



Coding Standards



Lars Kr. Lundin



Coding Standards

The Coding Standard for instrument pipelines is the one adopted for CPL.

**It is based on “*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).**



Readability:

- Maximum characters per line is 78.
- No tabulators may be used for indentation.
- The indentation width is 4 spaces.
- Only one statement per line.
- Use blanks around all binary operators, except “->” and “.”.
- Use a blank after comma, colon, semicolon, control flow keywords.
- Do not use blanks between an identifier and “(“, “)”, “[“ and “]” or before a semicolon.



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.



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 “_”.



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.



Types, Variables, Operators and Expressions (cont):

- Avoid using macros.
- Whenever possible use enumeration constants rather than preprocessor symbols.
- Avoid writing code that requires excessive stack sizes:

```
function(int n)
{
    double a[n];
    ...
}
```



Functions:

- ANSI C function prototypes must be used.
- 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).
- Don't use `goto` unless absolutely necessary (i.e. never).



Code Comments:

- All files, in particular source and header files must begin with the standard header (cf. GPL).
- Public functions, data type, 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:

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



Header files:

- Header files should be used as interface specification for a software module.
- Use code guards to prevent multiple inclusion.
- It should be possible to use the header files with a C++ compiler.



Header files (cont):

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

#ifdef __cplusplus
extern "C" {
#endif
...
...
#ifdef __cplusplus
}
#endif

#endif /* MY_HEADER_H */
```



Source Files:

- 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 especially why something was done.
- Comment the '**why**' not the 'what', for example:

```
/* Perform the bias correction */  
cpl_image_subtract(corrected, bias);
```



Function interface is documented with doxygen comments, e.g. :

```
/**
 * @brief
 *   Get a recipe integer parameter value
 *
 * @param parlist      The input parameter list
 * @param recipe_name  The recipe name
 * @param name         The parameter name
 *
 * @return The parameter value
 *
 * @see cpl_parameter_get_int()
 *
 * @note This function will abort() on NULL input
 */
```



El Fin



CVS working cycle:

- Initial checkout or working copy update:

```
$ cvs -d <repository path> co -P <project> or
```

```
$ cvs update -dP
```

- Modify, add and remove files:

```
$ cvs add [-kb] <filename>
```

```
$ cvs rm [-f] <filename>
```



CVS working cycle (cont):

- Checking file status, reading log messages, comparing file versions:

```
$ cvs status [-v] <filename>
```

```
$ cvs log <filename>
```

```
$ cvs diff -r <revision|tag> <filename>
```

- Committing changes:

```
$ cvs ci -m "log message" <filename>
```

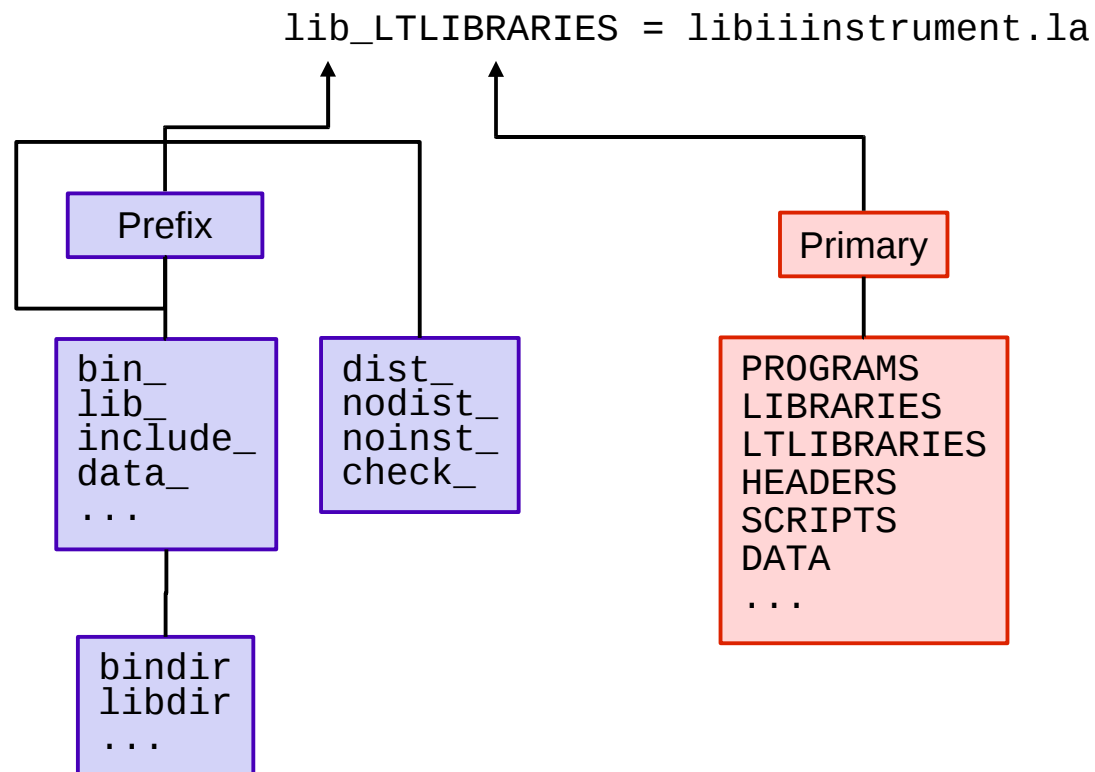



The Concept:

- Reduce Makefile maintenance to the minimum.
- Offer the possibility of adapting a source package to a variety of POSIX like systems.
- Release packages are independent of the build tools.



The Automake Naming Scheme:





Typical Cases:

- Adding a new library (using libtool):

```
INCLUDES = $(CPLCORE_INCLUDES)
lib_LTLIBRARIES = libmylib.la
```

```
include_HEADERS = mylib.h
noinst_HEADERS = file2.h file3.h
```

```
libmylib_la_SOURCES = file1.c file2.c file3.c
libmylib_la_LDFLAGS = $(CPL_LDFLAGS)
libmylib_la_LIBADD = $(LIBCPLCORE)
```

- Adding a program:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```



Typical Cases (cont):

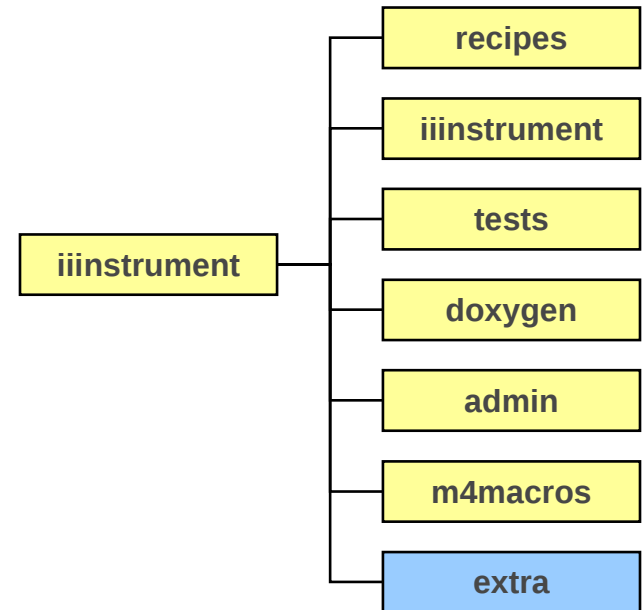
- Adding a subdirectory extra:
 - ▶ Create the directory.
 - ▶ Put the `Makefile.am` into `extra`.
 - ▶ Update `configure.ac`:

```
AC_CONFIG_FILES(Makefile
                 doxygen/Doxyfile
                 iinstrument/Makefile
                 recipes/Makefile
                 extra/Makefile
                 tests/Makefile)
```

- ▶ Update the `Makefile.am` in the top level directory:

```
SUBDIRS = iinstrument extra recipes
```

- ▶ Re-run `bootstrap`, `configure` and `make`





CVS – Keep Track of your Work

Each Instrument Pipeline has/gets a CVS repository on the ESO server (after FDR at the latest):

- This allows you and us to keep track of the project.
- The code is checked every night with our in house verification tools
- Problems can be caught early and we can advise you properly.
- If needed we can assist in the repository administration.



CVS – Keep Track of your Work

CVS best practices:

- Run cvs commands on individual files only.
- Write useful log messages (**don't leave them empty**). It should contain the function or variable which is affected and what has changed:
`my_function(): variable 'a' replaced by 'A'`
`function_a(), function_b(): argument 'setup' added`
- All changes committed to the repository must (at least) compile.
- **Commit often.**
- Use tags when preparing a release (`cvs tag iinstrument-1_3`)



CVS – Keep Track of your Work

CVS best practices (cont):

- Tag your project frequently (before major changes, `iiinstrument-20070419`).
- Don't prepare releases from your working copy, make use of `cvs export`.
- Consider branches to separate patch releases from the main development line.



The tools:

- autoconf (2.59, 2.60)
- automake (automake 1.9.6)
- libtool (1.5.22)

Installation:

- Use the same installation prefix (`configure --prefix` option) for all 3 tools.
- The tools must be installed in the order autoconf, automake and libtool.
- Adjust the PATH after the installation of autoconf.



What is needed to start:

- A source tree (with standard layout as in `iiinstrumentp`).
- `configure.ac`: The template of the configure script.
- `Makefile.am`: The Makefile template (per directory).
- `(acinclude.m4)`: Local m4 macro definitions used by configure.



How does it work:

- Checkout of the project from CVS.
- To create the `configure` script and the input Makefiles (`Makefile.in`) run `aclocal`, `autoheader`, `libtoolize`, `automake` and `autoconf` in the right order.
→ **`bootstrap` and/or `autogen.sh` (or `autoreconf`)**
- Run `configure` with the appropriate options.
 - ▶ In particular use `--enable-maintainer-mode` to allow an automatic update of the build system



How does it work (cont):

- Run make targets as needed:
 - ▶ make
 - ▶ make install/uninstall
 - ▶ make clean
 - ▶ make dist
 - ▶ make maintainer-clean
 - ▶ make html/clean-html
- To compile an individual file, `hello.c` (using `libtool`) use:
 - ▶ make `hello.lo`
- To build a single recipe/library (using `libtool`):
 - ▶ make `hello.la`



What needs to be maintained:

- `configure.ac`:
 - ▶ Package and library version
 - ▶ Additional feature tests
 - ▶ Adding or removing files/directories to `configure` (cf. Carlo's talk)
- `Makefile.am`:
 - ▶ Adding/removing source files to/from libraries, programs
 - ▶ Adding/removing subdirectories (cf. Carlo's talk)
 - ▶ Adapting compiler/linker flags and options
- `(acinclude.m4)`



configure.ac:

```
AC_INIT([IIINSTRUMENT Instrument Pipeline], [0.0.1] [flastname@eso.org], [iiinstrument])
AC_PREREQ([2.59])
...
...
# Immediately before every release do:
#-----
#   if (the interface is totally unchanged from previous release)
#       REVISION++;
#   else {
#       /* interfaces have been added, removed or changed */
#       REVISION = 0;
#       CURRENT++;
#       if (any interfaces have been _added_ since last release)
#           AGE++;
#       if (any interfaces have been _removed_ or incompatibly changed)
#           AGE = 0;
#   }
# Order of arguments: VERSION, CURRENT, REVISION, AGE
IIINSTRUMENT_SET_VERSION_INFO([$VERSION], [0], [0], [0])
```



Makefile.am:

- Adding/Removing sources:

```
lib_LTLIBRARIES = libiiinstrument.la
pkginclude_HEADERS = file1.h file2.h file3.h
libiiinstrument_la_SOURCES = file1.c file2.c file3.c
```

- Adding a plugin library (recipe):

```
plugin_LTLIBRARIES = recipe1.la recipe2.la
recipe1_la_SOURCES = recipe1.c
recipe1_la_LIBADD = $(LIBIIINSTRUMENT)
recipe1_la_LDFLAGS = -module -avoid-version
recipe1_la_DEPENDENCIES = $(LIBIIINSTRUMENT)
recipe2_la_SOURCES = recipe2.c
recipe2_la_LIBADD = $(LIBIIINSTRUMENT)
recipe2_la_LDFLAGS = -module -avoid-version
recipe2_la_DEPENDENCIES = $(LIBIIINSTRUMENT)
```



Makefile.am (cont):

- Use `AM_CFLAGS` instead of `CFLAGS` in the `Makefile.am`:
 - ▶ `CFLAGS` is a user variable, intended to override flags set in the `Makefile`
 - ▶ Compiler calls are constructed so that overriding the `Makefile` is possible, don't ignore that.