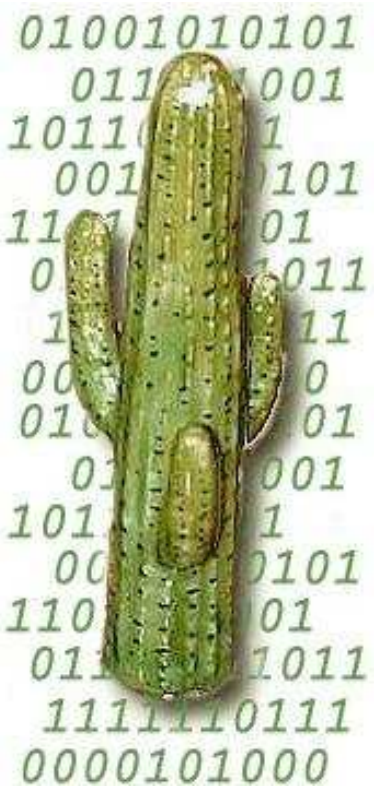
Cactus 4.10

Maintainers' Guide DRAFT VERSION



commit bfcb6cc4d3d1988b1f57602bd0849995434752c0

Documentation compiled on: December 6, 2021

Contents

Preface

This document should describe the Cactus flesh. In particular it should describe

- The philosophy of the flesh
- The coding style used
- The make system
- The various source directories and all their files
- The perl scripts

In addition it should contain ideas for future enhancements.

Overview of documentation

This guide covers the following topics

Part A: Philosphy and Style.
The philosophy behind the flesh and the coding style used.

Part B: The Make System.
The nitty-gritty of the make system.

Part C: The CST.
The nitty-gritty of the CST.

Part D: General. General miscellaneous things used all over the flesh.

Part E: Main. Everything you never wanted to know about the files in the Main subdirectory of the flesh.

Part F: Comm. Everything you never wanted to know about the files in the Comm subdirectory of the flesh.

Part G: IO. Everything you never wanted to know about the files in the IO subdirectory of the flesh.

Part H: Util. Everything you never wanted to know about the various utility files.

Part I: Schedule. Everything you never wanted to know about the Schedule system.

Part J: Appendices.
I'm sure we'll need something here.

Other topics to be discussed in separate documents include:

Computational Thorn Guide
This will contain details about the arrangements and thorns making up the standard Cactus Computation Tool Kit

Relativity Thorn Guide
: This will contain details about the arrangements and thorns making up the Cactus Relativity Tool Kit, one of the major motivators, and still the driving force, for the Cactus Code.

Users' Guide
: The stuff users need to know. This in particular documents the functions the flesh needs to make available to the thorns.

Typographical Conventions

`Typewriter`
: Is currently used for everything you type, for program names, and code extracts.

`< ... >`
: Indicates a compulsory argument.

`[ ... ]`
: Indicates an optional argument.

How to Contact Us

Please let us know of any errors or omissions in this guide, as well as suggestions for future editions. These can be reported via `cactusmaint@cactuscode.org`.

Acknowledgements

Hearty thanks to all those who have helped with documentation for the Cactus Code.

Part A

1

Revision

Chapter A1

Philosophy

- Portable
- Extensable
- Modular

Chapter A2

Coding Style

The flesh has been written with the following coding guidelines; all `Cactus*` thorns should also follow them.

A2.1 Indentation

Two spaces, no tabs.

Two spaces are enough to clearly indent, more would be a waste of space, less not really noticeable.

A2.2 Brace positioning

Each opening brace should appear on a line by itself, aligned with the preceeding statement.

Closing braces should line up with the corresponding opening brace, and should be on lines by themselves apart from the `while` in a

```
do
{

...

} while(...);
```

statement.

This brace positioning stategy makes it easy to run ones eye from a closing or opening brace to its matching opening or closing brace.

Braces should be used for all `if` and `for` statements.

A2.3 GRDOC

All files should start with a grdoc header, and all functions should have grdoc headers.

The file grdoc should contain a description of the contents of the file and a version with the CVS $Header$ tag.

The function grdoc should contain

- a description of the function, saying what it does.
- the functions called by this function.
- all function arguments with descriptions of what they are and what they are used for.
- the return codes should be described.

Note that the 'calledby' field *should not* be filled in as this is unmaintainable.

The standard grdoc function header is of the form

```
 /*@@
   @routine    Template
   @date       Fri Oct  6 10:51:49 2000
   @author     Tom Goodale
   @desc
   An example of grdoc
   @enddesc
   @calls     templatefunc2
   @calledby
   @history

   @endhistory
   @var     templatestring
   @vdesc   string describing foobar
   @vtype   const char *
   @vio     in
   @vcomment

   @endvar

   @returntype int *
   @returndesc
   0 - success
   or the returncode of @seeroutine templatefunc2
   @endreturndesc
@@*/
```

This is the form which will be created if you use the grdoc emacs mode. The variable descriptions and the return code decription should be placed after the history so that they are close to the actual function.

After the first actual release the history should be filled in with each change.

A2.4 Header Files

Header files should not include any system header file, but should contain in the initial comment a list of system header files which are assumed to have been included before this file is included.

The body of a header should be protected against multiple inclusion by lines of the form

```
#ifndef _NAMEOFHEADERFILEINCAPITALS_H_
#define _NAMEOFHEADERFILEINCAPITALS_H_ 1

...body of header file...

#endif /* _NAMEOFHEADERFILEINCAPITALS_H_ */
```

Function prototypes in header files should be protected against C++ linkage by

```
#ifdef __cplusplus
extern "C"
{
#endif

...prototypes...

#ifdef __cplusplus
}
#endif
```

The Cactus header files (`cctk_<name>`) should only include information relevant for thorn programmers.

There is a template file in the doc/MaintGuide directory.

A2.5 Source Files

Source files should have as their first lines after all the include files:

static const char * const rcsid = "$Header$";
CCTK_FILEVERSION(<source file>);

or the expanded rcs version of this. The macro **CCTK_FILEVERSION** is simply there to prevent compiler warnings, and <**source file**> should be replaced by

- Flesh: <**directory**>_<**core filename**>_<**extension**> (e.g. **main_Groups_c**)
- Thorn: <**arrangement**>_<**thorn**>_<**core filename**>_<**extension**>
  (e.g. **CactusBase_CartGrid3D_CartGrid3D_c**)

Globally visable functions should appear before local functions.

(NOTE: currently the schedule stuff is a good example of what I'm coming to like as a style, e.g. src/main/ScheduleInterface.c)

There is a template file in the doc/MaintGuide directory.

A2.6 Naming Conventions

All functions which may be used by thorns should have names beginning with `CCTK_` and then capitalised words with no underscores.

All functions used internally by the flesh should have names beginning with `CCTKi_` and then capitalised words with no underscores.

Header files to be included by thorns should have names beginning with `cctk_`, and followed by capitalised words with no underscores.

Structures which may be used by thorns should have names beginning with `c` and then capitalised words, *e.g.* `cGroup`. The exception here is structures associated with utility routines which are not Cactus specific, there the structure names should start with a `t_`.

Structures which are purely internal to the flesh should have names beginning with `i`.

All Cactus sourcefile names (except general utility files) should use capitilised words without underscores.

A2.7 Functions

All functions should have a grdoc header.

They should have a single place of return at the end of the function to make it easy to tidy up and work out what is going on.

Where possible variables should be declared at the top of the function with no initialisation, and then initialised after all variable declarations. Of course this can't apply to static variables, 'though these should be kept to a minimum so we can make a thread-safe version of Cactus later.

Part B

2

Revision

Note that this whole Chapter is out-dated and needs a rewrite.

Chapter B1

Use of git

Version control in Cactus is maintained by the use of the git software. This software allows one to trace any change to a file from the creation of a file to the present version, and provides an automatic notification system to alert interested parties of changes to files. In order to make effective use of the system, the following commit procedure is recommended as a guideline

Only make one change at a time
: Don't make a commit which changes several distinct things at once, as it is difficult then for people tracing changes back to distinguish which bit was changed for which reason. See the note on commit messages below.

Run the test suite
: This makes sure the code compiles, runs, and produces the correct results.

Know which files you are going to commit
: Always check what you are about to commit by use of the

  `git status`

  command. This ensures that you know which files have been modified, which files have been removed and which files have been added, and provides a useful reminder to use the `git add` and `git rm` commands.

Know what has changed
: The use of the

  `git diff`

  command or

  `git add --patch`

  is a good check that you are not just committing an accidental keystroke or a debug statement. Moreover it is a good reminder of what has changed and needs to be mentioned in the commit message.

Provide clear and meaningful and relevant commit messages
: The commit message should explain what has changed and why, for details people can use `git diff`, however the commit message should be clear enough for people to have a good idea of what is going on. This is strongly coupled to the item about making only one change listed above - if two distinct things have been changed, they should be committed separately, with relevant commit messages. If the change

resulted from a `Problem Report` (PR) the PR number should be noted in the commit message.

Chapter B2

Use of the Issue Tracker

Bug tracking in Cactus is maintained by use of the bitbucket platform. This software provides audit trails of the status and all correspondence concerning any problem report (PR). Each problem is given a unique number and assigned a responsible person.

Correspondence — All correspondence with the author of the PR should be copied to the bitbucket ticket. This ensures that the correspondence is entered into the audit trail.

Responsiblity — When a PR comes in, it is assigned a responsible person. If another person wishes to tackle the problem they should check with the responsible person, and then assign themselves as the responsible person.

Initial auditing — The responsible person should review the PR and check that the user supplied fields are sensible. In particular the `Synopsis` should be an accurate reflection of the problem, and the `Priority` and `Severity` fields should be set to the correct levels. If the `Release` field is badly filled out, attempts should be made to determine the relase version used by the PR submitter. If it is a duplicate of a previous PR it should be marked as `Duplicate`.

Analysing the PR — Once the responsible person has had a chance to review the PR, they should either seek further information from the submitter and mark the PR state as `Feedback`, or they should seek to determine the cause of the problem and mark it as `Analysed`.

Closing a PR — Once a problem is fixed, the PR state should be changed to the current version number of Cactus. The `Fix` field should be filled out with what was done, and `git` hash for the change should be noted. Any miscellaneous comments about the problem should be noted in the `Release-Note` field.

Chapter B3

Release procedure

In the release cycle, Cactus is maintained in two git branches - the 'stable' `Cactus_XXX` and the development `master` branches. The stable version is the last release and no non-reviewed commits should ever be made to it - it is for people who do not want to worry about things breaking from day to day. The development version is the tree used for developing the next release.

Making a release consists of creating a new release branch from the master branch. The following procedure is used:

Notify committers of start of release procedure
: This ensures that no commits are made during the following procedure. If it is impossible, for some reason, to notify a person of the start of the procedure, that person's commit rights should be revoked during the procedure to prevent accidents.

Check the code on all supported architectures
: The code should be checked out (in a fresh place), compiled and the test-suites run on all suppported architectures. Problems found should be fixed or noted in the release notes. This is an iterative procedure, as any commits made to fix problems need to be checked on all other architectures.

Check example parameter files
: The example parameter files in thorn *par* directories should be run and updated for any additional or changed thorns or parameters.

Update ReleaseNotes
: The release notes should be added to the `doc/ReleaseNotes` file.

Tag the code
: Tag the code with the latest release tag and update the `LATEST` and `STABLE` tags. The easiest way to do this is from a clean checkout.

```
git tag Cactus_4_0_Beta_X_v0
```

Notify people
: The relase notes should be sent out to the cactus mailing lists and any other relevant places such as linux-announce and Freshmeat.

Update web page
: The release should be noted in the news section of the web page. Most information such as generating documentation takes place automatically for the web pages, the only thing which needs to be done manually is to checkout any new arrangements in the Stable Release in the relevent directories in the `CheckOut` directory as `cactus_web`.

Close PRs Any problem reports which were closed in the beta relase should be audited for correct entries in the `Fix` field and then their state should be marked as `closed`.

Part C

3

Revision

Chapter C1

Introduction

The make system has several design criteria:

- Must be able to build the code on all supported platforms.
- Must allow object files and executables for different architectures to co-exist without conflicts.
- Must allow object files and exectutables for different compiler options to co-exist without conflict.
- Must allow object files and executables for different thornsets to co-exist without conflicts.
- Thorn-writers must be able to easily add and remove files from their thorns without having to edit files provided by the flesh or other thorns.
- Thorn-writers must be able to control the compilation options and dependencies of their own thorns.

The first criterion is achieved by standarising to the use of the freely available GNU make programme, which is available on all platforms, and the use of *Autoconf* to detect the specific features of the particular machine the user is compiling on.

The next three criteria are achieved by the introduction of *configurations* which contain all the information and object files associated with a particular combination of machine, compilation options and thorns.

The final criteria are achieved by allowing the thorn-writer to specify their files and options in configuration files readable by the GNU make program, or by specifying their own make file.

C1.1 Note on philosophy of the make system

Make options can be divided into two classes.

- Configuration-time options Things which have an effect on the resulting executable. E.g. optimisation or debugging options.
- Make-time options Things which don't effect the final executable. E.g. warning-flags, flags to make in parallel.

Whenever an option is added to the make system care should be taken to preserve this distinction. It should be possible to go to the *config-data* directory of a configuration and examine the files there to determine how an executable was built. It should not be necessary to know the command-line used to build it.

Chapter C2

Make files

C2.1 Makefile

This is the master makefile.

In normal operation it calls *make.configuration* with the -j TJOBS flag to build TJOBS thorns in parallel.

C2.2 lib/make/make.configuration

This is the makefile which actually builds a configuration.

All built objects for a configuration go into a subdirectory called *build* of the configuration.

For each thorn listed in the *make.thornlist* file generated by the *CST* it runs *make.thornlib* or a file called *makefile* in the thorn's own source directory to generate a library for that thorn. Before running the sub-makefile it changes directory to a subdirectory of the *build* directory with the same name as the thorn and sets

- TOP The CCTK top-level directory.
- SRCDIR The thorn's source directory.
- CONFIG The *config* subdirectory of the configuration.
- NAME The name of the library which should be created for the thorn (including directory info).
- THORN The name of the thorn.

The sub-makefile is passed the -j FJOBS flag to build FJOBS files in parallel.

If *make.thornlist* doesn't exist, it runs the *CST* to generate it from the *ThornList* file.

If *ThornList* doesn't exist, it generates a list of thorns in the *arrangements* and then gives the user the option to edit the list.

C2.3 lib/make/make.thornlib

This makefile is responsible for producing a library from the contents of a thorn's source directory.

In each source directory of a thorn the author may put two files.

- make.code.defn This should contain a line

  `SRCS =`

  which lists the names of source files *in that directory*.
- make.code.deps This is an optional file which gives standard make dependency rules for the files in that directory.

In addition the thorn's top-level *make.code.defn* file can contain a line

`SUBDIRS =`

which lists *all* subdirectories of the thorn's *src* directory which contain files to be compiled.

To process the subdirectories the makefile runs the sub-makefile *make.subdir* in each subdirectory.

Once that is done it compiles files in the *src* directory and then all the object files into a library which is named by the *NAME* make-variable.

All object files are compiled by the rules given in *make.config.rules*.

Since the make language doesn't contain looping constructions it is a bit tricky to construct the full list of object files. To do this the makefile uses the GNU make *foreach* function to include all the subdirectory *make.code.defn* files, along with two auxiliary files *make.pre* and *make.post* which are included respectively before and after each *make.code.defn* file. These auxiliary files allow the *SRCS* variables set in the *make.code.defn* files to be concatanated onto one make variable *CCTK_SRCS*.

Extensive use is made of the two different flavours of variable (simply-expanded and recursively-expanded) available within GNU make.

The GNU '-include' construct is used to suppress warnings when an optional file is not available.

C2.4 lib/make/make.subdir

This builds all the object files for a specific subdirectory according to the list of files provided by the *make.code.defn* file in that subdirectory. Extra dependencies can be provided for these files by the presence of the optional file *make.code.deps* in the directory.

C2.5 lib/make/make.pre and lib/make/make.post

These are auxiliary files used to construct the full list of source files for a particular thorn.

make.pre resets the *SRCS* variable to an empty value. *make.post* adds the name of the subdirectory onto all filenames in the *SRCS* variable and adds the resulting list to the *CCTK_SRCS* make variable.

Chapter C3

Autoconf stuff

GNU autoconf is a program designed to detect the features available on a particular platform. It can be used to determine the compilers available on a platform, what the CPU and operating system are, what flags the compilers take, and as many other things as m4 macros can be written to cover.

Autoconf is configured by a file *configure.in* which autoconf turns into a file called *configure* (which should never be editted by hand). The cactus configuration includes the resulting *configure* file and this should not need to be regenerated by other than flesh-maintainers.

When the *configure* script is run it takes the files *config.h.in*, *make.config.defn.in*, *make.config.rules.in*, and *make.config.deps.in* and generates new files in the configuration's *config-data* subdirectory with the same names with the *.in* stripped off. The configure script replaces certain parts of these files with the values it has detected for this architecture.

In addition *configure* runs the *configure.pl* perl script to do things which can only be done easily by perl.

C3.1 configure.in

This and the macro-definition file *aclocal.m4* are the sources for the *configure* script. Autoconf should be run in this directory if either of these files is editted.

Once the script has determined the host architecture, it checks the *known-architecture* directory for any preferred compilers. By default autoconf macros will choose GNU CC if it is available, however for some architectures this may not be desirable.

It then proceeds to determine the available compilers and auxiliary programs if they haven't alrady been specified in an environment variable or in the *known-architecture* file for this architecture.

Once the set of programs to be used has been detected or chosen, the known-architecture files are again checked for specific features which would otherwise require the writing of complicated macros to detect. (Remember that the goal is that people don't need to write autoconf macros or run autoconf themselves.)

Once that is done it looks at each subdirectory of the *extras* directory for packages which have their own configuration process. If a subdirectory has an exectubale file called *setup.sh* this is called.

The rest of the script is concerned with detecting various header files, sizes of various types, and of setting defaults for things which haven't been set by the *known-architecture* file.

C3.2 config.h.in

This file is turned into *config.h* in the *config-data* directory in the configuration.

It contains C preprocessor macros which define various features or configuration options.

C3.3 make.config.defn.in

This file is turned into *make.config.defn* in the *config-data* directory in the configuration.

It contains make macros needed to define or build the particular configuration.

C3.4 make.config.rules.in

This file is turned into *make.config.rules* in the *config-data* directory in the configuration.

It contains the rules needed to create an object file from a source file. Note that currently this isn't modified by the configuration process, as everything is controlled from variables in the *make.config.defn* file. However this situation may change in the future if really necessary.

C3.5 make.config.deps.in

This file is turned into *make.config.deps* in the *config-data* directory in the configuration.

Currently this file is empty; it may gain content later if we need to use autoconf to generate dependency stuff.

C3.6 aclocal.m4

This contains m4 macros not distributed with autconf.

C3.7 CCTK_functions.sh

This contains Bourne-shell functions which can be used by the configure script, or by stuff in the *extras* or *known-architrectures* subdirectories.

C3.7.1 CCTK_Search

This can be used to search a set of directories for a specific file or files and then set a variable as a result.

Usage: CCTK_Search <variable> <subdirectories to search> <what to search for> [base directory]

It will search each of the listed subdirectories of the base directory for the desired file or directory, and, if it's found, set the variable to the name of the subdirectory.

C3.7.2 CCTK_CreateFile

Creates a file with specific contents.

Usage: CCTK_CreateFile <filename> <content>

Note that this can only put one line in the file to begin with. Additional lines can be added with *CCTK_WriteLine*.

C3.7.3 CCTK_WriteLine

Write one line to a file.

Usage: CCTK_WriteLine <file> <line>

C3.8 known-architectures

This contains files which tell autoconf about specific not-easily-detectable features about particular architectures. Each file in this directory is named with the name held by the host_os autoconf variable.

Each file is called twice by the configure script. Once to determine the 'preferred-compilers' for that architecture, and once for everything else.

The first time is straight after the operating system is determined, and the variable *CCTK_CONFIG_STAGE* is set to 'preferred-compilers'. It should only set the names of compilers, and not touch anything else.

The second time it is called is after the compilers and auxiliary programs have been detected or otherwise chosen. *CCTK_CONFIG_STAGE* is set to 'misc' in this case. This stage can be used to set compiler options based upon the chosen compilers. The scripts are allowed to write (append) to *cctk_archdefs.h* in this stage if it needs to add to the C preprocessor macros included by the code. *CCTK_WriteLine* can be used to write to the file.

C3.9 extras

This directory is used to hold configuration programs for optional extra packages.

If a subdirectory of this directory contains an executable file *setup.sh*, this file is run.

The two files *cctk_extradefs.h* and make.extra.defn can be appended to, to add c preprocessor macros or add/modify make variables respectively. *CCTK_WriteLine* can be used to do this.

Note that include directories should be added to *SYS_INC_DIRS* and not directly to *INC_DIRS*.

C3.10 config.sub and config.guess

These files are provided in the autoconf distribution. They are used to determine the host operating system, cpu, etc and put them into a canonical form.

The files distributed with Cactus are slightly modified to allow recognition of the Cray T3E, to work with the Portland compilers under Linux, and to not do something stupid with unrecognised HP machines.

Chapter C4

Perl scripts

Various perl scripts are used in the make system.

C4.1 setup_configuration.pl

This is called by the top level makefile to create a new configuration or to modify an old one. It parses an options file setting environment variables as specified in that file, and then runs the autoconf-generated configure script.

C4.2 configure.pl

This file is called from the configure script to determine the way Fortran names are represented for the specified Fortran compiler. It works out the names for subroutines/functions, and for common blocks, and writes a perl script which can be used to convert a name to the appropriate form so C and Fortran can be linked together.

C4.3 new_thorn.pl

This generates the skeleton for a new thorn.

Part D

4

Chapter D1

Introduction

The CST is really the glue which holds the code together. It takes the specifications which users have provided in their *.ccl* files and generates C header and source files which are used to tell the flesh about the thorns.

The processing is done in three stages. In the first stage the *.ccl* files from each thorn in the *ThornList* are parsed and the data from them is stored internally in databases. In the second stage the data is cross-indexed for consistency. Finally the files are written out into the *bindings* directory in the configuration directory.

Chapter D2

The Databases

Chapter D3

The Generated Files

Chapter D4

The Parsing Routines

Chapter D5

The Output Routines

Chapter D6

Miscellaneous Routines

Part E

5

Revision

Chapter E1

Introduction

Part F

6

Revision

Chapter F1

Introduction

The Comm subdirectory contains the routines which deal with communication issues.

Part G

7

Chapter G1

Introduction

The IO subdirectory contains the routines which deal with IO methods.

Part H

8

Revision

Chapter H1

Introduction

The utils subdirectory contains miscelleanous utilities which are in principle independent of the rest of Cactus.

Part I

9

Revision

Chapter I1

Introduction

The schedule system is used to determine the order of executaion of user-supplied subroutines as scheduled by thorns in their *schedule.ccl*.

Chapter I2

Design and algorithms

Chapter I3

Implementation

Chapter I4

Summary of Interfaces

Part J

10

Chapter J1

XEmacs customisation

Here's the relevent section from my .emacs file for the coding guidelines

```
(require 'grdoc)

; C-mode customisation

(defun my-c-mode-common-hook ()
  ;; my customizations for all of c-mode, c++-mode, objc-mode, java-mode
  (c-set-offset 'substatement-open 0)
  (c-set-offset 'case-label '+)
  ;; other customizations can go here
  (turn-on-grdoc-mode)
  (font-lock-mode)
  (setq indent-tabs-mode nil)
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```