - API Reference and User Manual (update is in progress) is also available on the web site

System Requirements

- **Target Environment:**
  - VLT/(ELT) Standard Platform, i.e. CentOS 7.3 (64bit)
  - Unix(like), POSIX compliant operating system (any current Linux distribution, Mac OS X)
- **Runtime Dependencies:**
  - CFITSIO library (3.350 or newer)
  - FFTW (3.3.4 or newer)
  - Mark Calabretta's wcslib (4.24 or newer)
- **Additional Runtime Dependencies for Pipelines:**
  - GNU gsl 2.1
  - Anything else needs to be approved by ESO

System Requirements

- **Developer Tools**

Tool	VLT Platform Baseline	Recommended (for compatibility)
GNU GCC	4.8.5	7.x, 8.x, 9.x
GNU make	3.82	
GNU gdb	7.6.1	8.2
Autoconf	2.69	2.69
Automake	1.13.4	1.16.1
Libtool	2.4.2	2.4.6
Doxygen	1.8.5	1.8.14
Valgrind	3.11.0	3.11.0
pdflatex		
...		

System Requirements

- **Hardware Limitations:**
  - What is defined by the ESO IT Standard.
  - Anything else needs to be approved by ESO
- **VLT Standard Platform (as of April 2018):**
  - Dell M630
  - 2 x Intel Xeon E5-2697 v4, 2.3 GHz
  - 128GB RAM (UT1, UT2, UT3, VST, VISTA, Espresso),
  - 256GB (UT4, VLTI)
  - 300GB storage (internal disk) + external storage (fileserver)

Interface to Data Reduction Pipelines

- **CPL Plugin Interface:**
  - Pipeline recipes are implemented as dynamically loaded modules, i.e. **plugins**
  - It standardizes the pipeline recipe interface
  - Recipes can be implemented without a detailed knowledge of the run-time environment (Paranal, Garching, at home)
  - Different front-ends can use the same recipe installation
  - Updating a recipe installation has no impact on the rest of the system: no re-compilation of other parts of the system is required

Interface to Data Reduction Pipelines

- **Pipeline Frontends:**
  - Command line utility ***EsoRex***
  - File browser ***Gasgano***
  - Scientific workflow ***Reflex*** (using EsoRex)

Sharing the same single pipeline installation!

Developing with CPL

- **Pipeline Recipes High Level Requirements:**
  - Open source, GPL2+ license
  - Compliant with the VLT/ELT Standard Environment (hardware, software, OS, compiler, etc.)
  - Pipelines must be implemented in C following the ISO/IEC 9899:1999(E) standard (→ to be aligned with ELT C standard).
  - CPL must be used whenever it is possible, in particular FITS file I/O shall be done exclusively through CPL.
  - Coding style should follow the conventions used for CPL, i.e it should follow:

    "*Recommended C Style and Coding Standards*", L.W. Cannon, et al., 15th March 2000, (modified version of the Indian Hill C Style and Coding Standards)

Developing with CPL

- **Start a new pipeline project from the:**
  - *iiinstrument* template,
  - an already setup SVN repository (in general your code will be hosted in the ESO SVN repository)

Both will provide a standard pipeline directory tree (it is a subdirectory in the SVN repository setup), containing a recipe skeleton which should be used as a starting point.

Developing with CPL

- **The standard pipeline tree contains 2 main directories:**
  - *<instrument name>*, e.g. the template uses *iiinstrument*
    - Type definitions
    - Utility functions
    - Data reduction task implementations
    - Instrument specific CPL extensions
    - Implements the instrument DRS library
  - *recipes*
    - Implementations of the plugin interface functions for each recipe
    - Implementations of the sequence of data reduction tasks (bias subtraction, flat field correction, ...)

Developing with CPL

- **Pipeline Recipes are implemented as CPL plugins using the CPL recipe datatype:**
  - A CPL recipe is derived from a CPL plugin, and provides the hooks for the plugin interface functions (fixed calling calling sequence):
    - `example_flat_create()`: recipe initialization, including parameter definition
    - `example_flat_exec()`: execution of data reduction sequence by calling one or more data processing functions
    - `example_flat_destroy()`: recipe cleanup
    - `cpl_plugin_get_info()`: Creates the plugin instance. Fixed function name!
  - It provides 2 data members for data I/O
    - `cpl_parameterlist`: Used to pass configuration data (options) to the recipe. (input only). Created by `example_flat_create()`, cleaned up by `example_flat_destroy()`
    - `cpl_frameset`: Used to pass data frames to and from the recipe (input/output). Managed (created/destroyed) by the application invoking the plugin (*EsoRex*)

Developing with CPL

- **Outline of the sequence of data reduction tasks, e.g. an implementation of an `example_flat()` recipe:**
  1. Identify raw and calibration frames
  2. Get recipe configuration options and required data from the recipe's parameter list and the frame set
  3. Execute the algorithm (i.e. call to DRS library function)
  4. Compute QC parameters (quality control information)
  5. Add DFS and QC keywords to the appropriate product header
  6. Create a product frame for each product and add it to the frame set
  7. Save each product as a DFS compliant local file
  8. Repeat steps 2 – 7 as needed

Developing with CPL

- **Main CPL Classes and Services:**
  - Recipe I/O:
    - `cpl_parameterlist`: for recipe parameters
    - `cpl_frameset`: for files
  - Basic data types:
    - `cpl_image`: 2D pixel data array manipulation
    - `cpl_table`: Table data handling (e.g. catalog information)
    - `cpl_propertylist`: Meta data handling (keyword value pairs, e.g. FITS header information)

Developing with CPL

- **Main CPL Classes and Services (cont.):**
  - Basic mathematical types:
    - `cpl_vector`
    - `cpl_matrix`
    - `...`
  - Error handling
  - Interfaces to FFTW and wcslib
  - Utilities to create DFS compliant products

Developing with CPL

- **DFS Compliant Recipe Products:**
  - All products must be FITS files
  - They must be saved to the current working directory
  - Local file names should be predictable (naming and order)
  - They have to comply to the *Data Interface Control Document* (ESO-044156)
  - They have to comply to the *Science Data Products Standard* (ESO-044286)

Developing with CPL

- **DFS Compliant Recipe Products (cont.):**
  - DFS compliant products
    - Inherit keywords from the first raw frame (in general).
    - Header(s) contain the standard PRO keywords (ESO product dictionary 1.16)
    - Headers contain the QC quality control parameters (QC dictionary is a deliverable)
    - Headers may contain DRS related keywords (ESO.DRS prefix, if used the DRS dictionary is a deliverable)
  - CPL product saving functions will create DFS compliant products:
    - `cpl_dfs_save_image()`
    - `cpl_dfs_save_table()`
    - `cpl_dfs_save_propertylist()`
    - `cpl_dfs_setup_product_header()`

Developing with CPL

- **Building recipes:**
  - Bootstrapping the build tree:
    - `autogen.sh`
    - `configure --enable-maintainer-mode`
    - The build tree must support out-of-source builds, i.e. using a separate build directory
  - Building the tree, initial checks
    - `make`
    - `make install`
    - `make doxygen` (optional)
    - `make check` (for unit tests, self-contained, no external dependencies including data)

Developing with CPL

- **Running recipes:**
  - For developers using *EsoRex* is recommended
    - it is the most flexible tool
    - can easily be used with the debugger
  - General *EsoRex* syntax:

```
esorex [esorex options] [<recipe> [recipe options] [sof]]
```

  - Useful options:

```
esorex --help

esorex --recipes

esorex --man-page <recipe>
```

Developing with CPL

- Set of Frames file format:

```
/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:32:47.341.fits ARC
#/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:33:50.290.fits ARC
/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:38:05.933.fits ARC
#/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:39:26.889.fits ARC
/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:44:57.510.fits ARC
#/diskb/data_muse/raw/2014-10-14/MUSE.2014-10-14T10:46:00.568.fits ARC
$MUSE_CAL/muse_line_catalog.fits LINE_CATALOG
$MUSE_CAL/MUSE_MASTER_BIAS_slow.fits MASTER_BIAS
$MUSE_CAL/MUSE_MASTER_FLAT_slow_wfm-e.fits MASTER_FLAT
$MUSE_CAL/MUSE_TRACE_TABLE_slow_wfm-e.fits TRACE_TABLE
```

Developing with CPL

- **Debugging Recipes:**
  - Requires debug build of CPL, *EsoRex* and the recipes
  - Run `configure` with `–enable-debug`
  - Pass debugging flags to configure:

```
configure ... CFLAGS=”-pipe -rdynamic -g3 -ggdb -O0 -fno-builtin -Wextra -Wall -W -Wcast-align -Winline -Wmissing-noreturn -Wpointer-arith -Wshadow -Wsign-compare -Wundef -Wunreachable-code -Wwrite-strings -Wmissing-field-initializers -Wmissing-format-attribute”
```

  - Use the debugger (can even serve as a C interpreter)
  - Use valgrind, ThreadSanitizer, AddressSanitizer to check for memory leaks and/or memory access violations.

Developing with CPL

- **The Build System – what needs to be maintained?**
  - `configure.ac`:
    - Package and library version
    - Additional feature tests
    - Adding or removing files/directories to configure
  - `Makefile.am`:
    - Adding/removing source files to/from libraries, programs
    - Adding/removing subdirectories
    - Adapting compiler/linker flags and options
  - (`acinclude.m4`)

Developing with CPL

- **Support for multi-threaded recipe execution:**
  - Multi-threaded pipeline recipes may be implemented using OpenMP. Only what is compliant with the baseline compiler may be used!
  - CPL is thread-safe (unless otherwise stated), i.e. it properly manages access to it's internal data structures, it does **not** do this for objects created by the recipes
  - Managing access to shared data structures in the recipes is your responsibility and requires an appropriate design
  - If there is no need to use it do not use it
  - If you use it use thread analysis tools extensively!

Developing with CPL

- **Best Practices, Tips & Pitfalls:**
  - CPL is not designed to update parts of an existing file other than adding FITS extensions at the end.
  - Do not use high-level data objects to feed low-level functions, but progressively use less complex objects the lower a function is in the hierarchy.
  - Be explicit when writing code and express your intentions in the code, e.g. use `const` when you deal with objects that should not be modified.
  - Avoid obscure constructs, e.g. pointer aliasing.
  - The number of function parameters should be small. If necessary related parameters should be grouped into structures.
  - Function implementations should be reasonably short (no “spaghetti code”). If necessary it should be divided into individual tasks coded as individual functions.

Developing with CPL

- **Best Practices, Tips & Pitfalls (cont.):**
  - Take the time to write good source code documentation, in particular be verbose when it comes to the dark corners in your code.
  - Take the time to write concise but informative log-messages when you commit changes to the repository.
  - Commit often and individual changes. Avoid committing massive changes as a single commit.

Roadmap

- **Development of CPL:**
  - Development is mostly driven be the needs of new instruments, e.g. thread-safety was required for MUSE → ELT Instruments
  - If new features are needed please keep the CPL release cycle in mind (once per year, around the end of March for major updates)
  - If you need a new feature, ask us and ask us as early as possible
  - Future developments:
    - Python bindings for CPL/HDRL (approved)
    - HARMONI, MICADO, METIS (TBD)

Questions

- **Questions?**

Coding Standards

- **Readability:**
  - Maximum characters per line is 80.
  - No tabulators may be used for indentation.
  - The indentation width is 4 spaces.
  - Non-empty blocks shall use K&R style
  - Only one statement per line.
  - Use blanks around all binary operators, except "`->`" and "`.`".
  - Use a blank after comma, colon, semicolon, control flow keywords.
  - Don't use blanks between an identifier and "`(`", "`)`", "`[`" and "`]`" or before a semicolon.
  - No spaces after pointer operators "`*`" and "`&`". In variable declarations the "`*`" shall be placed adjacent to the variable name.

Coding Standards

- **Naming Conventions:**
  - General:
    - Function and variable names shall be all lower case letters, with words separated by an underscore.
    - Use clear and informative names for functions and variables.
  - Functions:
    - Function names must be prefixed with the instrument acronym followed by an underscore.
    - Function names must identify the action performed, or the information provided.
  - Variables:
    - Variable names should be short, but meaningful.
    - Variables should be named with their content.

Coding Standards

- **Naming Conventions (cont):**
  - File Names:
    - Header file names have the extension `.h`
    - Source file names have the extension `.c`
    - Header and source file names are prefixed with the instrument acronym followed by an underscore.
  - Other Names:
    - Preprocessor symbols and enumeration constants should be all upper case letters, with words separated by "_".

Coding Standards

- **Types, Variables, Operators and Expressions:**
  - Use the same name for the structure tag and the typedef name, i.e.:

```
typedef struct my_type my_type;
```

  - Avoid global variables.
  - Variables should be declared in the smallest possible scope.
  - All variables have to be initialized when they are defined (as far as possible).
  - Don't write code that depends on the byte order, or word alignment boundaries of an architecture.

Coding Standards

- **Types, Variables, Operators and Expressions (cont):**
  - Do not use macros unless it is absolutely necessary.
  - Whenever possible use enumeration constants rather than preprocessor symbols.
  - Avoid writing code that requires excessive stack sizes:

```
function(int huge)
{
    double a[huge];

    ...
```

Coding Standards

- **Functions:**
  - ANSI C function prototypes must be used.
  - In the definition the function return type should appear on a separate line
  - All functions should return an error code (as far as possible).
  - If no return value is required, a function has to be declared `void`. The return statement is still required.
- **Statements and Control Flow:**
  - Always provide a default case for `switch` statements and `break` each case of the switch statement.
  - Always use braces to delimit blocks of `if`, `for`, `while` and `do ... while` statements (even if it is just one line or empty).
  - Don't use `goto` unless absolutely necessary (i.e. never).

Coding Standards

- **Code Comments:**
  - All files, in particular source and header files must begin with the standard header (cf. GPL).
  - Functions, global variables, enumerations constants shall be documented.
  - Public functions, data types, enumeration constants, modules must be commented so that a reference manual can be build using *doxygen*.
  - Block comments should be preceded by 2 and followed by 1 empty line, have the same indentation as the code it describes and look like:

```
/*
 *  ...
 *  ...
 */
```

  - Do not break in-line comments into multiple lines.

```
/*...
  ...
  ... */
```

Coding Standards

- **Header files:**
  - Header files should be used as interface specification for a software module.
  - Use code guards to prevent multiple inclusion.
  - Header files shall be self contained.
  - It should be possible to use the header files with a C++ compiler.
  - In function declarations all parameters shall be named and have the same name as in the implementation.

Coding Standards

- **Header files (cont.):**

```
#ifndef FILE_NAME_H
#define FILE_NAME_H

#ifdef _cplusplus
extern “C” {
#endif
...
...
#ifdef _cplusplus
}
#endif

#endif /* FILE_NAME_H */
```

Coding Standards

- **Source Files:**
  - The encoding should be “utf-8” (applies to header files too).
  - Non-ASCII characters should be rare and must use UTF-8 formatting.
  - All comments in source files must be up to date at any time.
  - Code comments must be in English.
  - Comments should give a synopsis of a section of code, outline the steps of an algorithm, or clarify a piece of code when it is not immediately obvious what or why something was done.
  - Don't comment what is already clearly expressed by the code itself, for example:

```
/* return from the function */

 return 0;
```