Cactus 4.14

Users' Guide

commit cd7e0d57217bec3023091c951729faf982cf5f54

Documentation compiled on: May 16, 2023

Contents

Preface

This document contains a quick-start guide to installing and running a Cactus application. In subsequent chapters, it provides more detailed information on advanced user's topics, as well as an introduction to thorn writing. Please report omissions, errors, or suggestions to any of our contact addresses below.

Overview of documentation

Part A: Introduction to Cactus.
A guide through the process of obtaining and installing Cactus and running a simple example application with it.

Part B: Additional Notes.
A more in-depth description of required hardware and software, along with configuration, installation and running options. Describes how to check the installation with Cactus test suites.

Part C: Thorn Writing.
An introduction to thorn concepts and description of how to create, write and maintain application thorns. Explanation of use of the programming interface to take advantage of parallelism and modularity. This is followed by a more advanced discussion of user supplied infrastructure routines such as additional *output routines*, *drivers*, etc.

Part D: Appendices.
These contain a glossary, a description of the Cactus Configuration Language, the Utility routines and other odds and ends, such as how to use GNATS and TAGS.

Related topics are discussed in separate documents including:

Reference Manual Contains detailed descriptions of the functions provided by the Cactus flesh API, along with other reference material.

Typographical Conventions

`Typewriter`	Is currently used for everything you type, for program names, and code extracts.
`< ... >`	Indicates a compulsory argument.
`[ ... ]`	Indicates an optional argument.
`|`	Indicates an exclusive or.

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 email to `cactusmaint@cactuscode.org`.

Acknowledgements

Hearty thanks to all those who have helped with documentation for the Cactus Code. Special thanks to those who struggled with the earliest sparse versions of this guide and sent in mistakes and suggestions, in particular John Baker, Carsten Gundlach, Ginny Hudak-David, Sai Iyer, Paul Lamping, Nancy Tran and Ed Seidel.

Part A

Introduction

Chapter A1

Getting Started

A1.1 Obtaining Cactus

Cactus is distributed, extended, and maintained using the free git software (https://git-scm.com/) git allows many people to work on a large software project together without getting into a tangle. Since Cactus thorns are distributed from several repositories on the main git site, and from a growing number of user sites, we provide a `GetComponents` script on our website for checking out the flesh and thorns. The script is available at

https://github.com/gridaphobe/CRL/raw/master/GetComponents.

The script takes as an argument the name of a file containing a *ThornList*, that is a list of thorns with the syntax

```
<arrangement name>/<thorn name>
```

Optional directives in the ThornList indicate which repository to fetch thorns from. The ThornList is written in the *Component Retrieval Language*, documented at https://github.com/gridaphobe/CRL/wiki/Component-Retrieval-Language.

The same script can be used to checkout additional thorns, or to update existing ones.

The components that make up Cactus can also be checked out directly using git from https://bitbucket.org/cactuscode/.

Another script, `MakeThornList`, can be used to produce a minimal ThornList from a given Cactus par file. It needs a *master* ThornList to be copied into your `Cactus` directory.

See http://www.cactuscode.org/download/thorns/MakeThornList.

A1.1.1 Directory Structure

A fresh checkout creates a directory `Cactus` with the following subdirectories:

`repos`	created by GetComponents to hold the checked out repositories
`doc`	Cactus documentation
`lib`	contains libraries
`src`	contains the source code for Cactus
`arrangements`	contains the Cactus arrangements. The arrangements (the actual "physics") are not supplied by just checking out just Cactus. If the arrangements you want to use are standard Cactus arrangements, or reside on our git repository (`https://bitbucket.org/cactuscode/`), they can be checked out in similar way to the flesh.

When Cactus is first compiled, it creates a new directory `Cactus/configs`, which will contain all the source code, object files and libraries created during the build process.

Configurations are described in detail in Section A1.2.1.

A1.2 Compiling a Cactus application

Cactus can be built in different configurations from the same copy of the source files, and these different configurations coexist in the `Cactus/configs` directory. Here are several instances in which this can be useful:

1. Different configurations can be for *different architectures*. You can keep executables for multiple architectures based on a single copy of source code, shared on a common file system.
2. You can compare different *compiler options*, and *debug-modes*. You might want to compile different communication protocols (e.g. MPI or Globus), or leave them out all together.
3. You can have different configurations for *different thorn collections* compiled into your executable.

A1.2.1 Creating a Configuration

At its simplest, this is done by `gmake <config>`. This generates a configuration with the name *config*, doing its best to automatically determine the default compilers and compilation flags suitable for the current architecture.

There are a number of additional command-line arguments which may be supplied to override some parts of the procedure; they are listed in Section B2.1.

Once you have created a new configuration, the command

```
gmake <configuration name>
```

will build an executable, prompting you along the way for the thorns which should be included. There is a range of `gmake` targets and options which are detailed in Section B2.4.1.

A1.3 Running a Cactus application

Cactus executables always run from a parameter file (which may be provided as a command-line argument taken from standard input), which specifies which thorns to use and sets the values of each thorn's parameters (the parameters that are not set will take on default values, see D2.3).

There is no restriction on the name of the parameter file, although it is conventional to use the file extension `.par`. Optional command-line arguments can be used to customise runtime behaviour, and to provide information about the thorns used in the executable. The general syntax for running Cactus from a parameter file is then

`./cactus_<`*`config`*`> <`*`parameter file`*`> [`*`command-line options`*`]`

A parameter file is a text file whose lines are either comments or parameter statements. Comments are blank lines or lines that begin with '`#`'. A parameter statement consists of one or more parameter names, followed by an '`=`', followed by the value(s) for this (these) parameter(s). Note that all string parameters are case insensitive.

The first parameter statement in any parameter file should set `ActiveThorns`, which is a special parameter that tells the program which *thorns* are to be activated. Only parameters from active thorns can be set (and only those routines *scheduled* by active thorns are run). By default all thorns are inactive. For example, the first entry in a parameter file which is using just the two thorns `CactusPUGH/PUGH` and `CactusBase/CartGrid3D` should be

```
ActiveThorns = "PUGH CartGrid3D"
```

Parameter specifications following `ActiveThorns` usually are carried out by listing the name of the *thorn* which defined the parameter, two colons, and the name of the parameter — e.g. `wavetoyF77::amplitude` (see Section C1.4.2 for more information).

Notes:

- You can obtain lists of the parameters associated with each thorn using the command-line options `-o` and `-O` (Section B3.1).
- For examples of parameter files, look in the `par` directory which can be found in most thorns.
- The Cactus make system provides a mechanism for generating a *Thorn Guide* containing separate chapters for each thorn and arrangement in your configuration. Details about parameters, grid variables and scheduling are automatically included in from a thorn's CCL files into the Thorn Guide. To construct a Thorn Guide for the configuration <*`config`*> use

  `gmake <`*`config`*`>-ThornGuide`

  or to make a Thorn Guide for all the thorns in the `arrangements` directory

  `gmake <`*`config`*`>`.

Chapter A2

Getting and looking at output

A2.1 Screen output

As your Cactus executable runs, standard output and standard error are usually written to the screen. Standard output provides you with information about the run, and standard error reports warnings and errors from the flesh and thorns.

As the program runs, the normal output provides the following information:

Active thorns
: A report is made as each of the thorns in the `ActiveThorns` parameters from the parameter file (see Section B3.2) is attempted to be activated. This report shows whether the thorn activation was successful, and if successful gives the thorn's implementation. For example

```
Activating thorn idscalarwave...Success -> active implementation idscalarwave
```

Failed parameters
: If any of the parameters in the parameter file does not belong to any of the active thorns, or if the parameter value is not in the allowed range (see Section C1.4.1), an error is registered. For example, if the parameter is not recognised

```
Unknown parameter time::ddtfac
```

or if the parameter value is not in the allowed range

```
Unable to set keyword CartGrid3D::type - ByMouth not in any active range
```

Scheduling information
: The scheduled routines (see Section C1.5), are listed, in the order that they will be executed. For example

```
--------------------------------------------------------------------------------
  Startup routines
    Cactus: Register banner for Cactus
    CartGrid3D: Register GH Extension for GridSymmetry
```

```
    CartGrid3D: Register coordinates for the Cartesian grid
    IOASCII: Startup routine
    IOBasic: Startup routine
    IOUtil: IOUtil startup routine
    PUGH: Startup routine
    WaveToyC: Register banner

  Parameter checking routines
    CartGrid3D: Check coordinates for CartGrid3D
    IDScalarWave: Check parameters

  Initialisation
    CartGrid3D: Set up spatial 3D Cartesian coordinates on the GH
    PUGH: Report on PUGH set up
    Time: Set timestep based on speed one Courant condition
    WaveToyC: Schedule symmetries
    IDScalarWave: Initial data for 3D wave equation

  do loop over timesteps
    WaveToyC: Evolution of 3D wave equation
    t = t+dt
    if (analysis)
    endif
  enddo
--------------------------------------------------------------------------
```

Thorn banners

Usually a thorn registers a short piece of text as a *banner*. This banner of each thorn is displayed in the standard output when the thorn is initialised.

A2.2 File output

Output methods in Cactus are all provided by thorns. Any number of output methods can be used for each run. The behaviour of the output thorns in the standard arrangements are described in those thorns' documentation.

In general, output thorns decide what to output by parsing a string parameter containing the names of those grid variables, or groups of variables, for which output is required. The names should be fully qualified with the implementation and group or variable names.

There is usually a parameter for each method to denote how often, in evolution iterations, this output should be performed. There is also usually a parameter to define the directory in which the output should be placed, defaulting to the directory from which the executable is run.

See Chapter C2.7 for details on creating your own IO method.

Chapter A3

Checkpointing/Recovery

Checkpointing is defined as saving the current state of a run (parameter settings, contents of grid variables, and other relevant information) to a file. At a later time, this run can then be restarted from that state by recovering all the data from the checkpoint file.

Cactus checkpointing and recovery methods are provided by thorns. In general, these thorns decide how often to generate a checkpoint. They also register their recovery routines with the flesh; these recovery routines may then be called during initialisation of a subsequent run to perform the recovery of the state of the run. Such a recovery is requested by setting a parameter in the parameter file.

See Chapter C2.8 for details of how to create your own checkpointing and recovery methods.

Chapter A4

Reporting bugs

For tracking problem reports and bugs, we use the Bitbucket issue tracker at https://bitbucket.org/einsteintoolkit/tickets/ which allows easy submission and browsing of problem tickets.

A description of the issue categories we use is provided in Appendix D6.

Part B

Additional notes

Chapter B1

Installation

B1.1 Required Software

In general, Cactus *requires* the following set of software to function in single processor mode. Please refer to the architecture section B1.2 for architecture specific items.

Perl5.0	Perl is used extensively during the Cactus thorn configuration phase. Perl is available for nearly all operating systems known to man, and can be obtained at `http://www.perl.org`.
GNU make	The make process works with the GNU make utility (referred to as `gmake` henceforth). While other make utilities may also work, this is not guaranteed. Gmake can be obtained from your favorite GNU site, or from `http://www.gnu.org`.
C	C compiler. Cactus requires it to support the C99 standard. For example, the GNU compiler. This is available for most supported platforms. Platform specific compilers should also work.
CPP	C Preprocessor. For example, the GNU `cpp`. These are normally provided on most platforms, and many C compilers have an option to just run as a preprocessor.
C++	C++ compiler. Cactus requires it to support the C++11 standard. For example, the GNU compiler. This is available for most supported platforms. Platform specific compilers should also work.
GIT	*git* is not needed to run/compile Cactus, but you are strongly encouraged to install this software to take advantage of the update procedures. It can be downloaded from `https://git-scm.com/`.

To use Cactus, with the default driver[1] (`CactusPUGH/PUGH`) on multiple processors you also need to include the thorn `ExternalLibraries/MPI` in your thornlist, to include support for

MPI	The *Message Passing Interface*, which provides inter-processor communication. Supercomputing sites often supply a native MPI implementation that is very likely

[1] For help with unfamiliar terms, please consult the glossary, Appendix D1.

to be compatible with Cactus. Otherwise, there are various freely available ones available, e.g. the OpenMPI version of MPI is available for various architectures and operating systems at http://www.open-mpi.org/.

If you are using any thorns containing routines written in CUDA (Compute Unified Device Architecture), a parallel computing architecture developed by NVIDIA, you also need

CUCC — a CUDA compiler. For example, the NVIDIA C compiler. In many cases, you can compile your C and C++ code with a CUDA compiler without encountering any problems, but you are advised to use a CUDA compiler exclusively for CUDA code.

If you are using any thorns containing routines written in Fortran you also need

F90 — a Fortran compiler.

While not required for compiling or running Cactus, for thorn development it is useful to install

`ctags/etags` — These programs enable you browse through the calling structure of a program by help of a function call database. Navigating the flesh and arrangements becomes very easy. Emacs and `vi` both support this method. See D7 for a short guide to tags.

B1.2 Supported Architectures

Cactus runs on many machines, under a large number of operating systems, on a large number of CPU architectures. Here, we list the machines we have recently (in the past few years) compiled and verified Cactus on, including some architecture specific notes.

If your system is not on the list, the it is very likely that Cactus will work anyway. As a rule of thumb, if the GCC compiler is available, Cactus will run.

Operating systems:

Linux (this includes Blue Gene and Cray systems)

OS X

Cygwin

CPU architectures:

ARM

Blue Gene/P

IBM PowerA2 (aka Blue Gene/Q)

IBM Power (Power 5, 6, 7)

MIC (aka Xeon Phi)

x86 (aka IA32)

x86-64 (aka AMD64)

Systems that are of historic interest only:

SGI 32 or 64 bit running Irix.

Cray T3E

Compaq Alpha Compaq operating system and Linux. Single processor mode and MPI supported. The Alphas need to have the GNU C/C++ compilers installed.

IA64 running Linux.

Macintosh PowerPC (MacOS X and Linux PPC)

IBM SP2,SP3,SP4 32 or 64 bit running AIX.

Hitachi SR8000-F1

Sun Solaris

Fujitsu

NEC SX-5, SX-6

HP Exemplar (only partially supported)

B1.2.1 Note

Disk space may be a problem on supercomputers where home directories are small. A workaround is to first create a configs directory on scratch space, say `scratch/cactus_configs/` (where `scratch/` is your scratch directory), and then either

- set the environment variable `CACTUS_CONFIGS_DIR` to point to this directory

or

- soft link this directory (`ln -s scratch/cactus_configs Cactus/configs/`) to the Cactus directory, if your filesystem supports soft links.

Chapter B2

Compilation

B2.1 Configuration Options

There are four ways to pass options to the configuration process.

1 Pass options individually in shell environment variables:

```
export <option name>=<chosen value> # for bash
setenv <option name> <chosen value> # for (t)csh
gmake <configuration name>-config
```

2a Either: create a default configuration file `${HOME}/.cactus/config`.

All available configuration options may be set in a default options file `${HOME}/.cactus/config`, any option which are not set will take a default value. The file should contain lines of the form:

`<option> [=] ...`

The equals sign is optional. Spaces are allowed everywhere. Text starting wit a `'#'` character will be ignored as a comment.

2b Or: list your Cactus configuration files in an environment variable `CACTUS_CONFIG_FILES`:

`gmake <config name>-config CACTUS_CONFIG_FILES=<list of config files>`

Multiple configuration files, with their file names separated by a `':'` character, will be processed in order. Each file should be given by its full path. The options file has the same format as `${HOME}/.cactus/config`.

3 Add the options to a configuration file and use,

`gmake <config name>-config options=<filename>`

The options file has the same format as `${HOME}/.cactus/config`. (Note that these options are *added* to those from the `${HOME}/.cactus/config` file.)

4 Pass the options individually on the command line,

`gmake <config name>-config <option name>=<chosen value>, ...`

Not all configuration options can be set on the command line. Those that can be set are indicated in the table below.

The options are listed here in order of increasing precedence, e.g. options set on the command line will take priority over (potentially conflicting) options set in `${HOME}/.cactus/config` or other Cactus configuration files. Default options from `${HOME}/.cactus/config` will only be read if the environment variable `CACTUS_CONFIG_FILES` is not set.

Options read from configuration files can use expressions of the form `${VARIABLE}` which are replaced by the value of the environment variable `$VARIABLE` at the time `gmake <config name>-config` runs. The `$` sign can be escaped by duplicating it `$$`, and `$` followed by any character other than `$` or `{` is left as is (this may change in the future as more expansions are added). Since `gmake` also interprets `$$` in order to have a `$` show up to the shell it must be quadrupled `$$$$`. For example setting

```
CC = $${CXX} $(CXXFLAGS) $$(CPPFLAGS) -license ${HOME}/.license -I$$$${HDF5_HOME}/include
```

expands `${CXX}`, `$(CXXFLAGS)`, and `$(CPPFLAGS)` in the Makefile, `${HOME}` when `<config name>-config` runs, and `${HDF5_HOME}` when the compiler `${CXX}` runs.

It is important to note that these methods cannot be used to, for example, add options to the default values for `CFLAGS`. Setting any variable in the configuration file or the command line will overwrite completely the default values.

B2.1.1 Available Options

There is a plethora of available options.

- Cross compiling

  If you are compiling on an architecture other than the one you are producing an executable for, you will need to pass the

  `HOST_MACHINE=x-x-x`

  option, where *x-x-x* is the canonical name of the architecture you are compiling for, such as `sx6-nec-superux`; the format is *processor-vendor-OS*.

- Compiled thorns

  These specify the chosen set of thorns for compilation. If the thorn choice is not provided during configuration, a list containing all thorns in the `arrangements` directory is automatically created, and the user is prompted for any changes.

`THORNLIST`	Name of file containing a list of thorns with the syntax `<arrangement name>/<thorn name>`. Lines beginning with `#` or `!` are ignored.
`THORNLIST_DIR`	Location of directory containing `THORNLIST`. This defaults to the current working directory.

- Compiler and tool specification

`CC`	The C compiler.
`CXX`	The C++ compiler.
`CUCC`	The CUDA compiler.

F90	The Fortran compiler.
F77	Ignored
CPP	The preprocessor used to generate dependencies for and to preprocess C and C++ code.
FPP	The preprocessor used to generate dependencies for and to preprocess Fortran code.
LD	The linker.
AR	The archiver used for generating libraries.
RANLIB	The archive indexer to use.
MKDIR	The program to use to create a directory.
PERL	The name of the Perl executable.

- Output Directory

  By default, Cactus generates intermediate and object files underneath a directory named "configs" inside the Cactus directory. This location may be changed through the use of the CACTUS_CONFIGS_DIR environment variable. See the section on File Layout B2.3.

- Compilation and tool flags

  Flags which are passed to the compilers and the tools.

CFLAGS	Flags for the C compiler.
CUCCFLAGS	Flags for the CUDA compiler.
CXXFLAGS	Flags for the C++ compiler.
F90FLAGS	Flags for the Fortran compiler.
F77FLAGS	Ignored
CPPFLAGS	Flags for the preprocessor (used to generate compilation dependencies for and preprocess C and C++ code).
FPPFLAGS	Flags for the preprocessor (used to generate compilation dependencies for and preprocess Fortran code).
MKDIRFLAGS	Flags for MKDIR, so that no error is given if the directory exists.
LDFLAGS	Flags for the linker. *Warning:* This variable is ignored while the compilers and linkers are autodetected. This can lead to strange errors while configuring. You can pass the linker flags in the variable LD instead.
BEGIN_WHOLE_ARCHIVE_FLAGS, END_WHOLE_ARCHIVE_FLAGS	Optional set of flags for the linker that change the behaviour how archives are handled. These flags are used just before and just after listing the flesh and all thorn libraries. This mechanism can be used to force linking in all object files from the flesh and all thorns, which can help detect duplicate definitions. Otherwise, duplicate routines may go undetected.
ARFLAGS	Flags for the archiver.
C_LINE_DIRECTIVES	Whether error messages and debug information in the compiled C and C++ files should point to the original source file or to an internal file created by Cactus. The only options available are yes and no, the default is yes. Set this to no if your compiler reports error messages about unrecognised # directives.

`F_LINE_DIRECTIVES` Whether error messages and debug information in the compiled Fortran files should point to the original source file or to an internal file created by Cactus. The only options available are `yes` and `no`, the default is `yes`. Set this to `no` if your compiler reports error messages about unrecognised `#` directives.

`CROSS_COMPILE` Enables cross compilation. Available options are `yes` and `no`, the default is `no`. To create a cross-compiled configuration one must explicitly set this option to `yes`. You will also have to set `ENDIAN`.

`ENDIAN` Explicitly sets the endianess of multi-byte types. Normally this is detected automatically but needs to be specified for cross-compilations. Available options are `little` and `big`.

`DISABLE_INT16` Disable support for the data type `CCTK_INT16`. The only options available are `yes` and `no`, the default is `no`. Cactus autodetects this data type only for C. If the C compiler supports it, but the Fortran compiler does not, it may be necessary to disable `CCTK_INT16` altogether, since Cactus assumes that data types are fully supported if they exist.

`DISABLE_REAL16` Disable support for the data type `CCTK_REAL16`. The only options available are `yes` and `no`, the default is `no`. Cactus autodetects this data type only for C. If the C compiler supports it, but the Fortran compiler does not, it may be necessary to disable `CCTK_REAL16` altogether, since Cactus assumes that data types are fully supported if they exist.

`DEBUG` Specifies what type of debug mode should be used, the default is no debugging. Current options are `yes`, `no`, or `memory`. The option `yes` switches on all debugging features, whereas `memory` just employs memory tracing (Section C1.9.3).

`C_DEBUG_FLAGS` Debug flags for the C compiler, their use depends on the type of debugging being used.

`CUCC_DEBUG_FLAGS` Debug flags for the CUDA compiler, their use depends on the type of debugging being used.

`CXX_DEBUG_FLAGS` Debug flags for the C++ compiler, their use depends on the type of debugging being used.

`F90_DEBUG_FLAGS` Debug flags for the Fortran 90 compiler, their use depends on the type of debugging being used.

`F77_DEBUG_FLAGS` Ignored.

`OPTIMISE, OPTIMIZE`
Specifies what type of optimisation should be used. The only options currently available are `yes` and `no`. The default is to use optimisation.
Note that the British spelling `OPTIMISE` will be checked first and, if set, will override any setting of the American-spelled `OPTIMIZE`.

`C_OPTIMISE_FLAGS` Optimisation flags for the C compiler, their use depends on the type of optimisation being used.

`CUCC_OPTIMISE_FLAGS`
Optimisation flags for the C compiler, their use depends on the type of optimisation being used.

`CXX_OPTIMISE_FLAGS`
Optimisation flags for the C++ compiler, their use depends on the type of optimisation being used.

`F90_OPTIMISE_FLAGS`
Optimisation flags for the Fortran 90 compiler, their use depends on the type of optimisation being used.

`F77_OPTIMISE_FLAGS`
: Ignored.

`C_NO_OPTIMISE_FLAGS`
: Optimisation flags used to indicate that no optimisation should be performed. These are invoked when OPTIMISE=no is used.

`CUCC_NO_OPTIMISE_FLAGS`
: Optimisation flags used to indicate that no optimisation should be performed. These are invoked when OPTIMISE=no is used.

`CXX_NO_OPTIMISE_FLAGS`
: Optimisation flags used to indicate that no optimisation should be performed. These are invoked when OPTIMISE=no is used.

`F90_NO_OPTIMISE_FLAGS`
: Optimisation flags used to indicate that no optimisation should be performed. These are invoked when OPTIMISE=no is used.

`F77_NO_OPTIMISE_FLAGS`
: Ignored.

`PROFILE`
: Specifies what type of profiling should be used. The only options currently available are `yes` and `no`. The default is to use no profiling.

`C_PROFILE_FLAGS`
: Profile flags for the C compiler, their use depends on the type of profiling being used.

`CUCC_PROFILE_FLAGS`
: Profile flags for the CUDA compiler, their use depends on the type of profiling being used.

`CXX_PROFILE_FLAGS`
: Profile flags for the C++ compiler, their use depends on the type of profiling being used.

`F90_PROFILE_FLAGS`
: Profile flags for the Fortran 90 compiler, their use depends on the type of profiling being used.

`F77_PROFILE_FLAGS`
: Ignored.

`WARN`
: Specifies what type of build warnings should be used. The only options currently available are `yes` and `no`. The default is to produce no warnings.

`C_WARN_FLAGS`
: Warning flags for the C compiler, their use depends on the type of warnings used during compilation (Section B2.4.4).

`CUCC_WARN_FLAGS`
: Warning flags for the CUCC compiler, their use depends on the type of warnings used during compilation (Section B2.4.4).

`CXX_WARN_FLAGS`
: Warning flags for the C++ compiler, their use depends on the type of warnings used during compilation (Section B2.4.4).

`F90_WARN_FLAGS`
: Warning flags for the Fortran 90 compiler, their use depends on the type of warnings used during compilation (Section B2.4.4).

`F77_WARN_FLAGS`
: Ignored.

- Architecture-specific flags

  `IRIX_BITS=32|64`
  : For Irix SGI systems: whether to build a 32- or 64-bit configuration.

  `AIX_BITS=32|64`
  : For IBM SP systems: whether to build a 32- or 64-bit configuration.

- Library flags

  Used to specify auxiliary libraries and directories to find them in.

`LIBS`	Additional libraries. This variable can also contain linker options, e.g. to switch between static and dynamic linking. (Cactus adds a `-l` prefix to library names, but does not modify linker options.) *Warning:* This variable is ignored while the compilers and linkers are autodetected. This can lead to strange errors while configuring. You can pass the additional libraries in the variable `LD` instead.
`LIBDIRS`	Any other library directories. This variable can also contain linker options. (Cactus adds an `-L` prefix to library directories, but does not modify linker options.)

- Extra include directories

`SYS_INC_DIRS`	Used to specify any additional directories for system include files.

- Precision options

  Used to specify the precision of the default real and integer data types, by the number of bytes the data takes up. Note that not all values will be valid on all architectures.

`REAL_PRECISION`	Allowed values are `16, 8, 4`.
`INTEGER_PRECISION`	Allowed values are `8, 4, 2`.

- Executable name

`EXEDIR`	The directory in which to place the executable.
`EXE`	The name of the executable.

- Extra packages

  Compiling with extra packages is described fully in Section B2.2, which should be consulted for the full range of configuration options.

- Miscellaneous

`PROMPT`	Setting this to `no` turns off all prompts from the make system.
`VERBOSE`	Setting this to `yes` instructs `gmake` to print the commands that it is executing.
`SILENT`	Setting this to `no` is an depreciated way of using `VERBOSE = yes`.

B2.2 Compiling with Extra Packages

Extra packages are provided through the *ExternalLibraries* thorns each of which usually supports a set of variables *`THORN_DIR`*, *`THORN_LIBS`*, *`THORN_LIB_DIRS`*, and *`THORN_INC_DIRS`* to control where include and library files are located, where `THORN` is the name of the `ExternalLibrary`. The actual list of supported variables can be found in each thorn's `configuration.ccl` file.

While not all `ExternalLibraries` interpret all options in the same way, typically options have the following meaning and allowed values:

`THORN_DIR` can take the values:

`BUILD`	compile included `THORN` copy.

NO_BUILD	search for THORN in *THORN_DIR*.
CUSTOM	do not search for or build THORN, instead use information provided in *THORN_LIBS*, *THORN_LIB_DIRS*, and *THORN_INC_DIRS*.
NONE	use native THORN implementation provided by the compiler or operating system; no further options are used.
any valid path	search for THORN in the given directory.
missing	if *THORN_DIR* is missing THORN is searched for in a number of well known locations and built from the included copy if it is not found.

The location of include files, modules and Fortran module files is given via:

THORN_DIR	where to search for THORN.
THORN_LIBS	libraries.
THORN_LIB_DIRS	library directories.
THORN_INC_DIRS	include file directories.
THORN_INSTALL_DIR	where to install THORN if the built in copy is used.

B2.2.1 Note on possible ExternalLibraries' location stripping

Each thorn which is compiled as one external library will automatically use the library version contained in the `<library>/dist` folder. In particular, the tarball in `<library>/dist` is only used if either THORN_DIR is set to BUILD or is left empty and no precompiled copy of the library is found. If another location is specified via the THORN_DIR variable in the configuration file at compilation, then the `lib/sbin/strip-incdirs.sh` script will automatically strip away (for safety reasons) the locations:

```
/include
/usr/include
/usr/local/include
```

from THORN_INC_DIRS which default to THORN_DIR/include. Therefore, if there is any need for using one already installed version of one external library, the aforementioned location should be avoided (e.g. indicating /home as the THORN_DIR will work with no problems if the required library is installed there) or should be carefully checked, in order to avoid unwanted stripping. It is worth mentioning that the compiler normally automatically searches in /usr and /usr/local so stripping does not prevent it from being found (basically they are only found later in the search order of the compiler). The same stripping happens to THORN_LIB_DIRS in `lib/sbin/strip-libdirs.sh` with a larger list of directories:

```
/lib
/usr/lib
/usr/local/lib
```

/lib64

/usr/lib64

/usr/local/lib64

B2.3 File Layout

The configuration process sets up various subdirectories and files in the configuration directory (this is either a directory `configs` inside the main Cactus directory, or the directory pointed to by the `CACTUS_CONFIGS_DIR` environment variable). to contain the configuration specific files; these are placed in a directory with the name of the configuration.

`config-data` contains the files created by the configure script:

The most important ones are

`make.config.defn`	contains compilers and compilation flags for a configuration.
`make.extra.defn`	contains details about extra packages used in the configuration.
`cctk_Config.h`	The main configuration header file, containing architecture specific definitions.
`cctk_Archdefs.h`	An architecture specific header file containing things which cannot be automatically detected, and have thus been hand-coded for this architecture.

These are the first files which should be checked or modified to suit any peculiarities of this configuration.

In addition, the following files may be informative:

`fortran_name.pl`	A Perl script used to determine how the Fortran compiler names subroutines. This is used to make some C routines callable from Fortran, and Fortran routines callable from C.
`make.config.deps`	Initially empty. It can be edited to add extra architecture specific dependencies needed to generate the executable.
`make.config.rule`	The `make` rules for generating object files from source files.

Finally, `autoconf` generates the following files.

`config.log`	A log of the `autoconf` process.
`config.status`	A script which may be used to regenerate the configuration.
`config.cache`	An internal file used by `autoconf`.

`lib`	An empty directory which will contain the libraries created for each thorn.
`build`	An empty directory which will contain the object files generated for this configuration, and preprocessed source files.
`config-info`	A file containing information about the configuration (including the options used to configure the configuration).
`bindings`	A directory which contains all the files generated by the CST from the `.ccl` files.
`scratch`	A scratch directory which is used to accommodate Fortran 90 modules.

B2.4 Building and Administering a Configuration

Once you have created a new configuration, the command

`gmake <configuration name>`

will build an executable, prompting you along the way for the thorns which should be included. There is a range of `gmake` targets and options which are detailed in the following sections.

B2.4.1 gmake Targets for Building and Administering Configurations

A target for `gmake` can be naively thought of as an argument that tells it which of several things listed in the `Makefile` it is to do. The command `gmake help` lists all `gmake` targets:

`gmake <config>` builds a configuration. If the configuration doesn't exist, it will create it.

`gmake <config>-build BUILDLIST="<list of thorns>"`
compiles only the thorns listed.

`gmake <config>-clean`
removes all object and dependency files from a configuration.

`gmake <config>-cleandeps`
removes all dependency files from a configuration.

`gmake <config>-cleanobjs`
removes all object files from a configuration.

`gmake <config>-config`
creates a new configuration or reconfigures an existing one overwriting any previous configuration options.
The configuration options are stored in a file `configs/<config>/config-info`.

`gmake <config>-configinfo`
displays the options of the configuration (`cat configs/<config>/config-info`).

`gmake <config>-delete`
deletes a configuration (`rm -r configs/<config>`).

`gmake <config>-editthorns`
edits the `ThornList`.

`gmake <config>-examples`
copies all the example parameter files relevant for this configuration to the directory `examples` in the Cactus home directory. If a file of the same name is already there, it will not overwrite it.

`gmake <config>-realclean`
removes from a configuration all object and dependency files, as well as files generated from the CST (stands for *Cactus Specification Tool*, which is the set of Perl scripts which parse the thorn configuration files). Only the files generated by configure and the `ThornList` file remain.

`gmake <config>-rebuild`
rebuilds a configuration (reruns the CST).

`gmake <config>-reconfig`
: reconfigures an existing configuration using its previous configuration options from the file `configs/<config>/config-info`.

`gmake <config>-testsuite`
: runs the test programs associated with each thorn in the configuration. See section B2.6 for information about the test suite mechanism.

`gmake <config>-ThornGuide`
: builds documentation for the thorns in this configuration (see section B2.5, page B15, for other targets to build documentation for thorns).

`gmake <config>-thornlist`
: regenerates the `ThornList` for a configuration.

`gmake <config>-utils [UTILS=<list>]`
: builds all utility programs provided by the thorns of a configuration. Individual utilities can be selected by giving their names (i.e. name of the source file without extension) in the `UTILS` variable.

B2.4.2 Compiling in Thorns

Cactus will try to compile all thorns listed in `configs/<config>/ThornList`. The `ThornList` file is simply a list of the form `<arrangement>/<thorn>`. All text after a pound sign '`#`' or exclamation mark '`!`' on a line is treated as a comment and ignored. If you did not specify a ThornList already, the first time that you compile a configuration you will be shown a list of all the thorns in your arrangement directory, and asked if you with to edit them. You can regenerate this list at anytime by typing

```
gmake <config>-thornlist
```

,

or you can edit it using

```
gmake <config>-editthorns
```

.

Instead of using the editor to specify the thorns you want to have compiled, you can *edit* the `ThornList` outside the make process. It is located in `configs/<config>/ThornList`, where `<config>` refers to the name of your configuration. The directory `./configs` exists *after* the very first make phase for the first configuration.

B2.4.3 Notes and Caveats

- If during the build you see the error "`missing separator`", you are probably not using GNU make.
- *The EDITOR environment variable.* You may not be aware of this, but this thing very often exists and, may be set by default to something scary like `vi`. If you don't know how to use `vi`, or wish to use your favorite editor instead, reset this environment variable. (To exit `vi` type `<ESC> :q!`)

B2.4.4 `gmake` Options for building configurations

An *option* for `gmake` can be thought of as an argument which tells it how it should make a *target*. Note that the final result is always the same.

`gmake <target> PROMPT=no`
: turns off all prompts from the make system.

`gmake <target> VERBOSE=yes`
: prints the commands that gmake is executing.

`gmake <target> WARN=yes`
: shows compiler warnings during compilation.

`gmake <target> FJOBS=<number>`
: compiles in parallel, across files within each thorn.

`gmake <target> TJOBS=<number>`
: compiles in parallel, across thorns.

Note that with more modern versions of gmake, it is sufficient to pass the normal `-j <number>` flag to gmake to get parallel compilation.

B2.5 Other gmake Targets

`gmake help`	lists all make options.
`gmake configinfo`	prints configuration options for every configuration found in user's `configs` subdirectory.
`gmake default`	creates a new configuration with a default name.
`gmake distclean`	deletes your `configs` directory, and hence all your configurations.
`gmake downsize`	removes non-essential files as documents and test suites to allow for minimal installation size.
`gmake newthorn`	creates a new thorn, prompting for the necessary information and creating template files.
`gmake TAGS`	creates an Emacs style TAGS file. See section D7 for using tags within Cactus.
`gmake tags`	creates a `vi` style tags file. See section D7 for using tags within Cactus.

Targets to generate Cactus documentation:

`gmake <arrangement>-ArrangementDoc`
: builds the documentation for the arrangement.

`gmake ArrangementDoc`
: builds the documentation for all arrangements.

`gmake MaintGuide` runs LaTeX to produce a copy of the Maintainers' Guide.

`gmake ReferenceManual`
: runs LaTeX to produce a copy of the Reference Manual.

`gmake <thorn>-ThornDoc`
: builds the documentation for the thorn.

`gmake ThornDoc`
: builds the documentation for all thorns.

`gmake UsersGuide`
: runs LaTeX to produce a copy of the Users' Guide.

`gmake AllDoc`
: creates all of the above documentations.

B2.6 Testing

Some thorns come with a test suite, consisting of test parameter files and the output files generated by running these. To run the test suite for the all thorns you have compiled use

`gmake <configuration>-testsuite`

which interactively query for which tests to run and how many MPI processes to use.

You can use environment variables or variables set on the `gmake` command line to choose defaults for these for non-interactive use.

Setting *CCTK_TESTSUITE_RUN_TESTS* to a space separated list of *thorn* and *thorn/test* entries lets you choose which tests to run.

Setting *CCTK_TESTSUITE_PARALLEL_TESTS* to a non-zero value lets you run that many tests simulataneously. Depending on your MPI stack used this may require disabling automatic core binding avoid binding rank 0 of multiple tests to the same compute core.

Setting *TESTS_DIR* lets you choose the directory where tests are run in, by default this is the `TEST` directory in the Cactus main directory.

Setting *CCTK_TESTSUITE_RUN_PROCESSORS* chooses the number MPI processes (ranks) to use. It defaults to 2.

Setting *CCTK_TESTSUITE_RUN_COMMAND* sets the command to execute to start a single test. Cactus will replace the strings *$nprocs*, *$exe*, and *$parfile* by the requested number of MPI processes, the Cactus executable of the configuration being tested and the test parameter file. It defaults to `mpirun -np $nprocs $exe $parfile` if MPI is used and `$exe $parfile` otherwise.

Setting *PROMPT* to `no` skips the interactive prompts and uses the (potentially customized) defaults.

These test suite serve the dual purpose of

Regression testing
: i.e. making sure that changes to the thorn or the flesh don't affect the output from a known parameter file.

Portability testing
: i.e. checking that the results are independent of the architecture—this is also of use when trying to get Cactus to work on a new architecture.

Chapter B3

Runtime options

This chapter covers all aspects for running your Cactus executable. These include: command-line options, parameter file syntax, understanding screen output, environment variables, and creating thorn documentation.

B3.1 Command-Line Options

Cactus uses the standard GNU style of long-named command-line options; many of these options also have traditional Unix single-letter short forms. The options follow the usual GNU rules:

- A long-named option `--foo` which takes an argument `bar` may be written as either `--foo bar` or as `--foo=bar`.
- A long-named option may be abbreviated, so long as the abbreviation is unambiguous.
- The preferred way of spelling a long-named option is `--foo`, but `-foo` also accepted, though this is deprecated.
- A short option, `-X`, which takes an argument `bar` may be written as either `-Xbar` or as `-X=bar`.
- An option which can be interpreted as either a short option, or as an abbreviated `-foo`-style long option, is interpreted as the former. In particular, `-re` is interpreted as an abbreviation for `-redirect`, rather than as `-r=e`.

The Cactus command-line options are specified in Table B3.1, and are as follows:

`-O` or `--describe-all-parameters`
: Prints a full list of all parameters from all thorns which were compiled, along with descriptions and allowed values. This can take an optional extra parameter `v` (i.e. `-Ov` to give verbose information about all parameters).

`-o<`*param*`>` or `--describe-parameter=<`*param*`>`
: Prints the description and allowed values for a given parameter—takes one argument.

Short Version	Long Version
-O[v]	--describe-all-parameters
-o<*param*>	--describe-parameter=<*param*>
-S	--print-schedule
-T	--list-thorns
-t<*arrangement/thorn*>	--test-thorn-compiled=<*arrangement/thorn*>
-h,-?	--help
-v	--version
-L<*level*>	--logging-level=<*level*>
-W<*level*>	--warning-level=<*level*>
-E<*level*>	--error-level=<*level*>
-r[o|e|oe|eo]	--redirect=[o|e|oe|eo]
-R[o|e|oe|eo]	--Redirect=[o|e|oe|eo]
	--logdir=<*directory*>
-b[no|line|full]	--buffering=[no|line|full]
	--parameter-level=<strict|normal|relaxed>
-i	--ignore-next

Table B3.1: This table shows all the Cactus command-line options.

`-S` or `--print-schedule`
: Print only the schedule tree.

`-T` or `--list-thorns`
: Prints a list of all the thorns which were compiled in.

`-t<`*arrangement or thorn*`>` or `--test-thorn-compiled=<`*arrangement or thorn*`>`
: Checks if a given thorn was compiled in—takes one argument.

`-h`, `-?` or `--help`
: Prints a help message.

`-v` or `--version`
: Prints version information of the code.

`-L<`*level*`>` or `--logging-level=<`*level*`>`
: Sets the logging level of the code. All warning messages are given a level—the lower the level the greater the severity. This parameter `-L` controls the level of messages to be seen, with all warnings of level $\leq$ `<`*level*`>` printed to standard output. The default is a logging level of 0, meaning that only level 0 messages should be printed to standard output.

`-W<`*level*`>` or `--warning-level=<`*level*`>`
: Similar to `-W`, but for standard error instead of standard output. All warnings of level $\leq$ `<`*level*`>` are printed to standard error. The default is a warning level of 1, meaning that level 0 and level 1 messages should be printed to standard error.

`-E<`*level*`>` or `--error-level=<`*level*`>`
: Similar to `-W`, but for fatal errors: Cactus treats all warnings with level $\leq$ `<`*level*`>` as fatal errors, and aborts the Cactus run immediately (after printing the warning message[1]). The default value is zero, i.e. only level 0 warnings will abort the Cactus run.

`-r[o|e|oe|eo]` or `--redirect=[o|e|oe|eo]`
: Redirects the standard output (‘`o`’) and/or standard error (‘`e`’) of each processor to a file. By default, the standard outputs from processors other than processor 0 are discarded.

[1] Cactus imposes the constraint, `-W` level $\geq$ `-E` level ≥ 0, so any fatal-error message will always be printed (first).

`-R[o|e|oe|eo]` or `--Redirect=[o|e|oe|eo]`
: Redirects the standard output ('`o`') and/or standard error ('`e`') of each processor to a file like `-r` does, however different from `-r` it also redirects standard output and/or standard error from processor 0.

`--logdir=<`*directory*`>`
: Sets the output directory for logfiles created by the `-r` option. If the directory doesn't exist yet, it will be created by Cactus.

`-b[no|line|full]` or `--buffering=[no|line|full]`
: Set the `stdout` buffering mode. Buffered I/O is a standard feature of C programmes. This delays writing the actual output; instead, the output is collected into an internal buffer, and is then written in large chunks. This improves performance considerably. Line buffering means that output is written whenever a newline character is encountered; full buffering means that output is written, say, once 1000 characters have accumulated. The default setting is line buffering for I/O that goes to a terminal, and full buffering for I/O that goes to a file. For debugging purposes, it is sometimes useful to reduce the amount of buffering. Error messages, i.e. the `stderr` stream, is always unbuffered (and hence usually slower than `stdout`).

`--parameter-level=<strict|normal|relaxed>`
: Sets the level of parameter checking to be used, one of `strict` (the default), `normal`, or `relaxed`. See Section B3.2 for details.

`-i` or `--ignore-next`
: Causes the next argument on the command line to be ignored.

A dash ("`-`") appended at the end of the command line like this:

`./cactus_<`*config*`> [`*command-line options*`] -`

lets the user specify parameter values from standard input rather than from a parameter file.

B3.2 Parameter File Syntax

A *parameter file* (or *par file*) is used to control the behaviour of a Cactus executable. It specifies initial values for parameters as defined in the various thorns' `param.ccl` files (see Chapter C1.4). The name of a parameter file is often given the suffix `.par`, but this is not mandatory.

A parameter file is a text file whose lines are either blank lines, or parameter statements. Comments may also be included and consist of a '`#`' and all following characters. A parameter statement is an expression of the form *Left-Hand-Side* = *Right-Hand-Side*. The *Left-Hand-Side* may be a fully qualified parameter name (i.e. a thorn or implementation name, two colons, and a name defined in a `param.ccl` file), a variable name (the '*$*' character followed by a C-identifier), or the special variable *ActiveThorns*. The *Right-Hand-Side* is a value.

Values can be any of the following (all of which are case insensitive):

- Booleans: Booleans are either *true* (i.e. 1, `true`, `on`, `"true"`, or `"on"`) or *false* (i.e. 0, `false`, `off`, `"false"`, or `"off"`).
- Integers: Integers can be positive or negative.

- Real numbers: Real numbers can be positive or negative and may be written with exponents (e.g. `1.0e-3`, or `-2.94d+10`).
- Keywords: For keyword parameters, only the values enumerated in the parameter file are allowed. Like booleans, keywords may be put inside quotes. If they contain characters other than those found in C-identifiers, they must be.
- Strings: Sequences of characters delimited by quotes. If a quote is preceded by a backslash, it is not counted as a closing quote. A sequence of two backslashes will be interpreted as a single backslash.
- Variables: Parameter values can also contain variables of the form `${VARIABLE}` or `$ENV{VARIABLE}`. In the first form, braces are required when `VARIABLE` is followed by a character which is not to be interpreted as part of its name and optional otherwise. The second form provides access to all environment variables defined in the environment of the executable. Table B3.2 provides a list of recognized variable strings.

variable string	expands to
`$ENV{envname}`	the value of the environment variable `$envname`
`$parfile`	the path to the variable file with `.par` removed at the end
`$pi`	the numerical value of π

Table B3.2: Variable expansions recognized by the variable file parser when expanding variables in parameter files

In addition to the above, it is possible to create and use other variables by assignment, e.g.

```
$my_factor = 0.5
CoordBase::dx = 3.0*$my_factor
CoordBase::dy = 3.0*$my_factor
```

- Expressions: Parameters statements of numeric or boolean type can use arithmetic expressions in place of explicit values. The usual arithmetic operations as well as C-like transcendental functions and relational operations are supported. Integer division is handled as in C. Logical comparisons and variables expect a boolean type. The exponentiation operator `**` is supported, but can only apply to two values. The expression `3**4**2` is not supported, but `(3**4)**2` or `3**(4**2)` is supported. Operator precedence follows the C language, but when in doubt use explicit parenthesis to force a desired order of evaluation. Table B3.3 lists the supported functions. Expressions can refer to parameters which are already set by using the fully qualified name `thorn::parameter` as described below.

  The `%` operator is applicable only to integers. In cases where this is clear, types will be converted, e.g. `3.0` will convert to an integer, but not `3.1`. If you wish to prevent the parser from evaluating an expression, you can put it in double quotes and make it a string.

- Array assignments:

  Arrays of parameters can be set by including an integer expression inside the square brackets following the name, e.g. `thorn::parameters[0]`. Optionally, an array of parameters may be set by means of a comma delimited list of values inside square brackets. E.g. the following two examples are equivalent.

  Example 1:

```
thorn::parameters[0] = 4.8
thorn::parameters[1] = 3.2
```

Example 2:

```
thorn::parameters = [4.8, 3.2]
```

Please see the file `par.peg` in the directory `Cactus/src/piraha/pegs` for the full grammar describing the par file.

Logical operators		`!`	logical not
`&&`	logical and	Mathematical functions	
`||`	logical or	`acos`	inverse cosine
Relational operators		`asin`	inverse sine
`==`	tests for equality	`atan`	inverse tangent
`!=`	tests for inequality	`ceil`	round up to nearest integer
`<`	tests for less than	`cos`	cosine
`>`	tests for greater than	`cosh`	hyperbolic cosine
`<=`	tests for less or equal	`exp`	exponentiation e^x
`>=`	tests for greater or equal	`abs`	absolute value $\|x\|$
Binary operators		`floor`	round down to nearest integer
`+`	addition	`log`	natural logarithm
`-`	subtraction	`bool,int,real`	convert to bool, int, or real
`/`	C-like division	`sin`	sine
`%`	remainder of division	`sinh`	hyperbolic sine
`*`	multiplication	`sqrt`	square root
`**`	exponentiation x^y	`cbrt`	cube root
Unary operators		`tan`	tangent
`-`	negate sign	`tanh`	hyperbolic tangent
`+`	no-op	`trunc`	integer part of x

Table B3.3: Supported functions inside of expressions, in increasing order of precedence.

Every parameter file should set `ActiveThorns`, which is a special parameter that tells the program which *thorns* are to be activated. One may set `ActiveThorns` on any line or lines of the par file. In the case where multiple specifications of ActiveThorns are supplied, the values will be concatenated.

Only parameters belonging to active thorns can be set (and only those routines *scheduled* by active thorns are run). By default, all thorns are inactive. For example, the first entry in a parameter file which is using just the two thorns `CactusPUGH/PUGH` and `CactusBase/CartGrid3D` should be

```
ActiveThorns = "PUGH CartGrid3D"
```

All parameters following the `ActiveThorns` parameter have names whose syntax depends on the scope (see Section C1.4.2) of the parameter:

`Restricted parameters`
: The name of the *implementation* which defined the parameter, followed by two colons, then the name of the parameter—e.g. `driver::global_nx`.

`Private parameters`
: The name of the *thorn* which defined the parameter, two colons, and the name of the parameter—e.g. `wavetoyF77::amplitude`.

This notation is not currently strictly enforced in the code. It is sufficient to specify the first part of the parameter name using either the implementation name, or the thorn name. However, we recommend that the above convention be followed.

The Cactus flesh performs checks for consistency and range of parameters. The severity of these checks is controlled by the command-line argument `--parameter-level`, which can take the following values

`relaxed` Cactus will issue a level 0 warning (that is, the default behaviour will be to terminate) if

- The specified parameter value is outside of the allowed range.

`normal` This provides the same warnings as the `relaxed` level, with the addition of a level 0 warning issued for

- An implementation and/or thorn `foo` is active, but the parameter `foo::bar` was not defined.
- The parameter `foo::bar` was successfully set for both an active implementation `foo` not implemented by a thorn `foo`, and to a thorn `foo`.

`strict` This is the default, and provides the same warnings as the `normal` level, with the addition of a level 0 warning issued for

- The parameter `foo::bar` is specified in the parameter file, but no implementation or thorn with the name `bar` is active.

Notes:

- You can obtain lists of the parameters associated with each thorn using the command-line options `-o` and `-O` (Section B3.1).
- String parameter values can be specified either as unquoted tokens (not containing any whitespace), or as quoted values. If a quoted string parameter value spans multiple lines, all whitespaces, including newline characters, are preserved.
- Some parameters are *steerable*, and can be changed during the execution of a Cactus program using parameter steering interfaces, for example, thorn `CactusConnect/HTTPD`, or using a parameter file when recovering from a checkpoint file.
- For examples of parameter files, look in the `par` directory contained in most thorns.

B3.3 Thorn Documentation

The Cactus make system provides a mechanism for generating a *Thorn Guide* containing separate chapters for each thorn and arrangement in your configuration. The documentation provided for an individual thorn, obviously depends on what the thorn authors added, but the Thorn Guide is a good place to first look for special instructions on how to run and interpret the output from a thorn. Details about parameters, grid variables and scheduling are automatically read from a thorn's CCL files and included in the Thorn Guide. To construct a Thorn Guide for the configuration <*config*> use

`gmake <`*config*`>-ThornGuide`

or to make a Thorn Guide for all the thorns in the `arrangements` directory

`gmake <config>`.

See Section C1.8.4 for a guide to adding documentation to your own thorns.

Chapter B4

Getting and Looking at Output

B4.1 Screen Output

As your Cactus executable runs, standard output and standard error are usually written to the screen. Standard output provides you with information about the run, and standard error reports warnings and errors from the flesh and thorns.

As the program runs, the normal output provides the following information:

Active thorns

A report is made as each of the thorns in the `ActiveThorns` parameters from the parameter file (see Section B3.2) is attempted to be activated. This report shows whether the thorn activation was successful, and if successful, gives the thorn's implementation. For example

```
Activating thorn idscalarwave...Success -> active implementation idscalarwave
```

Failed parameters

If any of the parameters in the parameter file does not belong to any of the active thorns, or if the parameter value is not in the allowed range (see Section C1.4.1), an error is registered. For example, if the parameter is not recognised,

```
Unknown parameter time::ddtfac
```

or if the parameter value is not in the allowed range,

```
Unable to set keyword CartGrid3D::type - ByMouth not in any active range
```

Scheduling information

The scheduled routines (see Section C1.5) are listed, in the order that they will be executed. For example,

```
--------------------------------------------------------------------------------
  Startup routines
    Cactus: Register banner for Cactus
    CartGrid3D: Register GH Extension for GridSymmetry
```

```
    CartGrid3D: Register coordinates for the Cartesian grid
    IOASCII: Startup routine
    IOBasic: Startup routine
    IOUtil: IOUtil startup routine
    PUGH: Startup routine
    WaveToyC: Register banner

  Parameter checking routines
    CartGrid3D: Check coordinates for CartGrid3D
    IDScalarWave: Check parameters

  Initialisation
    CartGrid3D: Set up spatial 3D Cartesian coordinates on the GH
    PUGH: Report on PUGH set up
    Time: Set timestep based on speed one Courant condition
    WaveToyC: Schedule symmetries
    IDScalarWave: Initial data for 3D wave equation

  do loop over timesteps
    WaveToyC: Evolution of 3D wave equation
    t = t+dt
    if (analysis)
    endif
  enddo
--------------------------------------------------------------------------------
```

Thorn banners

Usually a thorn registers a short piece of text as a *banner*. The banner of each thorn is displayed in the standard output when the thorn is initialised.

B4.2 Output

Output methods in Cactus are all provided by thorns. Any number of output methods can be used for each run. The behaviour of the output thorns in the standard arrangements are described in those thorns' documentation.

In general, output thorns decide what to output by parsing a string parameter containing the names of those grid variables, or groups of variables, for which output is required. The names should be fully qualified with the implementation and group or variable names.

There is usually a parameter for each method to denote how often, in evolution iterations, this output should be performed. There is also usually a parameter to define the directory in which the output should be placed, defaulting to the directory from which the executable is run.

See Chapter C2.7 for details on creating your own I/O method.

Part C

Thorn Writing

C2

Chapter C1

Application thorns

This chapter goes into the nitty-gritty of writing a thorn. It introduces key concepts for thorns, then goes on to give a brief outline of how to configure a thorn. There is then some detail about concepts introduced by the configuration step, followed by discussion of code which you must put into your files in order to use Cactus functionality, and details of utility functions you may use to gain extra functionality.

C1.1 Thorn Concepts

C1.1.1 Thorns

A thorn is the basic working module within Cactus. All user supplied code goes into thorns, which are, by and large, independent of each other. Thorns communicate with each other via calls to the flesh API, plus, more rarely, custom APIs of other thorns.

The connection from a thorn to the flesh, or to other thorns, is specified in configuration files which are parsed at compile time and used to generate glue code which encapsulates the external appearance of a thorn.

Thorn names must be (case independently) unique, must start with a letter, can only contain letters, numbers or underscores, and must contain 27 characters or less. In addition, a thorn cannot have the name `doc`, this is reserved for arrangement documentation. Arrangement names which start with a '`#`', or finish with '`~`' or '`.bak`' will be ignored.

C1.1.2 Arrangements

Thorns are grouped into *arrangements*. This is a logical grouping of thorns which is purely for organisational purposes. For example, you might wish to keep all your initial data thorns in one arrangement, and all your evolution thorns in another arrangement, or you may want to have separate arrangements for your developments, private and shared thorns.

The arrangements live in the `arrangements` directory of the main Cactus directory. Arrangement names must be (case independently) unique, must start with a letter, and can only contain letters, numbers or

underscores. Arrangement names which start with a '`#`', or finish with '`~`' or '`.bak`' will be ignored.

Inside an arrangement directory there are directories for each thorn belonging to the arrangement.

C1.1.3 Implementations

One of the key concepts for thorns is the concept of the *implementation*. Relationships among thorns are all based upon relationships among the implementations they provide. In principle, it should be possible to swap one thorn providing an implementation with another thorn providing that implementation, without affecting any other thorn.

An implementation defines a group of variables and parameters which are used to implement some functionality. For example, the thorn `CactusPUGH/PUGH` provides the implementation `driver`. This implementation is responsible for providing memory for grid variables and for communication. Another thorn can also implement `driver`, and both thorns can be compiled in *at the same time*. At runtime, the user can decide which thorn providing `driver` is used. No other thorn should be affected by this choice.

When a thorn decides it needs access to a variable or a parameter provided by another thorn, it defines a relationship between itself and the other thorn's *implementation*, not explicitly with the other *thorn*. This allows the transparent replacement, at compile or runtime, of one thorn with another thorn providing the same functionality as seen by the other thorns.

C1.2 Anatomy of a Thorn

C1.2.1 Thorns

A thorn consists of a subdirectory of an arrangement containing four administrative files:

`interface.ccl`	the Cactus interface, which defines the grid functions, variables, etc. See Appendix D2.2.
`param.ccl`	the parameters introduced by this thorn, and the parameters needed from other thorns. See Appendix D2.3.
`schedule.ccl`	scheduling information for routines called by the flesh. See Appendix D2.4.
`configuration.ccl`	configuration options for the thorn. See Appendix D2.5.

Thorns can also contain

- a subdirectory called `src`, which should hold source files and compilation instructions for the thorn
- a subdirectory `src/include` for include files
- a `README` containing a brief description of the thorn
- a `doc` directory for documentation
- a `par` directory for example parameter files
- a `test` subdirectory may also be added, to hold the thorn's test suite. See Section C1.8.5 for details.

C1.2.2 Creating a Thorn

To simplify the creation of a thorn, a `make` target `gmake newthorn` has been provided. When this is run:

1. You will be prompted for the name of the new thorn.
2. You will be prompted for the name of the arrangement in which you would like to include your thorn. Either enter a new arrangement name or pick one from the list of available arrangements that are shown.

C1.2.3 Configuring your Thorn

The interaction of a thorn with the flesh and other thorns is controlled by certain configuration files.

These are:

`interface.ccl`	This defines the *implementation* (Section C1.1.3) the thorn provides, and the variables the thorn needs, along with their visibility to other implementations.
`param.ccl`	This defines the parameters that are used to control the thorn, along with their visibility to other implementations.
`schedule.ccl`	This defines which functions are called from the thorn and when they are called. It also handles memory and communication assignment for grid variables.
`configuration.ccl`	This file is optional for a thorn. If it exists, it contains extra configuration options of this thorn.

General Syntax of CCL Files

Cactus Configuration Language (CCL) files are text files used to define configuration information for a thorn. Their formal syntax is described using Piraha, a parsing expression grammar engine that supports multiple languages (see https://github.com/stevenrbrandt/piraha-peg for a description of Piraha patterns and Grammar Files).

A Grammar File for each type of CCL file is provided in the src/piraha/pegs directory of the Cactus source tree. These may be consulted for the precise details of any given Cactus CCL file.

CCL files are (mostly) case independent, and may contain comments introduced by the hash '`#`' character, which indicates that the rest of the line is a comment. If the last non-blank character of a line in a CCL file is a backslash '\', the following line is treated as a continuation of the current line.

The `interface.ccl` File

The `interface.ccl` file is used to declare

- the implementation provided by the thorn

- the variables provided by the thorn
- the include files provided by the thorn
- the functions provided by the thorn (in development)

The implementation is declared by a single line at the top of the file

`implements: <name>`

Where `<name>` can be any combination of alphanumeric characters and underscores, and is case independent.

There are three different access levels available for variables

`Public`	Can be 'inherited' by other implementations (see below).
`Protected`	Can be shared with other implementations which declare themselves to be friends of this one (see below).
`Private`	Can only be seen by this thorn.

Corresponding to the first two access levels there are two relationship statements that can be used to get variables (actually groups of variables, see below) from other implementations.

`Inherits: <name>`	This gets all `Public` variables from implementation `<name>`, and all variables that `<name>` has in turn inherited. An implementation may inherit from any number of other implementations.
`Friend: <name>`	This gets all `Protected` variables from implementation `<name>`, but, unlike `inherits`, it is symmetric and also defines a transitive relation by pushing its own implementation's `Protected` variables onto implementation *name*. This keyword is used to define a group of implementations which all end up with the same `Protected` variables.

So, for example, an `interface.ccl` starting

```
implements: wavetoy
inherits:   grid
friend:     wave_extract
```

declares that the thorn provides an implementation called `wavetoy`, gets all the `public` variables declared by an implementation called `grid`, and shares all `protected` variables with `wave_extract` and its friends.

Cactus variables, described in Chapter C1.3, are placed in groups with homogeneous attributes, where the attributes describe properties such as the data type, group type, dimension, ghostsize, number of timelevels, and distribution.

For example, a group, called `realfields` of 5 real grid functions (`phi`, `a`, `b`, `c`, `d`), on a 3D grid, would be defined by

```
CCTK_REAL realfields type=GF TimeLevels=3 Dim=3
{
  phi
  a,b,c,d
} "Example grid functions"
```

or, for a group called `intfields` consisting of just one distributed 2D array of integers,

```
CCTK_INT intfields type=ARRAY size=xsize,ysize ghostsize=gxsize,gysize dim=2
{
 anarray
} "My 2D arrays"
```

where `xsize`, `ysize`, `gxsize`, `gysize` are all parameters defined in the thorn's `param.ccl`.

By default, all groups are `private`, to change this, an access specification of the form `public:` or `protected:` (or `private:` to change it back) may be placed on a line by itself. This changes the access level for any group defined in the file from that point on.

All variables seen by any one thorn must have distinct names.

The `param.ccl` File

Users control the operation of thorns via parameters given in a file at runtime. The `param.ccl` file is used to specify the parameters used to control an individual thorn, and to specify the values these parameters are allowed to take. When the code is run, it reads a parameter file and sets the parameters if they fall within the allowed values. If a parameter is not assigned in a parameter file, it is given its default value.

There are three access levels available for parameters:

`Global`	These parameters are seen by all thorns.
`Restricted`	These parameters may be used by other implementations if they so desire.
`Private`	These are only seen by this thorn.

A parameter specification consists of:

- The parameter type (each may have an optional `CCTK_` in front)

`REAL`	
`INT`	
`KEYWORD`	A distinct string with only a few known allowed values.
`STRING`	An arbitrary string, which must conform to a given regular expression.
`BOOLEAN`	A boolean type which can take values `1`, `t`, `true`, `yes` or `0`, `f`, `false`, `no`.

- The parameter name

- An optional size (in square brackets)–if this is present, the parameter is a "parameter array", i.e. it will actually be an array of parameters, each of which has the same properties, but a different value. Such arrays appear as normal arrays in C or Fortran, 0-based in C, and 1-based in Fortran. In the parameter file the value of each element is specified with square brackets and is 0-based. The size must be an integer.
- A description of the parameter
- An allowed value block–this consists of a brace-delimited block of lines[1] describing the allowed values of the parameter. Each range may have a description associated with it by placing a `::` on the line, and putting the description afterwards.
- The default value–this must be one of the allowed values.

For the numeric types `INT` and `REAL`, a range consists of a string of the form lower-bound:upper-bound:step, where a missing number or an asterisk '`*`' denotes anything (i.e. infinite bounds or an infinitesimal step).

For example,

```
REAL Coeff "Important coefficient"
{
0:3.14 :: "Range has to be from zero to Pi, default is zero"
} 0.0

#No need to define a range for BOOLEAN
BOOLEAN nice "Nice weather?"
{
}"yes"

# A example for a set of keywords and its default (which has to be
# defined in the body)
KEYWORD confused "Are we getting confused?"
{
  "yes"    :: "absolutely positively"
  "perhaps" :: "we are not sure"
  "never"   :: "never"
} "never"

REAL Length[2] "Length in each direction"
{
0:* :: "Range has to be from zero to infinity, default is one"
} 1.0
```

defines a `REAL` parameter, a `BOOLEAN` parameter, a `KEYWORD`, and an array of `REAL` parameters.

By default, all parameters are `private`; to change this, an access specification of the form `global:` or `restricted:` (or `private:` to change it back) may be placed on a line by itself. This changes the access level for any parameter defined in the file from that point on.

[1] The beginning brace ({) must sit on a line by itself; the ending brace (}) must be preceded by a carriage return.

To access `restricted` parameters from another implementation, a line containing `shares: <name>` declares that all parameters mentioned in the file, from now until the next access specification, originate in implementation `<name>`. (Note that only one implementation can be specified on each `shares:` line.) Each of these parameters must be qualified by the initial token `USES` or `EXTENDS`, where

`USES`	indicates that the parameters range remains unchanged.
`EXTENDS`	indicates that the parameters range is going to be extended.

In contrast to parameter declarations in other access blocks, the default value must be omitted—it is impossible to set the default value of any parameter not originating in this thorn. For example, the following block adds possible values to the keyword `initial_data` originally defined in the implementation `einstein`, and uses the `REAL` parameter `speed`.

```
shares:einstein

EXTENDS KEYWORD initial_data
{
  "bl_bh"         :: "Brill Lindquist black holes"
  "misner_bh"     :: "Misner black holes"
  "schwarzschild" :: "One Schwarzschild black hole"
}

USES CCTK_REAL speed
```

Note that you must compile at least one thorn which implements `einstein`.

The `schedule.ccl` File

By default, no routine of a thorn will be run. The `schedule.ccl` file defines those that should be run, and when and under which conditions they should be run.

The specification of routine scheduling is via a schedule block which consists of lines of the form

```
schedule <name> at <time bin> [other options]
{
  LANG:     <FORTRAN|C>
  OPTIONS:     [list of options]
  TAGS:        [list of keyword=value definitions]
  STORAGE:     [group list with timelevels]
  READS:       [variable or group list with region specification]
  WRITES:      [variable or group list with region specification]
  TRIGGERS:    [group list]
  SYNC:        [group list]
} "A description"
```

where `<name>` is the name of the routine, and `<time bin>` is the name of a schedule bin (the `CCTK_` prefix is optional). A list of the most useful schedule bins for application thorns is given here, a complete and more descriptive list is provided in Appendix D4:

`CCTK_STARTUP`	For routines, run before the grid hierarchy is set up, for example, function registration.
`CCTK_PARAMCHECK`	For routines that check parameter combinations, routines registered here only have access to the grid size and the parameters.
`CCTK_BASEGRID`	Responsible for setting up coordinates, etc.
`CCTK_INITIAL`	For generating initial data.
`CCTK_POSTINITIAL`	Tasks which must be applied after initial data is created.
`CCTK_PRESTEP`	Stuff done before the evolution step.
`CCTK_EVOL`	The evolution step.
`CCTK_POSTSTEP`	Stuff done after the evolution step.
`CCTK_ANALYSIS`	For analysing data.

The *`other options`* allow finer-grained control of the scheduling. It is possible to state that the routine must run `BEFORE` or `AFTER` another routine or set of routines. It is also possible to schedule the routine under an alias name by using `AS <alias_name>`.

`LANG`	The `LANG` keyword specifies the linkage of the scheduled routine which determines how to call it from the scheduler. C and Fortran linkage are possible here. C++ routines should be defined as `extern "C"` and registered as `LANG: C`.
`OPTIONS`	Schedule options are used for mesh refinement and multi-block simulations, and they determine "where" a routine executes. Often used schedule options are `local` (also the default, may be omitted), `level`, or `global`. Routines scheduled in *local mode* can access individual grid points, routines scheduled in *level mode* are used e.g. to select boundary conditions, and routines schedule in *global mode* are e.g. used to calculate reductions (norms).
`TAGS`	Schedule tags, e.g. `Device=1` to specify that a routine executes on an OpenCL or CUDA device instead of on the host.
`STORAGE`	The `STORAGE` keyword specifies any groups for which memory should be allocated for the duration of the routine. The storage status reverts to its previous status after the routine returns. The format of the `STORAGE` statement includes specifying the number of timelevels of each group for which storage should be activated. `STORAGE: <group1>[timelevels1], <group2>[timelevels2]` This number can range from one to the maximum number of timelevels for the group, as specified in the group definition in its `interface.ccl` file. If this maximum number is one, the timelevel specification can be omitted from the `STORAGE` statement. Alternatively *`timelevels`* can be the name of a parameter accessible to the thorn. The parameter name is the same as used in C routines of the thorn, fully qualified parameter names of the form *`thorn::parameter`* are not allowed. In this case 0 (zero) *`timelevels`* can be requested, which is equivalent to the `STORAGE` statement being absent.

READS/WRITES

READS/WRITES are used to declare which grid variables are read/written by the routine. This information is used e.g. to determine which variables need to be synchronized, copied between host and device for OpenCL or CUDA kernel, or poisoned as part of error checking.

READS/WRITE directives can be applied to either a variable or a group of variables and may also contain a region specification. The region specification can be EVERYWHERE, INTERIOR, or BOUNDARY. If a function needs to read a given grid function everywhere and only the interior is valid (i.e. has been written to), then a synchronization is required and will happen automatically if enabled with the Cactus parameter `presync_mode`.

When grid functions are updated in this way, not only ghost zones, but boundary zones will be updated by the drier. Use either the function `Driver_SelectGroupForBC` or `Driver_SelectVarForBC` to tell the driver how it should update a group or variable. The arguments to both functions are the same as those found in the Boundary thorn, but with the prefix `Boundary_` instead of `Driver_`:

```
Driver_SelectGroupForBC(CCTK_ARGUMENTS,
                        CCTK_INT faces,
                        CCTK_INT width,
                        CCTK_INT table_handle,
                        CCTK_STRING group_name,
                        CCTK_STRING bc_name)

Driver_SelectVarForBC(CCTK_ARGUMENTS,
                      CCTK_INT faces,
                      CCTK_INT width,
                      CCTK_INT table_handle,
                      CCTK_STRING var_name,
                      CCTK_STRING bc_name)
```

Note that the above two functions, unlike their similarly named components in the Boundary thorn, only need to be called once. The information they provide will then be used each time the driver synchronizes the named grid function or group.

It is possible, however, that the thorn you are creating does not know the correct READS/WRITES information at compile time. I/O thorns, for example, do not know which grid functions they will access until run time. Likewise, the Method of Lines thorn does not know which grid functions it is operating on until run time.

For situations like these, we have the following functions which can be used to check READS/WRITES data and perform synchronization at run time: `Driver_RequireValidData`, `Driver_NotifyDataModified`, `Driver_GetValidRegion`, and `Driver_SetValidRegion`. For details on these functions, please consult the Reference Manual.

TRIGGERS

`TRIGGERS` is used when the routine is registered at `ANALYSIS`. This is a special time bin; a routine registered here will only be called if one of the variables from a group in `TRIGGERS` is due for output. (A routine without `TRIGGERS` declaration will always be called.)

SYNC

The keyword `SYNC` specifies groups of variables which should be synchronised (that is, their ghostzones should be exchanged between processors) on exit from the routine. Specifying synchronisation of grid variables in `schedule.ccl` is an alternative to calling the functions `CCTK_SyncGroup()` or `CCTK_SyncGroupsI()` (see the Reference Manual) from inside a routine. Using the `SYNC` keyword in the `schedule.ccl`

is the preferred method, since it provides the flesh with more information about the behaviour of your code.

As an alternative to this mechanism, synchornization can happen automatically as variables are read/written.

Besides schedule blocks, it's possible to embed C style `if/else` statements in the `schedule.ccl` file. These can be used to schedule things based upon the value of a parameter.

Example I:

If the parameter `evolve_hydro` is positively set, the Fortran routine `hydro_predictor` is scheduled to run in the *evolution* loop, after the routine `metric_predictor` and before `metric_corrector`. The routine names `metric_predictor` and `metric_corrector`, may either be real routine names from the same or a different thorn, or they may be `aliased` routine names (see the next example).

Before entry to `hydro_predictor`, storage will be allocated for one timelevel for the group of grid variables `hydro_variables` on exit from the routine this storage will be deallocated and the contents of the variables will be lost.

```
if(CCTK_Equals(evolve_hydro,"yes"))
{
  SCHEDULE hydro_predictor AT evol AFTER metric_predictor BEFORE metric_corrector
  {
    LANG:     FORTRAN
    STORAGE:  hydro_variables[1]
  } "Do a predictor step on the hydro variables"
}
```

If the parameter `evolve_hydro` is set negatively, the `hydro_predictor` routine will not be called by the scheduler. Note that if the `evolve_hydro` parameter is `STEERABLE`, it can be dynamically scheduled and de-scheduled during a run if a steering interface is available.

Example II:

The thorns `WaveToy77` and `WaveToyC`, each provide a routine to evolve the 3D wave equation: `WaveToyF77_Evolution` and `WaveToyC_Evolution`. The routine names have to be different, so that both thorns can be compiled at the same time, their functionality is identical though. Either one of them can then be activated at run time in the parameter file via `ActiveThorns`.

Since each evolution routine provides the same functionality, it makes sense to schedule them under the common alias `WaveToy_Evolution` to allow relative scheduling (`BEFORE/AFTER`) independent of the actual routine name (which may change depending on the activation in the parameter file).

In both cases, the group of variables `scalarfield` are synchronised across processes when the routine is exited.

```
schedule WaveToyF77_Evolution AS WaveToy_Evolution AT evol
{
  LANG: Fortran
```

```
  STORAGE: scalartmps
  SYNC: scalarfield
} "Evolution of 3D wave equation"

schedule WaveToyC_Evolution AS WaveToy_Evolution AT evol
{
  LANG: C
  STORAGE: scalartmps
  SYNC: scalarfield
} "Evolution of 3D wave equation"
```

The thorn `IDScalarWave` schedules the routine `WaveBinary` after the alias `WaveToy_Evolution`. It is scheduled independently of the C or Fortran routine name.

```
schedule WaveBinary AT evol AFTER WaveToy_Evolution
{
  STORAGE: wavetoy::scalarevolve
  LANG:    Fortran
} "Provide binary source during evolution"
```

Storage Outside of Schedule Blocks The keyword `STORAGE` can also be used outside of the schedule blocks to indicate that storage for these groups should be switched on at the start of the run. Note that the storage is only allocated in this way at the start; a thorn could explicitly switch the storage off (although this is not recommended practise). As for the `STORAGE` statement in schedule blocks, each group must also specify how many timelevels to activate storage for.

The `configuration.ccl`

The `configuration.ccl` file is optional. It can be used for two purposes: to detect certain features of the host system, such as the presence or absence of libraries, variable types, etc, or the location of libraries; or to provide access to certain functions too complex or otherwise not suitable for function aliasing.

The basic concept here is that a thorn can either provide or use a **capability**. A thorn providing a capability can specify a script which is run by the CST to detect features and write any configuration files; the script may output lines to its standard output to inform the CST of features to: add to the header files included by thorns using this capability; add to the make files used to build thorns using this capability; or add to the main Cactus link line. The script may also indicate that this capability is not present by returning a non-zero exit code—e.g. if the thorn is providing access to an external library, it should return an error if the library is not installed on the system.

A thorn may either require a capability to be present, in which case it is an error if there is no thorn providing that capability in the configuration's `ThornList`, or it may optionally use a capability, in which case a macro is defined in the thorn's header file if a thorn providing the capability is present.

A configuration.ccl file has the form:

```
PROVIDES <My_Capability>
{
  SCRIPT <My_ConfigScript>
```

```
  LANG <My_Language>
}

REQUIRES  <Another_Capability>

OPTIONAL <Yet_Another_Capability>
{
%  DEFINE <macro>
}
```

which states that this thorn provides the capability `My_Capability`, and a script `MyConfigScript` should be run to detect features of this capability; the script is in language `My_Language`—the CST will use the appropriate environment or interpreter to invoke the script.

The syntax of the output of the configure script is described in Appendix D2.5.1.

C1.2.4 Naming Conventions for Source Files

The make system uses file extensions to designate coding language, as well as other properties of the code in the file.

The following extensions are understood:

Extension	Language	Preprocess
`.c`	C	yes
`.cc` or `.C`	C++	yes
`.cl`	OpenCL	yes
`.cu`	CUDA	yes
`.F` or `.F77`	Fortran (fixed-format)	yes
`.f` or `.f77`	Fortran (fixed-format)	no
`.F90`	Fortran (free-format)	yes
`.f90`	Fortran (free-format)	no

In order to use Cactus `#include` directives in a file, it must be preprocessed.

A complete description of Fortran fixed and free format can be found in any textbook on Fortran. The most obvious differences are that in fixed format, code must begin after the 5th column and line continuations are indicated by a character in column 5, while in free format, lines can begin anywhere, and line continuations are indicated by an ampersand at the end of the line to be continued. Also note that statement labels are handled very differently.

The following restrictions apply to file names:

- For portability across all operating systems, the base names for any particular extension should not depend on the operating system being case sensitive (e.g. having `MyFile.c` and `MYFILE.f` is allright, but `MyFile.c` and `MYFILE.c` could cause problems).
- Currently, all source files within a thorn must have distinct names, regardless of whether they are placed in different subdirectories. We hope to relax this in future. Different thorns may have files with the same names, however.

C1.2.5 Adding Source Files

By default, the CCTK looks in the `src` directory of the thorn for source files.

There are two ways in which to specify the sources. The easiest is to use the `make.code.defn` based method in which the CCTK does all the work, but you may instead put a `Makefile` in the `src` directory and do everything yourself.

`make.code.defn` based thorn building

This is the standard way to compile your thorn's source files. The Cactus make system looks for a file called `make.code.defn` in that directory (if there is no file called `Makefile` in the `src` directory). At its simplest, this file contains two lines

- `SRCS = <`*`list of all source files in this directory`*`>`
- `SUBDIRS = <`*`list of all subdirectories, including subdirectories of subdirectories`*`>`

Each subdirectory listed should then have a `make.code.defn` file containing just a `SRCS =` line, a `SUBDIRS =` line will be ignored.

In addition, each directory can have a `make.code.deps` file, which, for files in that directory, can contain additional make rules and dependencies for files in that directory. See the GNU Make documentation for complete details of the syntax.

`Makefile` based thorn building

This method gives you the ultimate responsibility. The only requirement is that a library called `$NAME` be created by the `Makefile`.

The makefile is passed the following variables

`$(CCTK_HOME)`	the main Cactus directory
`$(TOP)`	the configuration directory
`$(SRCDIR)`	the directory in which the source files can be found
`$(CONFIG)`	the directory containing the configuration files
`$(THORN)`	the thorn name
`$(SCRATCH_BUILD)`	the scratch directory where Fortran module files should end up if they need to be seen by other thorns.
`$(NAME)`	the name of the library to be built

and has a working directory of `<`*`config`*`>/build/<`*`thorn_name`*`>` .

Other makefile variables

- `CPP`: The C preprocessor which is used to preprocess C and C++ source code, and to determine the C and C++ make dependencies
- `FPP`: The C preprocessor which is used to preprocess Fortran source code
- `CPPFLAGS`: Flags which are passed to `CPP`, to `CC`, and to `CXX`
- `FPPFLAGS`: Flags which are passed to `FPP`
- `CC`: The C compiler
- `CXX`: The C++ compiler
- `CUCC`: The CUDA compiler
- `F90`: The Fortran compiler
- `F77`: Same as `F90`
- `CFLAGS`: Flags which are passed to `CC`
- `CXXFLAGS`: Flags which are passed to `CXX`
- `CUCCFLAGS`: Flags which are passed to `CUDA`
- `F90FLAGS`: Flags which are passed to `F90`
- `F77FLAGS`: Same as `F90FLAGS`
- `LD`: The binder. This should not be directly `ld`, but should be a compiler driver such as C++. Often, `LD` is the same as `CXX`
- `LDFLAGS`: Flags which are passed to `LD`

Note that there are no makefile variables to specify an OpenCL compiler or its flags. OpenCL is implemented as a library, and code is compiled at run time via library functions to which the source code is passed as a string. OpenCL source code (files with the extension `.cl`) are thus not compiled when a Cactus configuration is built. Instead, the content of `.cl` files is converted into a string and placed into the executable. These strings have the type `char const *` in C, and can be accessed at run time under a (globally visible) name `OpenCL_source_THORN_FILE`, where `THORN` and `FILE` and are the thorn name and file name, respectively.

C1.3 Cactus Variables

A *grid variable* is a Cactus program variable passed among thorns, (or routines belonging to the same thorn interface), by way of calls to the flesh. They are the only variables known to Cactus. Such variables represent values of functions on the computational grid, and are, therefore, often called *grid functions*.

In the Cactus context, grid variables are often referred to simply as *variables*.

Cactus variables are used instead of local variables for a number of reasons:

- Cactus variables can be made visible to other thorns, allowing thorns to communicate and share data.
- Cactus variables can be distributed and communicated among processors, allowing parallel computation.
- A database of Cactus variables, and their attributes, is held by the flesh, and this information is used by thorns, for example, for obtaining a list of variables for checkpointing.
- Many Cactus APIs and tools can only be used with Cactus variables.
- Cactus provides features for error checking based on Cactus variables and their attributes.

Cactus variables are collected into *groups*. All variables in a group are of the same data type, and have the same attributes. Most Cactus operations act on a group as a whole. A group must be declared in its thorn's `interface.ccl` file.

The specification for a group declaration (fully described in Appendix D2.2) is,

```
<data_type> <group_name> [TYPE=<group_type>] [DIM=<dim>] [TIMELEVELS=<num>]
   [SIZE=<size in each direction>] [DISTRIB=<distribution_type>]
   [GHOSTSIZE=<ghostsize>]
[{
 [ <variable_name>[,]<variable_name>
   <variable_name> ]
} ["<group_description>"] ]
```

Currently, the names of groups and variables must be distinct.

C1.3.1 Data Type

Cactus supports integer, real, complex and character variable types, in various different sizes. (Sizes in the following refer to the number of bytes occupied by the a variable of the type).

INTEGER	`CCTK_INT`, `CCTK_INT1`, `CCTK_INT2`, `CCTK_INT4`, `CCTK_INT8`. (`CCTK_INT` defaults to being `CCTK_INT4`).
REAL	`CCTK_REAL`, `CCTK_REAL4`, `CCTK_REAL8`, `CCTK_REAL16`. (`CCTK_REAL` defaults to being `CCTK_REAL8`).
COMPLEX	`CCTK_COMPLEX`, `CCTK_COMPLEX8`, `CCTK_COMPLEX16`, `CCTK_COMPLEX32`. (`CCTK_COMPLEX` defaults to being `CCTK_COMPLEX16`).
BYTE	This is a 1 byte data type.

Normally a thorn should use the default types—`CCTK_INT`, `CCTK_REAL`, `CCTK_COMPLEX`—rather than explicitly setting the size, as this gives maximum portability. Also, the defaults can be changed at configuration time (see Section B2.1.1), and this allows people to compile the code with different precisions to test for roundoff effects, or to run more quickly with a lower accuracy.

C1.3.2 Group Types

Groups can be either *scalars*, *grid functions* (GFs), or *grid arrays*. Different attributes are relevant for the different group types.

`SCALAR` This is just a single number, e.g. the total energy of some field. These variables aren't communicated between processors—what would be the result of such communication?

`GF` This is the most common group type. A GF is an array with a specific size, set at run time in the parameter file, which is distributed across processors. All GFs have the same size, and the same number of ghostzones. Groups of GFs can also specify a dimension, and number of timelevels.

`ARRAY` This is a more general form of the GF. Each group of arrays can have a distinct size and number of ghostzones, in addition to dimension and number of timelevels. The drawback of using an array over a GF is that a lot of data about the array can only be determined by function calls, rather than the quicker methods available for GFs.

C1.3.3 Timelevels

These are best introduced by an example using finite differencing. Consider the 1-D wave equation

$$\frac{\partial^2 \phi}{\partial t^2} = \frac{\partial^2 \phi}{\partial x^2} \tag{C1.1}$$

To solve this by partial differences, one discretises the derivatives to get an equation relating the solution at different times. There are many ways to do this, one of which produces the following difference equation

$$\phi(t+\Delta t, x) - 2\phi(t,x) + \phi(t-\Delta t, x) = \frac{\Delta t^2}{\Delta x^2}\{\phi(t, x+\Delta x) - 2\phi(t,x) + \phi(t, x-\Delta x)\} \tag{C1.2}$$

which relates the three timelevels $t+\Delta t$, t, and $t-\Delta t$.

Obviously, the code could just use three variables, one for each timelevel. This turns out, however, to be inefficient, because as soon as the time is incremented to $t+\Delta t$, it would be necessary to copy data from the t variable to the $t-\Delta t$ variable and from the $t+\Delta t$ variable to the t variable.

To remove this extraneous copy, Cactus allows you to specify the number of timelevels used by your numerical scheme. It then generates variables with the base name (e.g. `phi`) suffixed by a qualifier for which timelevel is being referred to—no suffix for the next timelevel, and `_p` for each previous timelevel.

The timelevels are rotated (by the driver thorn) at the start of each evolution step, that is:

```
initial
poststep
analysis

loop:
  rotate timelevels
  t = t + dt
```

```
it = it + 1
prestep
evolve
poststep
analysis
```

Timelevel rotation means that, for example, `phi_p` now holds the values of the former `phi`, and `phi_p_p` the values of the former `phi_p`, etc. Note that after rotation, `phi` is undefined, and its values should not be used until they have been updated.

All timelevels, except the current level, should be considered *read-only* during evolution, that is, their values should not be changed by thorns. The exception to this rule is for function initialisation, when the values at the previous timelevels do need to be explicitly filled out.

C1.3.4 Size and Distrib

A Cactus grid function or array has a size set at runtime by parameters. This size can either be the global size of the array across all processors (`DISTRIB=DEFAULT`), or, if `DISTRIB=CONSTANT`, the specified size on *each* processor. If the size is split across processors, the driver thorn is responsible for assigning the size on each processor.

C1.3.5 Ghost Zones

Cactus is based upon a *distributed computing* paradigm. That is, the problem domain is split into blocks, each of which is assigned to a processor. For hyperbolic and parabolic problems the blocks only need to communicate at the edges.

To illustrate this, take the example of the wave equation again. Equation C1.2 describes a possible time-evolution scheme. On examination, you can see that to generate the data at the point $(t + \Delta t,\ x)$ we need data from the four points $(t,\ x)$, $(t - \Delta t,\ x)$, $(t,\ x + \Delta x)$, and $(t,\ x - \Delta x)$ only. Ignoring the points at x, which are obviously always available on a given processor, you can see that the algorithm requires a point on either side of the point x, i.e. this algorithm has *stencil size* 1. Similarly algorithms requiring two points on either side have stencil size 2, etc.

Now, if you evolve the above scheme, it becomes apparent that at each iteration the number of grid points you can evolve decreases by one at each edge (see Figure C1.1).

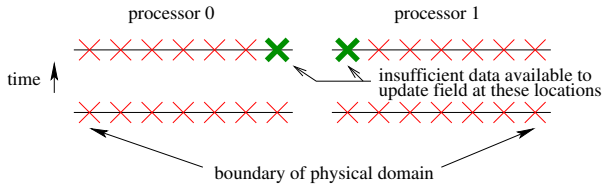

Figure C1.1: Distributed wave equation with no ghostzones

At the outer boundary of the physical domain, the data for the boundary point can be generated by the boundary conditions, however, at internal boundaries, the data has to be copied from the adjacent processor. It would be inefficient to copy each point individually, so instead, a number of *ghostzones* are

created at the internal boundaries. A ghostzone consists of a copy of the whole plane (in 3D, line in 2D, point in 1D) of the data from the adjacent processor. That is, the array on each processor is augmented with copies of points from the adjacent processors, thus allowing the algorithm to proceed *on the points owned by this processor* without having to worry about copying data. Once the data has been evolved one step, the data in the ghostzones can be exchanged (or *synchronised*) between processors in one fell swoop before the next evolution step. (See Figure C1.2.) Note that you should have at least as many ghostzones as your stencil size requires.

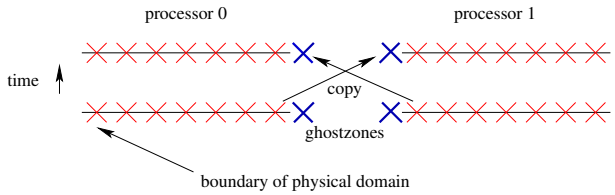

Figure C1.2: Distributed wave equation with ghostzones

C1.3.6 Information about Grid Variables

The flesh holds a database with information related to grid variables, and provides a set of querying APIs.

Group Information

Fundamental information about grid functions (e.g. local grid size and location, number of ghostzones) is passed through the argument list of scheduled routines (see Section C1.6.2). To obtain similar information from non-scheduled routines, or for general grid variables, a set of functions are provided, the last two letters of which specify whether the information is requested using a group name (`GN`) or index (`GI`), or a variable name (`VN`) or index (`VI`).

`CCTK_Grouplsh[GN|GI|VN|VI]`
: An array of integers with the local grid size on this processor.

`CCTK_Groupgsh[GN|GI|VN|VI]`
: An array of integers with the global grid size.

`CCTK_Groupbbox[GN|GI|VN|VI]`
: An array of integers which indicate whether the boundaries are internal boundaries (e.g. between processors), or physical boundaries. A value of `1` indicates a physical (outer) boundary at the edge of the computational grid, and `0` indicates an internal boundary.

`CCTK_Groupnghostzones[GN|GI|VN|VI]`
: An array of integers with the number of ghostzones used in each direction.

`CCTK_Grouplbnd[GN|GI|VN|VI]`
: An array of integers containing the lowest index (in each direction) of the local grid, as seen on the global grid. Note that these indices start from zero, so you need to add one when using them in Fortran thorns.

`CCTK_Groupubnd[GN|GI|VN|VI]`
: An array of integers containing the largest index (in each direction) of the local grid, as seen on the global grid. Note that these indices start from zero, so you need to add one when using them in Fortran thorns.

C1.4 Cactus Parameters

Parameters are the means by which the user specifies the runtime behaviour of the code. Each parameter has a data type and a name, as well as a range of allowed values and a default value. These are declared in the thorn's `param.ccl` file.

The thorn determines which parameters can be used in other thorns by specifying a *scope* for the thorn, as explained in Section C1.4.2.

The user may specify initial values for parameters in the parameter file (see Section B3.2); the flesh validates these values against the parameters' allowed ranges. Otherwise, the initial value of the parameter is taken to be its default. Once validated, parameter values are fixed, and cannot be changed, unless the parameter is specified to be *steerable* (see C1.4.3). For a detailed discussion of the `param.ccl` syntax, see Appendix D2.3.

The full specification for a parameter declaration is

```
[EXTENDS|USES] <parameter_type>[[<size>]] <parameter name> "<parameter description>"
{
  <PARAMETER_RANGES>
} <default value>
```

You can obtain lists of the parameters associated with each thorn using the Cactus command line options `-o` and `-O` (Section B3.1).

C1.4.1 Types and Ranges

Parameters can be of these types:

Int	Can take any integral value
Real	Can take any floating point value
Keyword	Can have a value consisting of one of a choice of strings
Boolean	Can be true or false (`1`, `t`, `true`, or `0`, `f`, `false`)
String	Can have any string value

Each parameter can be validated against a set of allowed *ranges*, each of which has a description associated with it. The nature of the range is determined by the type of parameter, as follows:

Int

The range specification is of the form

lower:*upper*:*stride*

where *lower* and *upper* specify the lower and upper allowed range, and *stride* allows numbers to be be missed out, e.g.

1:21:2

means the value must be an odd number between one and twenty-one (inclusive).

A missing end of range (or a '*') indicates negative or positive

Note that if *stride* is specified, then *upper* must be specified, or '*' (i.e. the specifier '1::2' is not legal)

infinity, and the default stride is one. A '(' (or ')') before (or after) the lower (or upper) range specifies an open endpoint.

Note that '*' is not allowed for the lower bound.

Real

The range specification is of the form

lower:*upper*

where *lower* and *upper* specify the lower and upper allowed range. A missing end of range (or a '*') implies negative or positive infinity. The above is inclusive of the endpoints. A '(' (or ')') before (or after) the lower (or upper) range specifies an open endpoint. Likewise, a '[(or ']') before (or after) the lower (or upper) range specifies a closed endpoint.

The numbers written in a `param.ccl` file are interpreted as C code. To express a number in 'scientific notation', use, e.g. '1e-10', which is a double precision constant in C. (If the floating precision of the variable to which it is assigned is not double, then C will typecast appropriately. If you *really* want to specify a single precision floating constant, or a long double constant, append the number with `f` or `l` respectively.)

Keyword

The range specification consists of a string, which will be matched in a case insensitive manner.

Boolean

There is no range specification for this type of parameter.

String

The range is a POSIX regular expression. On some machines you may be able to use extended regular expressions, but this is not guaranteed to be portable.

C1.4.2 Scope

Parameters can be `GLOBAL`, `RESTRICTED`, or `PRIVATE`. Global parameters are visible to all thorns. Restricted parameters are visible to any thorn which chooses to `USE` or `EXTEND` it. A private parameter is only visible to the thorn which declares it. The default scope is `PRIVATE`.

C1.4.3 Steerable

A parameter can be changed dynamically if it is specified to be *steerable* (see Section D2.3.2). It can then be changed by a call to the flesh function `CCTK_ParameterSet` (see the Reference Guide for a description of this function).

C1.5 Scheduling

Cactus contains a rule-based scheduling system, which determines which routines, from which thorns are run in which order. The scheduler determines if the specifications are inconsistent, but does allow the user to schedule a routine with respect to another routine which may not exist. For a detailed discussion of the `schedule.ccl` syntax see Appendix D2.4.

A usual simple specification for a schedule declaration is

```
schedule <function name> AT <schedule bin>
{
  LANG: <language>
  [STORAGE:       <group>[[timelevels]],<group>[[timelevels]]...]
} "Description of function"
```

The full specification for a schedule declaration is

```
schedule [GROUP] <function|schedule group name> AT|IN <schedule bin|group name>
         [AS <alias>]
         [WHILE <variable>] [IF <variable>]
         [BEFORE|AFTER <item>|(<item> <item> ...)]
{
  LANG: <language>
  [STORAGE:       <group>[[timelevels]],<group>[[timelevels]]...]
  [TRIGGER:       <group>,<group>...]
  [SYNC:          <group>,<group>...]
  [OPTIONS:       <option>,<option>...]
} "Description of function or schedule group"
```

This full schedule specification consists of a mandatory part, a set of options, and the main body limited by braces, referred to as the `schedule block`.

Each schedule item is scheduled either `AT` a particular *`scheduling bin`*, or `IN` a schedule *`group`*.

C1.5.1 Schedule Bins

These are the main times at which scheduled functions are run, from fixed points in the flesh and driver thorn (schedule bins can easily be traversed from any thorn, although this is not usual). When a schedule bin is *traversed*, all the functions scheduled in that particular are called, in the manner described in Section C1.5.5 and respecting the requested ordering of functions(Section C1.5.3). In the absence of any ordering, functions in a particular schedule bin will be called in an undetermined order.

The schedule bins are described in Section C1.2.3. Note that the preceding `CCTK_` is optional for the use of the bin names in the `schedule.ccl` file.

C1.5.2 Groups

If the optional `GROUP` specifier is used, the item is a schedule group rather than a normal function. Schedule groups are effectively new, user-defined, schedule bins. Functions or groups may be scheduled `IN` these, in the same way as they are scheduled `AT` the main schedule bins. (That is, groups may be nested.)

C1.5.3 Schedule Options

The options define various characteristics of the schedule item.

`AS` This assigns a new name to a function for scheduling purposes. This is used, for instance, to allow a thorn to schedule something before or after a routine from another implementation; two thorns providing this implementation can schedule a routine `AS` the same thing, thus allowing other thorns to operate independently of which one is active.

`WHILE` This specifies a `CCTK_INT` grid scalar which is used to control the execution of this item. As long as the grid scalar has a nonzero value, the schedule item will be executed repeatedly. This allows dynamic behaviour with scheduling.

`IF` This specifies a `CCTK_INT` grid scalar which is used to control the execution of this item. If the grid scalar has a nonzero value, the schedule item will be executed, otherwise the item will be ignored. This allows dynamic behaviour with scheduling.

If both an `IF` and a `WHILE` clause are present, then the schedule is executed according to the following pseudocode:

```
IF condition
  WHILE condition
    SCHEDULE item
  END WHILE
END IF
```

`BEFORE` or `AFTER` These specify either

- a function or group before or after which this item will be scheduled, or
- a list of functions and/or groups; this item will be scheduled (once) before any of them or after all of them respectively.

Note that a single schedule item may have multiple `BEFORE` and/or `AFTER` options; the scheduler will honor all of these (or abort with a fatal error). For example,

```
schedule FOO BEFORE A BEFORE B BEFORE C ...
```

schedules `FOO` before any of `A`, `B`, or `C`. This can also be written

```
schedule FOO BEFORE (A B C) ...
```

Note that the set of all `BEFORE`/`AFTER` options in all active schedule blocks of all active thorns, *must* specify a (directed) graph with no cycles; if there are any cycles, then the scheduler will abort with a fatal error.

C1.5.4 The Schedule Block

The schedule block specifies further details of the scheduled function or group.

`LANG` This specifies the language of the routine. Currently this is either C or Fortran. C++ routines should be defined as `extern "C"` and registered as `LANG: C`.

`STORAGE` The `STORAGE` keyword specifies groups for which memory should be allocated for the duration of the routine or schedule group. The storage status reverts to its previous status after completion of the routine or schedule group. Each group must specify how many timelevels to activate storage for, from 1 up to the maximum number for the group as specified in the defining `interface.ccl` file. If the maximum is 1 (the default) this number may be omitted. Alternatively ***`timelevels`*** can be the name of a parameter accessible to the thorn. The parameter name is the same as used in C routines of the thorn, fully qualified parameter names of the form ***`thorn::parameter`*** are not allowed. In this case 0 (zero) ***`timelevels`*** can be requested, which is equivalent to the `STORAGE` statement being absent.

`TRIGGER` This is only used for items scheduled at timebin `CCTK_ANALYSIS`. The item will only be executed if output is due for at least one variable in one of the listed groups. (The item will also be called if there is no group listed.)

`SYNC` On exit from this item, the ghost zones of the listed groups will be exchanged.

`OPTIONS` This is for miscellaneous options. The list of accepted options is given in Appendix D2.4.2.

C1.5.5 How Cactus Calls Scheduled Functions

For each scheduled function called, the flesh performs a variety of jobs at entry and exit.

On entry to a scheduled routine, if the routine is being called at the `CCTK_ANALYSIS` timebin first, a check is made to see if the routine should actually be called on this timestep. For this, all grid variables in the trigger groups for the routine are checked with all registered output methods to determine if it is time to output any triggers. The routine will only be called if at least one is due to be output. Note that once a grid variable has been analyzed, it gets marked as such, and will not be analyzed again during this iteration. (Note that a routine without any trigger groups will also be called.) Thus, if more than one analysis routine should be triggered on the same trigger variable(s), they must be scheduled in a single group. Routines from all timebins, other than `ANALYSIS`, are always called.

Next, storage is assigned for any required variables, remembering the original state of storage.

The routine is then called, and on exit, any required grid variables are first synchronised. Following synchronization, any required output methods are called for the triggers. Finally, the storage of grid variables is returned to the original state.

C1.6 Writing a Thorn

C1.6.1 Thorn Programming Languages

When you start writing a new thorn, the first decision to make is which programming language to use. The source code in Cactus thorns can be written in any mixture of C, C++, CUDA, or Fortran. The following points should be considered when choosing a language to work in:

- All functions designed for application thorn writers are available in all languages, however, some interfaces for infrastructure thorn writing are only available from C or C++.

Whatever language you choose, if you want your thorn to be portable, and compile and run on multiple platforms, stick to the standards and don't use machine dependent extensions.

C1.6.2 What the Flesh Provides

The flesh provides for thorns:

`Variables`

`Parameters`

`Cactus Functions`

- Driver (parallelisation) utilities
- I/O utilities
- Coordinates utilities
- Reduction utilities
- Interpolation utilities
- Information utilities

Fortran Routines

Any source file using Cactus infrastructure should include the header file `cctk.h` using the line

```
#include "cctk.h"
```

(Fortran programmers should not be put off by this being a C style header file—most Cactus files are run through a C preprocessor before compilation.)

Variables Any routine using Cactus argument lists (for example, all routines called from the scheduler at time bins between `CCTK_STARTUP` and `CCTK_SHUTDOWN`) should include at the top of the file the header

```
#include "cctk_Arguments.h"
```

A Cactus macro `CCTK_ARGUMENTS` is defined for each thorn to contain:

- General information about the grid hierarchy, for example, the number of grid points used. See Section C1.6.2 for a complete list.
- All the grid variables defined in the thorn's `interface.ccl`
- All the grid variables required from other thorns as requested by the `inherits` and `friend` lines in the `interface.ccl`

These variables must be declared at the start of the routine using the macro `DECLARE_CCTK_ARGUMENTS`. Alternatively, they can be declared with `DECLARE_CCTK_ARGUMENTS_CHECKED(<function name>)`, where `<function name>` is the name of the function as declared in `schedule.ccl`. The function-name specific version of the macro will declare only those variables identified by READS/WRITES directives in `schedule.ccl`, using `const` for the C/C++ variables and `intent(in)` for the Fortran variables. (Note that, for technical reasons, in Fortran variables declared in `DECLARE_CCTK_ARGUMENTS` but not in the `READS/WRITES` directives are actually declared as `intent(in), type(cctki_inaccessible_grid_variable)`, and are thus unusable.)

To pass the arguments to another routine in the same thorn use the macro `CCTK_PASS_FTOF` in the calling routine, and again the macro `CCTK_ARGUMENTS` in the receiving routine.

Note that you cannot use Cactus argument lists in routines scheduled at the `CCTK_STARTUP` and `CCTK_SHUTDOWN` time bins, because at this time no grid hierarchy exists.

Parameters All parameters defined in a thorn's `param.ccl` and all `global` parameters, appear as local variables of the corresponding `CCTK` data type in Fortran source code, i.e. Booleans and Integers appear as `CCTK_INT` types (with nonzero/zero values for boolean `yes/no`), Reals as `CCTK_REAL`, and Keywords and String parameters as `CCTK_STRING` (see also note below). These variables are *read only*, and *changes should not be made to them.* The effect of changing a parameter is undefined (at best).

Any routine using Cactus parameters should include at the top of the file the header

```
#include "cctk_Parameters.h"
```

The parameters should be declared at the start of the routine using them with the macro `DECLARE_CCTK_PARAMETERS`.

In Fortran, special care should be taken with string valued parameters. These parameters are passed as C pointers, and can not be treated as normal Fortran strings. To compare a string valued parameter and Fortran string, use the macro `CCTK_EQUALS()` or the function `CCTK_Equals()` (see the reference manual for a description of the `CCTK_` functions). To print the value of a string valued parameter to screen, use the subroutine `CCTK_PrintString()`. A further function `CCTK_FortranString` provides a mechanism for converting a string parameter to a Fortran string. For example, if `operator` is a Cactus string parameter holding the name of a reduction operator whose handle you need to find, you cannot pass it directly into the subroutine `CCTK_LocalArrayReductionHandle`, which is expecting a Fortran string. Instead, the following is needed:

```
      character*200 fortran_operator
      CCTK_INT      fortran_operator_len
      integer       handle

      call CCTK_FortranString(fortran_operator_len,operator,fortran_operator)
      call CCTK_LocalArrayReductionHandle(handle,fortran_operator(1:fortran_operator_len))
```

Fortran Example The Fortran routine, `MyFRoutine`, is scheduled in the `schedule.ccl` file, doesn't use Cactus parameters, and calls another routine, in the same thorn, `MyNewRoutine`, which does use parameters. This routine needs to be passed an integer flag as well as the standard Cactus variables. The source file should look like

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

      subroutine MyFRoutine(CCTK_ARGUMENTS)

c     I'm very cautious, so I want to declare all variables
      implicit none

      DECLARE_CCTK_ARGUMENTS_CHECKED(MyFRoutine)
c Note: It is also possible to use
c     DECLARE_CCTK_ARGUMENTS
c However, this latter form will declare all variables
c available to the thorn and not simply those listed
c in the READS/WRITES clauses of the schedule.ccl declaration.
c
c Note: It is also possible to use all caps for the function
c name when writing in Fortran. Thus, the following also works:
c
c     DECLARE_CCTK_ARGUMENTS_CHECKED(MYFROUTINE)

      integer flag

      flag = 1
      call MyNewRoutine(CCTK_PASS_FTOF,flag)

      return
```

```
      end

      subroutine MyNewRoutine(CCTK_ARGUMENTS,flag)

      implicit none

      DECLARE_CCTK_ARGUMENTS_CHECKED(MyNewRoutine)
      DECLARE_CCTK_PARAMETERS
      integer flag

c     Main code goes here

      return
      end
```

Cactus Fortran Functions Cactus Fortran functions, for example, `CCTK_MyProc` and `CCTK_Equals`, can all be declared by adding the statement

```
#include "cctk_Functions.h"
```

near the top of the file, and adding the declaration

```
DECLARE_CCTK_FUNCTIONS
```

to a module or a subroutine after the `implicit none` statement, but before any executable code.

Fortran Modules Fortran modules should be placed into source files that have the same name as the module, followed by the corresponding file name suffix. A module `metric` should thus be placed, e.g. into a file `metric.F90`. This convention allows the Cactus build system to automatically deduce the compile time dependencies.

If you do not follow this convention, then you have to include the modules into the thorn's `make.code.deps` file (Section C1.2.5) to ensure they are compiled before the routines which use them. This is especially important for parallel building. For example, if a routine in `MyRoutine.F90` uses a module in `MyModule.F90`, then add the line:

```
MyRoutine.F90.o:        MyModule.F90.o
```

The `MOD` function The intrinsic function `MOD` in Fortran takes two integer arguments, which should both be of the same type. This means that it may be necessary to cast the arguments to, e.g. `INT` for some architectures. This can occur in particular when a `CCTK_INT` parameter and the Cactus variable `cctk_iteration` (which is declared to be `INTEGER`) are used, in which case the correct code is

```
MOD(cctk_iteration,INT(MyParameter))
```

C Routines

Any source file using Cactus infrastructure should include the header file `cctk.h` using the line

```
#include "cctk.h"
```

Variables Any routine using Cactus argument lists (for example, all routines called from the scheduler at time bins between `CCTK_STARTUP` and `CCTK_SHUTDOWN`), should include at the top of the file the header

```
#include "cctk_Arguments.h"
```

A Cactus macro `CCTK_ARGUMENTS` is defined for each thorn to contain

- General information about the grid hierarchy, for example, the number of grid points on the processor. See Section C1.6.2 for a complete list.
- All the grid variables defined in the thorn's `interface.ccl`
- All the grid variables required from other thorns as requested by the `inherits` and `friend` lines in the `interface.ccl`

These variables must be declared at the start of the routine using the macro `DECLARE_CCTK_ARGUMENTS`. This macro should always be the first line of the routine.

To pass the arguments to another routine in the same thorn, use the macro `CCTK_PASS_CTOC` in the calling routine, and again the macro `CCTK_ARGUMENTS` in the receiving routine.

Note that you cannot use Cactus argument lists in routines scheduled at the `CCTK_STARTUP` and `CCTK_SHUTDOWN` time bins, because at this time no grid hierarchy exists.

Parameters All parameters defined in a thorn's `param.ccl` and all `global` parameters, appear as local variables of the corresponding `CCTK` data type in C source code, i.e. Integers and Booleans appear as `CCTK_INT` types (with nonzero/zero values for boolean `yes/no`), Reals as `CCTK_REAL`, and Keywords and String parameters as `CCTK_STRING`. These variables are *read only*, and *changes should not be made to them*. The effect of changing a parameter is undefined (at best).

Any routine using Cactus parameters should include at the top of the file the header

```
#include "cctk_Parameters.h"
```

The parameters should be declared as the last statement in the declaration part of the routine using them with the macro `DECLARE_CCTK_PARAMETERS`.

Example The C routine `MyCRoutine` is scheduled in the `schedule.ccl` file, and uses Cactus parameters. The source file should look like

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

void MyCRoutine(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS_CHECKED(MyCRoutine)
  DECLARE_CCTK_PARAMETERS

  /* Here goes your code */
}
```

Complex variables Cactus supports complex grid variables of type `CCTK_COMPLEX` which are realized through the types `complex`, `std::complex` and `COMPLEX` in C, C++ and Fortran respectively. Complex number support is available in C in the C99 language standard which Cactus requires.

There is a known incompatibility when *returning* complex numbers from C and Fortran functions to C++ callers on some architectures. Access to variables through pointers and in arrays is not affected. A workaround is not to return values but instead pass in a pointer to a local variable to hold the return value.

Specifically for C Programmers Grid functions are held in memory as 1-dimensional C arrays. These are laid out in memory as in Fortran. This means that the first index should be incremented through most rapidly. This is illustrated in the example below.

Cactus provides macros to find the 1-dimensional index which is needed from the multidimensional indices which are usually used. There is a macro for each dimension of grid function. Below is an artificial example to demonstrate this using the 3D macro `CCTK_GFINDEX3D`:

```
for (k=0; k<cctk_lsh[2]; k++)
{
  for (j=0; j<cctk_lsh[1]; j++)
  {
    for (i=0; i<cctk_lsh[0]; i++)
    {
      int const ind3d = CCTK_GFINDEX3D(cctkGH,i,j,k);
      rho[ind3d] = exp(-pow(r[ind3d],2));
    }
  }
}
```

Here, `CCTK_GFINDEX3D(cctkGH,i,j,k)` expands to

```
((i) + cctkGH->cctk_lsh[0]*((j)+cctkGH->cctk_lsh[1]*(k)))
```

Note: In Fortran, grid functions are accessed as Fortran arrays, i.e. simply as `rho(i,j,k)`.

To access vector grid functions (vector grid functions are a "vector" of grid functions; see section D2.2.4), one also needs to specify the vector index. This is best done via the 3D macro `CCTK_VECTGFINDEX3D`:

```
for (k=0; k<cctk_lsh[2]; k++)
{
  for (j=0; j<cctk_lsh[1]; j++)
  {
    for (i=0; i<cctk_lsh[0]; i++)
    {
      /* vector indices are 0, 1, 2 */
      vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,0)] = 1.0;
      vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,1)] = 0.0;
      vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,2)] = 0.0;
    }
  }
}
```

Cactus Variables

The Cactus variables which are passed through the macro `CCTK_ARGUMENTS` are

`cctkGH`	A C pointer identifying the grid hierarchy.
`cctk_dim`	An integer with the number of dimensions used for this grid hierarchy.
`cctk_lsh`	An array of `cctk_dim` integers with the local grid size on this processor.
`cctk_ash`	An array of `cctk_dim` integers with the allocated size of the array. This may be larger than the local size; the additional points may not be used.
`cctk_gsh`	An array of `cctk_dim` integers with the *global* grid size.
`cctk_iteration`	The current iteration number.
`cctk_delta_time`	A `CCTK_REAL` with the timestep.
`cctk_time`	A `CCTK_REAL` with the current time.
`cctk_delta_space`	An array of `cctk_dim` `CCTK_REAL`s with the grid spacing in each direction.
`cctk_nghostzones`	An array of `cctk_dim` integers with the number of ghostzones used in each direction.
`cctk_origin_space`	An array of `cctk_dim` `CCTK_REAL`s with the spatial coordinates of the global origin of the grid.

The following variables describe the location of the local grid (e.g. the grid treated on a given processor) within the global grid.

`cctk_lbnd`	An array of `cctk_dim` integers containing the lowest index (in each direction) of the local grid, as seen on the global grid. Note that these indices start from zero, so you need to add one when using them in Fortran thorns.
`cctk_ubnd`	An array of `cctk_dim` integers containing the largest index (in each direction) of the local grid, as seen on the global grid. Note that these indices start from zero, so you need to add one when using them in Fortran thorns.

`cctk_bbox` An array of 2∗`cctk_dim` integers (in the order $[\dim_0^{\min}, \dim_0^{\max}, \dim_1^{\min}, \dim_1^{\max}, \ldots]$), which indicate whether the boundaries are internal boundaries (e.g. between processors), or physical boundaries. A value of 1 indicates a physical (outer) boundary at the edge of the computational grid, and 0 indicates an internal boundary.

The following variable is needed for grid refinement methods

`cctk_levfac` An array of `cctk_dim` integer factors by which the local grid is refined in the corresponding direction with respect to the base grid.

`cctk_levoff` and `cctk_levoffdenom` Two arrays of `cctk_dim` integers describing the distance by which the local grid is offset with respect to the base grid, measured in local grid spacings. The distance in direction `dir` is given by `1.0 * cctk_levoff[dir] / cctk_levoffdenom[dir]`.

`cctk_timefac` The integer factor by which the time step size is reduced with respect to the base grid.

The following variables are used for identifying convergence levels.

`cctk_convlevel` The convergence level of this grid hierarchy. The base level is 0, and every level above that is coarsened by a factor of `cctk_convfac`.

`cctk_convfac` The factor between convergence levels. The relation between the resolutions of different convergence levels is $\Delta x_L = \Delta x_0 \cdot F^L$, where L is the convergence level and F is the convergence factor. The convergence factor defaults to 2.

The following variables are used for identifying the coordinate patch in a multipatch/multiblock simulation.

`cctk_patch` An integer with the multipatch patch number on this processor.

`cctk_npatches` An integer with the total number of multipatch patches on all processors.

The variables `cctk_delta_space`, `cctk_delta_time`, and `cctk_origin_space` denote the grid spacings, time step size, and spatial origin on the *base* grid. If you are using a grid refinement method, you need to calculate these quantities on the grid you are on. There are Cactus macros provided for this, with the syntax `CCTK_DELTA_SPACE(dir)`, `CCTK_ORIGIN_SPACE(dir)`, and `CCTK_DELTA_TIME` for both C and Fortran. It is recommended that these macros are always used to provide the grid spacings, time step sizes, and spatial origins in your thorns. In doing so, you incorporate the effects of `cctk_levfac`, `cctk_levoff`, `cctk_levoffdenom`, and `cctk_timefac`, so that you do not explicitly have to take them into account.

Putting the above information together, Figure C1.3 shows two different ways to compute the global Cactus xyz coordinates of the current grid point. Because the "alternate calculation" (the one using `Grid::x`, `Grid::y`, and `Grid::z`) gives the true global xyz coordinates even in a multipatch/multiblock context, this is generally the preferred form for general use.

```
#include <stddef.h>                     /* defines size_t */
#include "cctk.h"

void MyThorn_MyFunction(CCTK_ARGUMENTS)
{
int i,j,k;

    for (k = 0 ; k < cctk_lsh[2] ; ++k)
    {
    for (j = 0 ; j < cctk_lsh[1] ; ++j)
    {
    for (i = 0 ; i < cctk_lsh[0] ; ++i)
    {
    const size_t posn = CCTK_GFINDEX3D(cctkGH, i,j,k);

    /* calculate the global xyz coordinates of the (i,j,k) grid point */
    /* (in a multipatch/multiblock context, this gives the per-patch coordinates) */
    const CCTK_REAL xcoord = CCTK_ORIGIN_SPACE(0) + (cctk_lbnd[0] + i)*CCTK_DELTA_SPACE(0);
    const CCTK_REAL ycoord = CCTK_ORIGIN_SPACE(1) + (cctk_lbnd[1] + j)*CCTK_DELTA_SPACE(1);
    const CCTK_REAL zcoord = CCTK_ORIGIN_SPACE(2) + (cctk_lbnd[2] + k)*CCTK_DELTA_SPACE(2);

    /* an alternate calculation, which requires that this thorn inherit from  Grid */
    /* (in a multipatch/multiblock context, this gives the true global xyz coordinates) */
    const CCTK_REAL xxcoord = /* Grid:: */ x[posn];
    const CCTK_REAL yycoord = /* Grid:: */ y[posn];
    const CCTK_REAL zzcoord = /* Grid:: */ z[posn];
    }
    }
    }
}
```

Figure C1.3: This figure shows two different ways to compute the global Cactus xyz coordinates of the current grid point. Because the "alternate calculation" (the one one using `Grid::x`, `Grid::y`, and `Grid::z`) gives the true global xyz coordinates even in a multipatch/multiblock context, this is generally the preferred form for general use.

Cactus Data Types

To provide portability across platforms, the Cactus grid variables and parameters are defined and declared using Cactus data types. The most important of these data types are described below, for a full description see Section C1.9.8. These data types should be used to declare local variables where needed, and to declare Cactus grid variables or parameters that need declarations.

`CCTK_INT`	default size 4 bytes
`CCTK_REAL`	default size 8 bytes
`CCTK_COMPLEX`	consists of two `CCTK_REAL` elements

Example In the following example, `MyScalar` is a grid scalar which is declared in the `interface.ccl` as `CCTK_REAL`.

```
subroutine InitialData(CCTK_ARGUMENTS)

DECLARE_CCTK_ARGUMENTS

CCTK_REAL local_var

local_var = 1.0/3.0
MyScalar = local_var

return
end
```

Declaring `local_var` to have a non-Cactus data type, e.g. `REAL*4`, or using one of the other Cactus real data types described in Section C1.9.8, could give problems for different architectures or configurations.

C1.6.3 Parallelisation

The flesh itself does not actually set up grid variables. This is done by a *driver* thorn. To allow the distribution of a grid over a number of processors, the driver thorn must also provide the grid decomposition, and routines to enable parallelisation. The method used to provide this parallelisation (e.g. MPI, PVM) is not usually important for the thorn writer, since the driver thorn provides routines which are called by standard interfaces from the flesh. Here, we describe briefly the most important of these routines for the application thorn writer. A more detailed description of these interfaces with their arguments, is given in the Reference Manual. A complete description of the routines that a driver thorn must provide, will be provided in the Infrastructure Thorn Writers Guide (Part C2). The standard driver thorn is currently `PUGH` in the `CactusPUGH` package, which is a parallel unigrid driver.

`CCTK_nProcs`	Returns the number of processors being used
`CCTK_MyProc`	Returns the processor number (this starts at processor number zero)

`CCTK_SyncGroup, CCTK_SyncGroupsI`
: Synchronises either a single group or a set of groups of grid arrays by exchanging the values held in each processor ghostzones, with the physical values of their neighbours (see the Reference Manual)

`CCTK_Barrier`
: Waits for all processors to reach this point before proceeding

C1.7 Cactus Application Interfaces

C1.7.1 Iterating Over Grid Points

A grid function consists of a multi-dimensional array of grid points. These grid points fall into several types:

interior regular grid point, presumably evolved in time

ghost inter-process boundary, containing copies of values owned by another process

physical boundary outer boundary, presumably defined via a boundary condition

symmetry boundary defined via a symmetry, e.g. a reflection symmetry or periodicity

Grid points in the edges and corners may combine several types. For example, a point in a corner may be a ghost point in the x direction, a physical boundary point in the y direction, and a symmetry point in the z direction.

The size of the physical boundary depends on the application. The number of ghost points is defined by the driver; the number of symmetry points is in principle defined by the thorn implementing the respective symmetry condition, but will in general be the same as the number of ghost points to avoid inconsistencies.

When iterating over grid points, one usually needs to know about the boundary sizes and boundary types present. Details about this is explained in thorn CactusBase/CoordBase.

The flesh provides a set of macros to iterate over particular types of grid points:

`CCTK_LOOP_ALL` Loop over all grid points

`CCTK_LOOP_INT` Loop over all interior grid points

`CCTK_LOOP_BND` Loop over all physical boundary points

`CCTK_LOOP_INTBND` Loop over all "interior" physical boundary points, i.e. over all those physical boundary points that are not also ghost or symmetry points

As described above, points on edges and corners can have several boundary types at once, e.g. can be both a physical and a symmetry point. `LOOP_BND` and `LOOP_INTBND` treat these different: `LOOP_BND` loops over all points that are physical boundaries (independent of whether they also are symmetry or ghost boundaries), while `LOOP_INTBND` loops over those points that are only physical boundaries (and excludes any points that belongs to a symmetry or ghost boundary). `LOOP_BND` does not require applying a symmetry condition or

synchronisation afterwards (but does not allow taking tangential derivatives); `LOOP_INTBND` allows taking tangential derivatives (but requires applying symmetry boundaries and synchronising afterwards).

In 3 dimensions, these macros should be called as follows:

```
CCTK_LOOP3_ALL(name, cctkGH, i,j,k) {
  ... body of the loop
} CCTK_ENDLOOP3_ALL(name);


CCTK_LOOP3_INT(name, cctkGH, i,j,k) {
  ... body of the loop
} CCTK_ENDLOOP3_INT(name);


CCTK_LOOP3_BND(name, cctkGH, i,j,k, ni,nj,nk) {
  ... body of the loop
} CCTK_ENDLOOP3_BND(name);


CCTK_LOOP3_INTBND(name, cctkGH, i,j,k, ni,nj,nk) {
  ... body of the loop
} CCTK_ENDLOOP3_INTBND(name);
```

and similarly for 2 dimensions and 1 dimension.

In all cases, `name` should be replaced by a unique name for the loop. `i`, `j`, and `k` are names of variables that will be declared and defined by these macros, containing the index of the current grid point. Similarly `ni`, `nj`, and `nk` are names of variables describing the (outwards pointing) normal direction to the boundary as well as the distance to the boundary.

`OpenMP` All `CCTK_LOOP` macros include support for `OpenMP` based multi-threading when called inside of an `OpenMP` parallel section. They do no themselves create such a parallel section and therefore should be preceded by a `#pragma omp parallel`:

```
#pragma omp parallel
CCTK_LOOP3_ALL(name, cctkGH, i,j,k) {
  ... body of the loop
} CCTK_ENDLOOP3_ALL(name);
```

and similar for `CCTK_LOOP3_INT`, `CCTK_LOOP3_BND`, and `CCTK_LOOP3_INTBND`.

C1.7.2 Coordinates

The flesh provides utility routines for registering and querying coordinate information. The flesh does not provide any coordinates itself, these must be supplied by a thorn. Thorns are not required to register coordinates to the flesh, but registering coordinates provides a means for infrastructure thorns to make use of coordinate information.

Coordinates are grouped into *coordinate systems*, which have a specified dimension. Any number of coordinate systems can be registered with the flesh, and a coordinate system must be registered before any coordinates can be registered, since they must be associated with their corresponding system. Coordinates can be registered, with any chosen name, with an existing coordinate system, along with their direction or index in the coordinate system. Optionally, the coordinate can also be associated with a given grid variable. A separate call can register the global range for a coordinate on a given grid hierarchy.

Following conventions for coordinate system and coordinate names, provides a means for other thorns to use the physical properties of coordinate systems, without being tied to a particular thorn.

A registered coordinate system can be referred to by either its name or an associated integer known as a *handle*. Passing a handle instead of the name string may be necessary for calling C routines from Fortran.

Registering Coordinates and Coordinate Properties

The APIs described in this section are deprecated, and will probably be phased out fairly soon. New code should use the APIs provided by the `CoordBase` thorn instead (this lives in the `CactusBase` arrangement).

Coordinate systems and their properties can be registered at any time with the flesh. The registration utilities for thorns providing coordinates are:

`CCTK_CoordRegisterSystem`
: Assigns a coordinate system with a chosen name and dimension. For example, a 3-dimensional Cartesian coordinate system could be registered with the name `cart3d` using the call from C

```
int ierr;
int dim=3;
ierr = CCTK_CoordRegisterSystem(dim,"cart3d");
```

`CCTK_CoordRegisterData`
: Defines a coordinate in a given coordinate system, with a given direction and name, and optionally associates it to a grid variable. The directions of the coordinates range from 1 to the dimension of the coordinate system. For example, to register the grid variable `grid::y3d` to have the coordinate name `y` in the `cart3d` system

```
int ierr;
int dir=2;
ierr = CCTK_CoordRegisterData(dir,"grid::y3d","y","cart3d");
```

`CCTK_CoordRegisterRange`
: Assigns the global computational maximum and minimum for a coordinate on a grid hierarchy, that is in a `cctkGH`. At this time the maximum and minimum values have to be of type `CCTK_REAL`. For example, if the `y` coordinate for the `cart3d` system ranges between zero and one

```
CCTK_REAL lower=0;
CCTK_REAL upper=1;
int ierr;
ierr = CCTK_CoordRegisterRange(cctkGH, lower, upper, -1, "y", "cart3d");
```

Note that the API allows either the coordinate name or the direction to be used, so that the following is also valid

```
CCTK_REAL lower=0;
CCTK_REAL upper=1;
int ierr;
ierr = CCTK_CoordRegisterRange(cctkGH, lower, upper, 2, NULL, "cart3d");
```

`CCTK_CoordRegisterPhysIndex`

Implementing such things as symmetry properties for a grid leads to the need to know the details of the *physical* section of a grid. Such information is typically needed by I/O thorns. The following call illustrates how to register the indices 3 and 25 as supplying the physical range of the `y` coordinate in the `cart3d` system

```
int loweri=3;
int upperi=25;
int ierr;
ierr = CCTK_CoordRegisterPhysIndex(cctkGH, loweri, upperi, -1, "y", "cart3d");
```

Using Coordinates

The APIs described in this section are deprecated, and will probably be phased out fairly soon. New code should use the APIs provided by the `CoordBase` thorn instead (this lives in the `CactusBase` arrangement).

The utilities for thorns using coordinates are:

`CCTK_NumCoordSystems`

Returns the number of coordinate systems registered with the flesh. For example,

```
int num;
num = CCTK_NumCoordSystems();
```

`CCTK_CoordSystemName`

Provides the name of a registered coordinate system, given the integer handle (or index) for the system in the flesh's coordinate data base. Note that the handle ranges between zero and the number of coordinate systems minus one: $0 \leq \text{handle} \leq$ `CCTK_NumCoordSystems()` $- 1$. It is important to remember that the handle given to a coordinate system depends on the order in which systems are registered, and can be different from one simulation to the next.

For example, to print the names of all registered coordinate systems:

```
for (i=0; i<CCTK_NumCoordSystems(); i++)
    printf("%s ",CCTK_CoordSystemNName(i));
```

`CCTK_CoordSystemDim`

Provides the dimension of a coordinate system. For example, if the `cart3d` system was registered as having 3 dimensions, the variable `dim` below will now be set to 3,

```
int dim;
dim = CCTK_CoordSystemDim("cart3d");
```

`CCTK_CoordSystemHandle`

Provides the integer handle for a given coordinate system name. The handle describes the index for the coordinate system in the flesh coordinate database, and its value will range between zero and the number of registered systems minus one. For example, the handle for the `cart3d` coordinate system can be found using

```
int handle;
handle = CCTK_CoordSystemHandle("cart3d");
```

`CCTK_CoordSystemName`

The inverse to the previous function call. This provides the name for a given coordinate system handle. For example, to find the first coordinate system in the flesh database

```
int handle = 0;
const char *name = CCTK_CoordSystemName(handle);
```

`CCTK_CoordIndex` Provides the grid variable index for a given coordinate. Note that it is not necessary for a registered coordinate to have an associated grid variable, and if no such grid variable is found, a negative integer will be returned. For example, to find the grid variable index associated with the `y` coordinate of the `cart3d` system, either of the two following calls could be made

```
int index;
index = CCTK_CoordIndex(2,NULL,"cart3d");
```

```
int index;
index = CCTK_CoordIndex(-1,"y","cart3d");
```

`CCTK_CoordDir` Provides the direction for a given coordinate. Directions are integers ranging from one to the number of dimensions for the coordinate system. For example, to return the direction of the `y` coordinate in the `cart3d` system

```
int dir;
dir = CCTK_CoordDir("y","cart3d");
```

The return of a negative integer indicates that the coordinate direction could not be found.

`CCTK_CoordRange` Provides the global range (that is, the minimum and maximum values across the complete grid) of a coordinate on a given grid hierarchy. The minimum and maximum values must be of type `CCTK_REAL`. The coordinate can be specified either by name or by its direction. Note that this call takes the *addresses* of the minimum and maximum values. For example, the range of the `y` coordinate of the `cart3d` coordinate system can be found using

```
CCTK_REAL lower, upper;
int ierr;
ierr = CCTK_CoordRange(cctkGH, &lower, &upper, -1, "y", "cart3d");
```

or alternatively, using the direction

```
CCTK_REAL lower, upper;
int ierr;
ierr = CCTK_CoordRange(cctkGH, &lower, &upper, 2, NULL, "cart3d");
```

`CCTK_CoordLocalRange`

Provides the local range of a coordinate on a processor for a given grid hierarchy. WARNING: This utility only works for regular cartesian grids. For example, the local processor range of the `y` coordinate of the `cart3d` coordinate system can be found using

```
CCTK_REAL lower, upper;
int ierr;
ierr = CCTK_CoordLocalRange(cctkGH, &lower, &upper, -1, "y", "cart3d");
```

or alternatively, using the direction

```
CCTK_REAL lower, upper;
int ierr;
ierr = CCTK_CoordLocalRange(cctkGH, &lower, &upper, 2, NULL, "cart3d");
```

`CCTK_CoordRangePhysIndex`

For a given coordinate, provides the indices describing the *physical* range of the coordinate. A negative return value signifies that no such range was registered for the coordinate.

This index range provides a mechanism for describing grid points which should not be considered part of the simulation results (for example, grid points used for different boundary conditions). The physical range of the `y` coordinate of the `cart3d` system can be found using

```
int ilower, iupper;
int ierr;
ierr = CCTK_CoordRangePhysIndex(cctkGH,&ilower,&iupper, -1, "y", "cart3d");
```

or using the coordinate direction

```
int ilower, iupper;
int ierr;
ierr = CCTK_CoordRangePhysIndex(cctkGH,&ilower,&iupper, 2, NULL, "cart3d");
```

`CCTK_CoordSystemImplementation`

This call returns the name of the implementation which registered a coordinate system. Note that there is no guarantee that a thorn, which registered a coordinate system, is the same thorn which registers each of the coordinates in the system, although this should usually be the case.

C1.7.3 I/O

To allow flexible I/O, the flesh itself does not provide any output routines, however it provides a mechanism for thorns to register different routines as I/O methods (see Chapter C2.7). Application thorns can interact with the different I/O methods through the following function calls:

`CCTK_OutputGH (const cGH *GH)`

This call loops over all registered I/O methods, calling the routine that each method has registered for `OutputGH`. The expected behaviour of any `OutputGH` routine is to loop over all GH variables, outputting them if the I/O method contains appropriate routines (that is, not all methods will supply routines to output all different types of variables), and if the method decides it is an appropriate time to output.

`CCTK_OutputVar (const cGH *GH, const char *varname)`

Outputs a variable *varname* looping over all registered I/O methods. *varname* may have an optional I/O option string appended. The output should take place if at all possible. If output goes into a file and the appropriate file exists, the data is appended, otherwise a new file is created.

`CCTK_OutputVarAs (const cGH *GH, const char *varname, const char *alias)`

Outputs a variable *varname* looping over all registered I/O methods. *varname* may have an optional I/O option string appended. The output should take place if at all possible. If output goes into a file and the appropriate file exists, the data is appended, otherwise a new file is created. Uses *alias* as the name of the variable for the purpose of constructing a filename.

`CCTK_OutputVarByMethod (const cGH *GH, const char *varname, const char *methodname)`

Outputs a variable *varname* using the I/O method *methodname* if it is registered. *varname* may have an optional I/O option string appended. The output should take place if at all possible. If output goes into a file and the appropriate file exists, the data is appended, otherwise a new file is created.

`CCTK_OutputVarAsByMethod (const cGH *GH, const char *varname, const char *methodname, const char *alias)`

Outputs a variable *varname* using the I/O method *methodname* if it is registered. *varname* may have an optional I/O option string appended. The output should take place if at all possible. If output goes into a file and the appropriate file exists, the data is appended, otherwise a new file is created. Uses *alias* as the name of the variable for the purpose of constructing a filename.

C1.7.4 Interpolation Operators

The flesh does not provide interpolation routines by itself. Instead, it offers a general function API to thorns, for the registration and invocation of interpolation operators.

There are two different flesh APIs for interpolation, depending on whether the data arrays are Cactus grid arrays or processor-local, programming language built-in arrays, and on what assumptions are made about the topology and spacing of the grid (these descriptions are for 3D, but the generalisations to other numbers of dimensions should be obvious):

`CCTK_InterpGridArrays()`

Interpolates Cactus grid arrays, with the topology of the grid implicitly specified by a Cactus coordinate system.

This API doesn't provide an interpolation functionality itself, it only takes care of the interprocessor communication necessary when interpolating distributed grid arrays, and invokes the `CCTK_InterpLocalUniform()` API on the each processor's local component of the data.

`CCTK_InterpLocalUniform()`

Interpolates processor-local arrays with *uniformly* spaced data points, i.e. where the coordinates xyz are related to the integer array subscripts `ijk` by *linear* functions

$$x = \texttt{origin}_x + i\,\texttt{delta}_x$$
$$y = \texttt{origin}_y + j\,\texttt{delta}_y$$
$$z = \texttt{origin}_z + k\,\texttt{delta}_z$$

where the caller specifies the `origin` and `delta` values.

The flesh provides an API to register local interpolation operators:

`CCTK_InterpRegisterOpLocalUniform()`
: Register a `CCTK_InterpLocalUniform()` interpolation operator

This is described in detail in the Reference Manual.

Each local interpolation operator is registered under a character string name; at registration, the name is mapped to a unique integer handle, which may be used to refer to the operator. `CCTK_InterpHandle()` is used to get the handle corresponding to a given character string name.

C1.7.5 Reduction Operators

A reduction operation can be defined as an operation on variables distributed across multiple processor resulting in a single number. Typical reduction operations are: sum, minimum/maximum value, and boolean operations. A typical application is, for example, finding the maximum reduction from processor local error estimates, therefore, making the previous processor local error known to all processors.

The exchange of information across processors needs the functionality of a communication layer, e.g. `CactusPUGH/PUGH`. For this reason, the reduction operation itself is not part of the flesh, instead, Cactus (again) provides a registration mechanism for thorns to register routines they provide as reduction operators. The different operators are identified by their name and/or a unique number, called a *handle*.

The registration mechanism gives the advantage of a common interface while hiding the individual communication calls in the layer.

In Cactus, reduction operators can be applied to grid functions, arrays and scalars, as well as to local arrays. Note that different implementations of reduction operators may be limited in the objects they can be applied to. There is a fundamental difference between the reduction operation on grid functions and quantities as arrays.

Currently the flesh supports the new and old reduction specification. The old APIs will be deprecated in the next beta cycle in favour of the new specification.

New Reduction Specification API documentation

In the new reduction specification, there are two different flesh APIs for reduction, depending on whether the data arrays are Cactus grid arrays or processor-local, programming language built-in arrays, and on what assumptions are made about the topology and spacing of the grid (these descriptions are for 3D, but the generalisations to other numbers of dimensions should be obvious):

`CCTK_ReduceGridArrays()`
: Reduces Cactus grid arrays, with the topology of the grid implicitly specified by a Cactus coordinate system.

  This API doesn't provide a reduction functionality itself, it only takes care of the interprocessor communication necessary when reducing distributed grid arrays, and invokes the `CCTK_ReduceLocalArrays()` API on the each processor's local component of the data.

`CCTK_ReduceLocalArrays()`
: Reduces processor-local arrays with various options including *offsets*, *strides* and

masks.

The flesh provides an API to register local reduction operators:

`CCTK_RegisterLocalArrayReductionOperator()`
: Register a `CCTK_ReduceLocalArrays()` interpolation operator

This is described in detail in the Reference Manual.

Each local reduction operator is registered under a character string name; at registration, the name is mapped to a unique integer handle, which may be used to refer to the operator. `CCTK_LocalArrayReductionHandle()` is used to get the handle corresponding to a given character string name.

Old Reduction Specification API Documentation

Obtaining the reduction handle

Before calling the routine which performs the reduction operation, the handle, which identifies the operation, must be derived from its registered name.

```
int CCTK_ReductionHandle(const char *reduction_name);

integer       reduction_handle
character*(*) reduction_name
call CCTK_ReductionHandle(reduction_handle, reduction_name)


int CCTK_ReductionArrayHandle(const char *reduction_name);

integer       reduction_handle
character*(*) reduction_name
call CCTK_ReductionArrayHandle(reduction_handle, reduction_name)
```

`reduction_handle` in Fortran, the name of the variable will contain the handle value after the call. In C, this value is the function value.

`reduction_name` is the name under which the operator has been registered by the providing thorn. The only thorn in the standard Computational Toolkit release, which provides reduction operators, is `CactusPUGH/PUGHReduce`.

error checking negative handle value indicates failure to identify the correct operator.

Get a integer handle corresponding to a given reduction operator. The operator is identified by the name it was registered with. (Note that although it would appear to be far more convenient to pass the name of the reduction operator directly to the following function call to `CCTK_Reduce` this causes problems with the translation of strings from Fortran to C with variable argument lists).

The general reduction interface. The main interfaces for reduction operations are quite powerful (and hence rather complicated). To ease the use of these main interfaces, wrappers designed for specific and more restricted use are described below. If uncertain, you should use these.

```
int CCTK_Reduce( const cGH *GH,
                 int proc,
                 int operation_handle,
                 int num_out_vals,
                 int type_out_vals,
                 void *out_vals,
                 int num_in_fields,
                 ...);


call CCTK_Reduce( int returnvalue,
                  cctkGH,
                  int processor,
                  int operation_handle,
                  int num_out_vals,
                  int type_out_vals,
                  out_vals,
                  int num_in_fields,
                  ... )

int CCTK_ReduceArray(  const cGH *GH,
                       int proc,
                       int operation_handle,
                       int num_out_vals,
                       int type_out_vals,
                       void *out_vals,
                       int num_dims,
                       int num_in_arrays,
                       int type_in_arrays,
                       ... )

call CCTK_ReduceArray(int returnvalue,
                      cctkGH,
                      int processor,
                      int operation_handle,
                      int num_out_vals,
                      int type_out_arrays,
                      void out_vals,
                      int num_dims,
                      int num_in_arrays,
                      int type_in_arrays,
                      ... )
```

`int returnvalue` the return value of the operation. Negative value indicates failure to perform reduction. Zero indicates a successful operation.

`cctkGH` in Fortran, the pointer to the grid hierarchy structure. Can not be used within Fortran, but can be used from within C. Since this name is fixed, write it out as shown.

`cGH *GH` in C, it is the pointer to the grid hierarchy.

`int processor` the processor which collects the information, a negative value (-1) will distribute the data to all processors.

`int operation_handle`
the number of the reduction operation handle, needs to be found by calling `CCTK_ReductionHandle` or `CCTK_ReductionArrayHandle`.

`int num_out_vals` integer defining the number of output values.

`int type_out_arrays, type_in_arrays`
specifies the type of the gridfunction you are communicating. Use the values as specified in Section C1.9.8. Note: Do not mix data types, e.g. in Fortran, do not declare a variable as `integer` and then specify the type `CCTK_VARIABLE_INT` in the reduction command. These types need not be the same on some architectures and will conflict.

`out_vals` an array that will contain the output values.

`int num_in_fields` specifies the number of input fields.

`...` indicates a variable argument list: specify the arrays which will be reduced, the number of specified arrays must be the same as the value of the `num_in_fields` variable.

error checking a return value, other than zero, indicates failure to perform the operation.

Special reduction interfaces. The routines are designed for the purpose of reducing scalars, arrays and grid functions. They hide many of the options of the generic interface described above.

Reduction of local scalars across multiple processors. The result of the reduction operation will be on the specified processor or on all processors.

```
int CCTK_ReduceLocScalar (const cGH *GH,
                          int processor,
                          int operation_handle,
                          void *in_scalar,
                          void *out_scalar,
                          int data_type)

call CCTK_ReduceLocScalar(int returnvalue,
                          cctkGH,
                          int processor,
                          int operation_handle,
                          in_scalar,
                          out_scalar,
                          int data_type)
```

`in_scalar` the processor local variable with local value to be reduced

`out_scalar` the reduction result: a processor local variable with the global value (same on all processors), if `processor` has been set to -1. Otherwise, `processor` will hold the reduction result.

data_type	specifies the type of the gridfunction you are communicating. Use the values as specified in Section C1.9.8.

Reduction of local 1d arrays to a local arrays. This reduction is carried out element by element. The arrays need to have the same size on all processors.

```
int CCTK_ReduceLocArrayToArray1D( const cGH *GH,
                                  int processor,
                                  int operation_handle,
                                  void *in_array1d,
                                  void *out_array1d,
                                  int xsize,
                                  int data_type)

call CCTK_ReduceLocArrayToArray1D(int returnvalue
                                  cctkGH,
                                  int processor,
                                  int operation_handle,
                                  in_array1d,
                                  out_array1d,
                                  int xsize,
                                  int data_type)
```

in_array1d	one dimensional local arrays to be reduced across a processors, element by element.
out_array1d	array holding the reduction result. out_array1d[1] = Reduction(in_array[1]).
xsize	the size of the one dimensional array.

Reduction of local 2d arrays to a local 2d array. This reduction is carried out element by element. The arrays need to have the same size on all processors.

```
int CCTK_ReduceLocArrayToArray2D( const cGH *GH,
                                  int processor,
                                  int opertaion_handle,
                                  in_array_2d,
                                  out_array2d,
                                  int xsize,
                                  int ysize,
                                  int data_type)


call CCTK_ReduceLocArrayToArray2D( int returnvalue
                                   cctkGH,
                                   int processor,
                                   int operation_handle,
                                   in_array2d,
                                   out_array2d,
                                   int xsize,
```

```
                                    int ysize,
                                    int data_type)
```

in_array1d	two dimensional local arrays, to be reduced across a processors, element by element.
out_array1d	two dimensional array holding the reduction result. out_array2d[i,j]= Reduction(in_array2d[i,j]).
xsize	the size of the one dimensional array in x direction.
ysize	the size of the one dimensional array in y direction.

Reduction of local 3D arrays to a local 3D array. This reduction is carried out element by element. The arrays need to have the same size on all processors.

```
int CCTK_ReduceLocArrayToArray3D(const cGH *GH,
                                 int processor,
                                 int opertaion_handle,
                                 in_array_3d,
                                 out_array3d,
                                 int xsize,
                                 int ysize,
                                 int zsize,
                                 int data_type)

call CCTK_ReduceLocArrayToArray3D(int returnvalue
                                 cctkGH,
                                 int processor,
                                 int operation_handle,
                                 in_array3d,
                                 out_array3d,
                                 int xsize,
                                 int ysize,
                                 int zsize,
                                 int data_type)
```

in_array3d	two dimensional local arrays, to be reduced across a processors, element by element.
out_array3d	two dimensional array holding the reduction result. out_array3d[i,j,k]= Reduction(in_array3d[i,j,k]).
xsize	the size of the one dimensional array in x direction.
ysize	the size of the one dimensional array in y direction.
ysize	the size of the one dimensional array in z direction.

Some brief examples:

Reduction of a local scalars: a local error is reduced across all processors with the maximum operation. The variable `tmp` will hold the maximum of the error and is the same on all processors. This quantity can then be reassigned to `normerr`.

```
      CCTK_REAL normerr, tmp
      integer   ierr, reduction_handle

      call CCTK_ReductionArrayHandle(reduction_handle,"maximum")

      if (reduction_handle.lt.0) then
         call CCTK_WARN(CCTK_WARN_ALERT, "Cannot get reduction handle for maximum operation.")
      endif

      call CCTK_ReduceLocScalar(ierr, cctkGH, -1,
     .            reduction_handle,
     .            normerr, tmp, CCTK_VARIABLE_REAL)
      if (ierr.ne.0) then
         call CCTK_WARN(CCTK_WARN_ALERT, "Reduction of norm failed!");
      endif
      normerr = tmp
```

Reduction of a local 2D array: a two dimensional array (2×3) is reduced, reduction results (array of same size: `bla_tmp`) are seen on all processors (-1 entry as the third argument); also demonstrates some simple error checking with the `CCTKi_EXPECTOK` macro.

```
      CCTK_REAL bla(2,3),bla_tmp(2,3);
      integer   ierr, sum_handle

      call CCTK_ReductionArrayHandle(sum_handle,"sum")
      bla         =  1.0d0
      write (*,*) "BLA ",bla

      call CCTK_ReduceLocArrayToArray2D(ierr, cctkGH, -1, sum_handle,
     .     bla, bla_tmp, 2, 3, CCTK_VARIABLE_REAL)
      call CCTKi_EXPECTOK(ierr, 0, 1, "2D Reduction failed")

      bla = bla_tmp
      write (*,*) "BLA ",bla
```

Note that the memory for the returned values must be allocated before the reduction call is made.

C1.8 Completing a Thorn

C1.8.1 Commenting Source Code

Note that since most source files (see Section C1.2.4 for exceptions) pass through a C preprocessor, C style comments can be used in Fortran code. Note that C++ comments (those ones starting with "`//`"), should only be used in C++ source code.

The flesh and the Cactus thorns use the `grdoc` Code Documenting System (http://jean-luc.aei.mpg.de/Codes/grdoc/) to document source code.

C1.8.2 Providing Runtime Information

To write from thorns to standard output (i.e. the screen) at runtime, use the macro `CCTK_INFO` or `CCTK_VINFO`.

For example, from the Fortran thorn `MyThorn`,

```
call CCTK_INFO("Starting Tricky Calculation")
```

will write the line:

```
INFO (MyThorn): Starting Tricky Calculation
```

For a multiprocessor run, only runtime information from processor zero will be printed to screen by default. The standard output of other processors will usually be discarded unless the "`-r`" command line option is used (Section B3.1).

Note that the macro `CCTK_VINFO` can only be called from C, because Fortran doesn't know about variable argument lists. So, including variables in the info message using `CCTK_INFO` is currently more tricky, since you need to build the string to be output.

For example, in C you would just write

```
int myint;

CCTK_VINFO("The integer is %d", myint);
```

But in Fortran you have to do the following

```
integer       myint
character*200 message

write (message, '("The integer is ",i4)') myint
call CCTK_INFO (message)
```

Note that

- `CCTK_INFO` is a macro which expands to a call to the internal function `CCTK_Info()` and automatically includes the thorn name in function call.
- `CCTK_INFO` should be used rather than print statements, since it will give consistent behaviour on multiprocessors, and also provides a mechanism for switching the output to screen on and off, even on a thorn-by-thorn basis. (Although this is not yet implemented).

C1.8.3 Error Handling, Warnings and Code Termination

The Cactus macros `CCTK_ERROR` and `CCTK_VERROR` should be used to output error messages and abort the code.

The Cactus macros CCTK_WARN and CCTK_VWARN should be used to issue warning messages during code execution.

Along with the warning message, an integer is given to indicate the severity of the warning. Only warnings with severity levels less than, or equal to, the global Cactus warning level threshold[2] will be printed. A level 0 warning indicates the highest severity (and is guaranteed to abort the Cactus run), while larger numbers indicate less severe warnings. The global Cactus warning level threshold defaults to 1, i.e. level 1 warnings are printed, but level 2 and higher are not printed.

The severity level may actually be any integer, and a lot of existing code uses bare "magic number" integers for warning levels, but to help standardize warning levels across thorns, new code should probably use one of the following macros, defined in "cctk_WarnLevel.h" (which is #included by "cctk.h"):

```
#define CCTK_WARN_ABORT    0    /* abort the Cactus run */
#define CCTK_WARN_ALERT    1    /* the results of this run will probably */
                                /* be wrong, but this isn't quite certain, */
                                /* so we're not going to abort the run */
#define CCTK_WARN_COMPLAIN 2    /* the user should know about this, but */
                                /* the results of this run are probably ok */
#define CCTK_WARN_PICKY    3    /* this is for small problems that can */
                                /* probably be ignored, but that careful */
                                /* people may want to know about */
#define CCTK_WARN_DEBUG    4    /* these messages are probably useful */
                                /* only for debugging purposes */
```

For example, to provide a warning for a serious problem, which indicates that the results of the run are quite likely wrong, and which will be printed to the screen by default, a level CCTK_WARN_ALERT warning should be used.

The syntax from Fortran is

```
call CCTK_WARN(CCTK_WARN_ALERT, "Your warning message")

call CCTK_ERROR("Your error message")
```

and from C

```
CCTK_WARN(CCTK_WARN_ALERT, "Your warning message");

CCTK_ERROR("Your error message");
```

Note that CCTK_ERROR and CCTK_WARN are macros which expand to calls to an internal function. The macros automatically include the thorn name, the source code file name and line number in the message.[3]

[2] As discussed in Section B3.1 of this manual, the Cactus warning level threshold is set with the -W or -warning-level command-line option when running Cactus; see Section B3.1.

[3] In calling CCTK_VError() or CCTK_VWarn(), you need to provide this information yourself. Cactus provides the macro CCTK_THORNSTRING, which is the character-string name of the current thorn. In C, you can get the source file name and line number from the predefined ! preprocessor macros __FILE__ and __LINE__, respectively.

(For this reason it is important for Fortran code that capital letters are always used in order to expand the macro.)

If the flesh parameter `cctk_full_warnings` is set to true, then the source file name and line number will be printed to standard error along with the originating processor number, the thorn name and the warning message. The default is to omit the source file name and line number.

Note that the macros `CCTK_VERROR` and `CCTK_VWARN` can only be called from C, because Fortran doesn't know about variable argument lists. So including variables in the warning message using `CCTK_ERROR` or `CCTK_WARN`, is currently more tricky since you need to build the string to be output.

For example, in C you would just write

```
int    myint;
double myreal;

CCTK_VWARN(CCTK_WARN_ALERT,
           "Your warning message, including %f and %d",
           myreal, myint);
```

But in Fortran you have to do the following

```
integer       myint
real          myreal
character*200 message

write (message, '("Your warning message, including ",g12.7," and ",i8)') myreal, myint
call CCTK_WARN (CCTK_WARN_ALERT, message)
```

Beside the default methods to handle error, warning, and information messages, the flesh also implements a callback scheme to let thorn writers get information and warning messages as they are produced.[4]

For warning messages, a function with the following prototype

```
void my_warnfunc(int level,
                 int line,
                 const char *file,
                 const char *thorn,
                 const char *message,
                 void *data);
```

should be implemented, and then registered with

```
CCTK_WarnCallbackRegister(int minlevel,
                          int maxlevel,
                          void *data,
                          cctk_warnfunc my_warnfunc);
```

[4]For the moment, these functions can only be used from C.

The data pointer can be used to pass arbitrary information to the registered function, e.g. a file descriptor or a format string.

Multiple functions can be registered as above; when `CCTK_WARN` is called, all the registered functions will be called, if the warning is within the minimum and maximum levels indicated.

The basic procedure is exactly the same for information messages.

A function registered for information messages will look like

```
void my_infofunc(const char *thorn,
                 const char *message,
                 void *data);
```

while the registration function looks like

```
CCTK_InfoCallbackRegister(void *data, cctk_infofunc my_infofunc);
```

C1.8.4 Adding Documentation

Documentation is a vital part of your thorn, helping to ensure its ease of use and longevity, not only for others, but also for the thorn authors. Although any kind of independent documentation can be added to a thorn (ideally in the `doc` directory), there are two standard places for adding thorn documentation, a `README` and a `doc/documentation.tex` file for including in Thorn Guides.

`README`

The `README`, in the top level of a thorn, should contain brief and essential details about the thorn, such as the authors, any copyright details, and a synopsis of what the thorn does.

Contribution to Thorn Guide

The LaTeX file, `doc/documentation.tex`, is included in Thorn Guides built by the Cactus make system. (e.g. by `gmake` <*config*>`-ThornGuide`). Ideally this file should contain complete (and *up-to-date*) details about the thorn, exactly what is relevant is for the authors to decide, but remember that the Cactus make system automatically parses the thorn CCL files to include information about all parameters, variables and scheduling. Suggested sections include:

- **Model.** A description of the system which the thorn is modelling, including the equations, etc., which are being solved or implemented.
- **Numerical implementation.** Details about how the model is numerically implemented in the thorn.
- **Using the thorn.** Any special details needed for using the thorn, tricky parameters, particular operating systems or additional required software, interactions with other thorns and examples of use.

- **History.** Here is where you should describe why the thorn was written, any previous software or experience which was made use of, the authors of the thorn source code and documentation, how to get hold of the thorn, etc.
- **References.** A bibliography can be included, referencing papers published using or about this thorn, or additional information about the model or numerics used.

A LaTeX template for the Thorn Guide documentation can be found in the flesh distribution at

`doc/ThornGuide/template.tex`,

this file is automatically copied to the correct location in a new thorn which is created with `gmake newthorn`.

Since Cactus scripts need to parse this documentation, and since the LaTeX document should be consistent across all thorns included in a Thorn Guide, please follow the guidelines below when filling in the documentation:

- Use the Cactus Thorn Guide style file, located in the flesh distribution at `doc/latex/cactus.sty`. This should be included using a relative link, so that updates to the style file are applied to all thorns.

  ```
  \usepackage{../../../../doc/latex/cactus}
  ```

- Aside from the `date`, `author`, and `title` fields, all of the documentation to be included in a Thorn Guide should be placed between the lines

  ```
  % START CACTUS THORNGUIDE
  ```

  and

  ```
  % END CACTUS THORNGUIDE
  ```

- The command `\def` can be used to define macros, but all such definitions must lie between the `START` and `END` line. Do not redefine any standard LaTeX command
- Do not use the `\appendix` command, instead include any appendices you have as standard sections.
- We only support PDF (`.pdf`) figures. Graphics figures should be included using the `includegraphics` command (not `epsffig`), with no file extension specified. For example,

  ```
  \begin{figure}[ht]
    \begin{center}
      \includegraphics[width=6cm]{MyArrangement_MyThorn_MyFigure}
    \end{center}
    \caption{Illustration of this and that}
    \label{MyArrangement_MyThorn_MyLabel}
  \end{figure}
  ```

- All **labels**, **citations**, **references**, and **graphic images** names should conform to the following guidelines: `ARRANGEMENT_THORN_LABEL`. For instance, if you arrangement is called CactusWave, your thorn WaveToyC, and your original image blackhole.eps, you should rename your image to be `CactusWave_WaveToyC_blackhole.eps`
- References should be formatted with the standard LaTeX **bibitem** command, for example, a bibliography section should look like:

```
\begin{thebibliography}{9}
  \bibitem{MyArrangement_MyThorn_Author99}
  {J. Author, \textit{The Title of the Book, Journal, or periodical}, 1 (1999),
  1--16. \url{http://www.nowhere.com/}}
\end{thebibliography}
```

C1.8.5 Adding a Test Suite

To add a test suite to your thorn, devise a series of parameter files which use as many aspects of your thorn as possible. Make sure that the parameter files produce ASCII output to files, and that these files are in the directory `./<parameter file base name>`.

Run Cactus on each of the parameter files, and move the parameter files, and the output directories they produced, to the `test` directory in your thorn.

Document carefully any situations or architectures in which your test suite does not give the correct answers.

You can also specify options for running your testsuite by adding an optional configuration file called `test.ccl` in the `test` directory. These are simple text files and may contain comments introduced by the hash '`#`' character, which indicates that the rest of the line is a comment. If the last non-blank character of a line in a config file is a backslash '\', the following line is treated as a continuation of the current line. Options include test specific absolute and relative tolerances, thorn specific absolute and relative tolerances, the number of processors required to run, possible postprocessing required to correctly compare numbers, and file extensions. The configuration file has the form:

```
ABSTOL  <thorn_absolute_tolerance> [filename_pattern]
RELTOL  <thorn_relative_tolerance> [filename_pattern]
POSTPROC <util_program_name> [filename_pattern]
NPROCS  <thorn_nprocs>
EXTENSIONS  <extension_1 extension_2 extension_3>

TEST <test_example>
{
  ABSTOL <absolute_tol> [filename_pattern]
  RELTOL <relative_tol> [filename_pattern]
  POSTPROC <util_program_name> [filename_pattern]
  NPROCS <nprocs>
}
```

which states that when comparing files of test `test_example`, both `absolute_tol` and `relative_tol` should be used as the absolute and relative tolerances. For all other tests in the thorn, the default value of absolute and relative tolerances are set to `thorn_absolute_tolerance` and `thorn_relative_tolerance`. Both absolute and relative tolerances can be specified on a per-file bases by supplying an optional `filename_pattern` regular expression to match against a filename and the tolerance value. The specified tolerances override more general tolerances for data files whose name matches these regular expressions. Any set of characters can be used for matching as long as there are no whitespaces in the regular expression.

For example:

```
ABSTOL 1e-8 ^Psi4\.[xy]
RELTOL 1e-12 gxx
```

More specific tolerances can be specified for all the tests of a thorn or just within a test's block. It is an error if a regular expression matches more than one filename.

The `NPROCS` option specifies the number of processors required to run a given testsuite *`test_example`* or all testsuites of a thorn successfully. If no `NPROCS` option is present, the testsuite(s) is (are) assumed to run with any number of processors. The `EXTENSIONS` option adds `extension_1`, `extension_2` and `extension_3` to the list of file extensions that are compared. This list is global over all tests in a configuration.

Test specific tolerances have precedence over all tolerances, next come thorn wide tolerances, and then cactus default tolerances. Absolute and relative tolerances are independent: you can choose to use test specific absolute tolerance and thorn specific relative tolerance when running a test. For example,

```
TEST test_rad
{
  ABSTOL 1e-5
}

ABSTOL  1e-8
RELTOL  1e-12
```

would use an absolute tolerance of 10^{-5} and a relative tolerance of 10^{-12} when running `test_rad` and an absolute tolerance of 10^{-8} and a relative tolerance of 10^{-12} when running all other tests.

In addition, postprocessing can be specified. By means of this mechanism, hdf5 files can be converted into text, or ascii files whose contents are re-ordered by running on multiple processes can be compared in a consistent manner.

The postproc directive is followed by one or two parameters. The first is the name of the postproc command, which should be an executable Linux program stored in the thorn's util directory. The second is a regular expression which matches the files to be processed (if no regular expression is provided, the expressin ".*" will be used, which will match anything).

The program should receive a single argument, a file name, and from it the program should produce columns of ascii formatted floating point numbers similar to what the IOASCII routines produce. The test system will run the postprocesser on both the files in the repo and the newly generated files and compare the output.

For example:

```
TEST sol-wave-test
{
  POSTPROC sort-txy.pl asc$
}
```

In the above example, files ending in "asc" are read using the sort-txy.pl program, and the output is streamed as text through the testing mechanism.

In this example, sort-txy.pl sorts columns first by time, then x, then y value, removing (for 2D simulations) the problems with ordering of data that sometimes occurr in simulations with more than one processor. The code for this example follows:

```
#!/usr/bin/env perl
use strict;
open(FD, $ARGV[0]) or die;

my $data = {};
my %t = ();
my %x = ();
my %y = ();

while(<FD>) {
    # skip blank or comment lines
    next if(/^\s(#.*)$/);

    # Remove trailing spaces
    s/\s+$//;

    my @cols = split(/\s+/);
    if($#cols < 2) {
        print;
        next;
    }
    my ($t,$x,$y) = ($cols[0], $cols[1], $cols[2]);
    $t{$t} = 1; $x{$x} = 1; $y{$y} = 1;
    $data->{$t}->{$x}->{$y} = \@cols;
}
for my $t (sort keys %t) {
    for my $x (sort keys %x) {
        for my $y (sort keys %y) {
            print join(" ",@{$data->{$t}->{$x}->{$y}}),"\n"
                if(defined($data->{$t}->{$x}->{$y}));
        }
    }
}
```

For details on running the test suites, see Section B2.6.

Best Practices for Test Suites

When writing a test suite, there are a few things you should keep in mind:

- The test suite will be run together with many other test suites. It should, therefore, finish quickly (say, in under two minutes), and not use too much memory (so that it can run on a "normal" workstation).
- The test suite will be run automatically, often in situations where no one checks the screen output. All important output should be via grid variables that are written to files. Alternatively, if the test

suite tests some low-level infrastructure, it may just abort the simulation if it fails; that will also be detected.

- Downloading many files is slow on many systems. A test suite should normally not have more than, say, hundred output files, and normally the output files should be small, so that there are not more than a few Megabytes of output files per test suite.
- The test suite output files should always be the same. That means that they should not contain time stamps, etc. It is, therefore, best to use the option `IO::out_fileinfo="none"`.
- Norms are unfortunately quite insensitive to changes to a few grid points only, even if the changes are significant. It is necessary to output grid point values directly, not only norms.
- Try to use as few thorns as possible in a test case. For example, do not active 3D output thorns (unless you use it). The fewer thorns you use, the easier it is to run the test suite.
- It is not necessary that a test suite result is "physically correct", or that it uses parameters that ensure a stable time evolution. A test suite will usually take only a few time steps, so that a grid size of, e.g. 20^3 grid points *without* dissipation can be sufficient. Test suites cannot test whether the result of a simulation is physically feasible; they only test whether anything changed at all. Ensuring that the physics is still correct has to be handled by different mechanisms.

C1.9 Advanced Thorn Writing

C1.9.1 Using Cactus Timers

What are Timers?

Cactus provides a flexible mechanism for timing different sections of your thorns using various *clocks* which have been registered with the flesh. By default, the flesh provides two clocks that measure time in different ways (provided the underlying functionality is available on your system):

`GetTimeOfDay`	Provides "wall clock time" via the unix `gettimeofday` function.
`GetrUsage`	Provides CPU usage time via the unix `getrusage` function.

Additional clocks can be implemented by thorns and registered with the flesh (see Chapter C2.9).

To use Cactus timing, you create a *timer*, which provides time information for all the registered clocks.

You can add any number of timers to your thorn source code, providing each with a name of your choice, and then use Cactus timing functions to switch on the timers, stop or reset them, and recover timing information.

Setting the flesh parameter `cactus::cctk_timer_output = "full"` will cause some summary timing information to be printed at the end of a run. Some other thorns have their own timer printing parameters as well.

Timing Calls

Many of the timing calls come in two versions, one whose name ends with the letter `I`, and one without. The calls whose names end with the letter `I` refer to the timer or clock by index, which is a non-negative `int` value; the other calls refer to a timer by name. If a timer is created without a name, it can be referred to only by its index, otherwise, it can be referred to by name or by index.

Typically, a negative return value from a timer function indicates an error.

`CCTK_TimerCreate`, `CCTK_TimerCreateI`
: Create a timer with a given name, or with no name (respectively) and return a timer index or an error code. Negative return values indicate errors. Only one timer with a given name can exist at any given time.

`CCTK_TimerDestroy`, `CCTK_TimerDestroyI`
: Reclaim resources used by a timer.

`CCTK_TimerStart`, `CCTK_TimerStartI`
: Start the given timer, using all registered clocks.

`CCTK_TimerStop`, `CCTK_TimerStopI`
: Stop the given timer on all registered clocks.

`CCTK_TimerReset`, `CCTK_TimerResetI`
: Reset the given timer on all registered clocks.

`CCTK_TimerCreateData`, `CCTK_TimerDestroyData`
: Allocate and reclaim (respectively) resources for a `cTimerData` structure, which will be used to hold clock values.

`CCTK_Timer`, `CCTK_TimerI`
: Fill the given `cTimerData` structure with clock values as of the last call to `CCTK_TimerStop`.

`CCTK_NumTimers` Return the number of created timers

`CCTK_TimerName` Return the name of the timer for a given timer index (or `NULL` if the timer is unnamed or any other error occurs).

`CCTK_NumTimerClocks`
: Take a pointer to `cTimerData` and return the number of clocks recorded in a timer measurement

`CCTK_GetClockValue`, `CCTK_GetClockValueI`
: Given a clock referred to by name or index, respectively, and a `cTimerData` pointer, return a `cTimerVal` pointer representing the value of the clock when the timer was stopped

`CCTK_TimerClockName`
: Return the name of the clock given by the `cTimerVal` pointer argument.

`CCTK_TimerClockResolution`
: Return the floating-point value of the resolution in seconds of the clock referred to by the `cTimerVal` pointer argument. This is a lower bound for the smallest non-zero difference in values between calls of `CCTK_TimerClockSeconds`.

`CCTK_TimerClockSeconds`
: Return the floating-point value of the measurement in seconds from the `cTimerVal` pointer argument.

How to Insert Timers in your Code

The function prototypes and structure definitions are contained in the include file `cctk_Timers.h`, which is included in the standard thorn header file `cctk.h`. At the moment, the timer calls are only available from C.

The following example, which uses a timer to instrument a section of code, illustrates how timers are used by application thorns. A working example is available in the thorn `CactusTest/TestTimers`.

Creating a timer

The first action for any timer is to create it, using `CCTK_TimerCreate`. This can be performed at any time prior to use of the timer:

```
#include "cctk_Timers.h"
index = CCTK_TimerCreate("TimeMyStuff");
```

Instrumenting a section of code

Code sections are instrumented using the Start, Stop and Reset functions. These functions are applied to the chosen timer using all the registered clocks.

```
ierr = CCTK_TimerStart("TimeMyStuff");
do_procedure_to_be_timed();
ierr = CCTK_TimerStop("TimeMyStuff");
```

Accessing the timer results

After calling `CCTK_TimerStop`, you then get the time value from any of the registered clocks.

The procedure is to allocate a `cTimerData` structure, and read the information from your timer into this structure using `CCTK_Timer`, then to extract time data of the desired clock from this structure. After using the structure, you should destroy it.

```
  cTimerData *info = CCTK_TimerCreateData();
  int ierr = CCTK_Timer("TimeMyStuff",info);

  /*  You can refer to a particular clock by name.  */
  const cTimerVal   *clock = CCTK_GetClockValue( "gettimeofday", info );
  if( clock ){
    printf ("\t%s: %.3f %s\n", "gettimeofday",
                               CCTK_TimerClockSeconds( clock ), "secs" );
  }

  /*  To get results from all available clocks, refer to them by index.*/
  nclocks = CCTK_NumTimerClocks( info );

  for (i = 0; i < numclocks; i++) {
    const cTimerVal   *clock = CCTK_GetClockValueI( i, info );
    printf ("\t%s: %.3f %s\n", CCTK_TimerClockName( clock ),
```

```
                                CCTK_TimerClockSeconds( clock ), "secs" );
  }
  CCTK_TimerDestroyData (info);
```

C1.9.2 Include Files

Cactus provides a mechanism for thorns to add code to include files which can be used by any other thorn. Such include files can contain executable source code, or header/declaration information. A distinction is made between these two cases, since included executable code is protected from being run if a thorn is compiled, but not active by being wrapped by a call to `CCTK_IsThornActive`.

Any thorn which uses the include file must declare this in its `interface.ccl` with the line

```
USES INCLUDE [SOURCE|HEADER]: <file_name>
```

(If the optional `[SOURCE|HEADER]` is omitted, `HEADER` is assumed. Note that this can be dangerous, as included *source* code, which is incorrectly assumed to be *header* code, will be executed in another thorn *even if the providing thorn is inactive*. Thus, it is recommended to always include the optional `[SOURCE|HEADER]` specification.) Any thorn that wishes to add to this include file, declares in its own `interface.ccl`

```
INCLUDE [SOURCE|HEADER]: <file_to_include> in <file_name>
```

Example As an example of this in practice, for the case of Fortran code, consider a thorn A, which wants to gather terms for a calculation from any thorn that wishes to provide them. Thorn A could have the lines in its source code

```
c Get source code from other thorns
      allterms = 0d0
#include "AllSources.inc"
```

and would then add to `interface.ccl` the line

```
USES INCLUDE SOURCE: AllSources.inc
```

If thorn B wants to add terms for the calculation, it would create a file, say `Bterms.inc` with the lines

```
c Add this to AllSources.inc
      allterms = allterms + 1d0
```

and would add to its own `interface.ccl`

```
INCLUDE SOURCE: Bterms.inc in AllSources.inc
```

The final file for thorn A which is compiled, will contain the code

```
c Get source code from other thorns
      allterms = 0d0
      if (CCTK_IsThornActive("B").ne.0) then
c Add this to AllSources.inc
        allterms = allterms + 1d0
      end if
```

Any Fortran thorn routines which include source code must include the declaration `DECLARE_CCTK_FUNCTIONS`.

C1.9.3 Memory Tracing

Cactus provides a mechanism for overriding the standard C memory allocation routines (`malloc, free,` ...) with Cactus specific routines that track the amount of memory allocated, and from where, the allocation call was made. This information can be accessed by the user to provide an understanding of the memory consumption between two instances, and to track down possible memory leaks. This feature is available in C only.

Activating Memory Tracing

Memory tracing has to be activated at configure time. The standard `malloc` statements are overridden with macros (`CCTK_MALLOC`). To activate memory tracing use either

`DEBUG=all`	Enables all debug options (compiler debug flags, redefines `malloc`)
`DEBUG=memory`	Redefine `malloc` only.

The `CCTK_MALLOC` statements can also be used directly in the C code. But by employing them this way, only a fraction of the total memory consumption is traced. Also, they cannot be turned off at configure time. For example:

```
machine> gmake bigbuild DEBUG=yes

machine> gmake bigbuild-config DEBUG=memory
```

The new configuration `bigbuild` is configured with all debugging features turned on. The already existing configuration `bigbuild` is reconfigured with memory tracing only.

Using Memory Tracing

You can request Cactus to store the memory consumption at a certain instance in the program flow, and return the difference in memory allocation some time later.

`int CCTK_MemTicketRequest(void)`
: Request a ticket: save the current total memory to a database. Return an integer (ticket). Use the ticket to calculate the difference in memory allocation between the two instances in `CCTK_MemTicketCash`.

`long int CCTK_MemTicketCash(int your_ticket)`
: Cash in your ticket: return the memory difference between now and the time the ticket was requested. Tickets can be cashed in several times. See example below. This only tracks the real data memory, which is the same as in undebug mode. It does not keep track of the internal allocations done to provide the database, the motivation is that this is not allocated either if you compile undebugged.

`int CCTK_MemTicketDelete(int your_ticket)`
: Delete the memory ticket. The ticket ID will not be reused, since it's incremented with every ticket request, but the memory of the memory datastructure is deallocated.

`unsigned long int CCTK_TotalMemory(void)`
: Returns the total allocated memory (not including the tracing data structures).

`void CCTK_MemStat` Prints an info string, stating the current, past, and total memory (in bytes) allocation between two successive calls to this routine, as well as the difference.

Sample C code demonstrating the ticket handling. Two tickets are requested during `malloc` operations. The `CCTK_MALLOC` statement is used directly. They are cashed in, and the memory difference is printed. Ticket 1 is cashed twice. The tickets are deleted at the end.

```
int ticket1;
int ticket2;

/* store current memstate, ticket: t1*/
t1 = CCTK_MemTicketRequest();

/* allocate data */
hi = (int*) CCTK_MALLOC(10*sizeof(int));

/* store current memstate, ticket: t2*/
t2 = CCTK_MemTicketRequest();

/* cash ticket t1, print mem difference */
printf("NOW1a: %+d \n",CCTK_MemTicketCash(t1));

/* allocte some more data */
wo = (CCTK_REAL*)CCTK_MALLOC(10*sizeof(CCTK_REAL));

/* cash ticket t1 and t2, print mem difference */
printf("NOW1b: %+d \n",CCTK_MemTicketCash(t1));
printf("NOW2 : %+d \n",CCTK_MemTicketCash(t2));

/* delete the tickets from the database */
CCTK_MemTicketDelete(t1);
CCTK_MemTicketDelete(t2);
```

C1.9.4 Calls between Different Programming Languages

Calling C Routines from Fortran

To make the following C routine,

```
int <routine name>(<argument list>)

...
```

also callable from Fortran, a new routine must be added, which is declared using the `CCTK_FCALL` and `CCTK_FNAME` macros:

```
void CCTK_FCALL CCTK_FNAME(<routine name>)(int *ierr, <argument list>)
<rewrite routine code, or call C routine itself>
```

The convention used in Cactus, is that `<routine name>` be the same as any C routine name, and that this is mixed-case. The macros change the case and number of underscores of the routine name to match that expected by Fortran.

All arguments passed by Fortran to the routine (except strings) are pointers in C, e.g. a call from Fortran

```
CCTK_INT arg1
CCTK_REAL arg2
CCTK_REAL arg3(30,30,30)

...

call MyCRoutine(arg1,arg2,arg3)
```

should appear in C as

```
void CCTK_FCALL CCTK_FNAME(MyCRoutine)(CCTK_INT *arg1,
                                       CCTK_REAL *arg2,
                                       CCTK_REAL *arg3)
{
...
}
```

String Arguments from Fortran

Fortran passes string arguments in a special, compiler-dependent, way. To facilitate this, the CCTK provides a set of macros to enable the translation to C strings. The macros are defined in `cctk_FortranString.h`, which should be included in your C file.

String arguments *must always come last* in the argument list for these macros to be effective (some Fortran compilers automatically migrate the strings to the end, so there is no portable workaround).

The macros to use depend upon the number of string arguments–we currently support up to three. The macros are `<ONE|TWO|THREE>_FORTSTRING_ARG`. Corresponding to each of these are two macros `<ONE|TWO|THREE>_FORTSTRING_CREATE` and `<ONE|TWO|THREE>_FORTSTRING_PTR`, which take one, two, or three arguments depending on the number of strings. The latter set is only necessary if a string is to be modified. In more detail:

`<ONE|TWO|THREE>_FORTSTRING_ARG`
: Used in the argument list of the C routine to which the Fortran strings are passed.

`<ONE|TWO|THREE>_FORTSTRING_CREATE`
: Used in the declaration section of the C routine to which the Fortran strings are passed. These macros have one, two or three arguments which are the variable names you choose to use for the strings in the C routine, created by null-terminating the passed-in Fortran strings. The `CREATE` macros create new strings with the names you provide, and thus should be treated as read-only and freed after use.

`<ONE|TWO|THREE>_FORTSTRING_PTR`
: These macros, used in the declaration section of the C routine *after* the `CREATE` macro, should be used if you need to modify one of the passed-in strings. They declare and define pointers to the passed-in strings.

`cctk_strlen<1|2|3>`
: These integer variables, automatically defined by the `CREATE` macro, hold the lengths of the passed in Fortran strings.

The use of the macros is probably best explained with examples. For read-only access to the strings, only the first two macros are needed, the following example compares two strings passed in from Fortran.

```
#include <stdlib.h>
#include <string.h>
#include <cctk_FortranString.h>

int CCTK_FCALL CCTK_FNAME(CompareStrings)(TWO_FORTSTRING_ARG)
{
  int retval;

  /* Allocate and create C strings with \0 at end. */
  /* This makes variable declarations, so it must be before
     any executable statements.*/

  TWO_FORTSTRING_CREATE(arg1,arg2)

  /* Do some work with the strings */
  retval = strcmp(arg1,arg2);

  /* Important, these must be freed after use */
  free(arg1);
  free(arg2);

  return retval;
}
```

Since the null terminated strings may be copies of the strings passed from Fortran, they should be treated as read-only.

To change the data in a string passed from Fortran, you need to use the `FORTSTRING_PTR` macros, which declare and set up pointers to the strings passed from C. Note that this macro must be used *after* the `FORTSTRING_CREATE` macro. For example, the following routine copies the contents of the second string to the first string

```
#include <stdlib.h>
#include <string.h>
#include <cctk_FortranString.h>

int CCTK_FCALL CCTK_FNAME(CopyStrings)(TWO_FORTSTRING_ARG)
{
  int retval;

  /* Allocate and create C strings with \0 at end. */
  /* This makes variable declarations, so it must be before
     any executable statements. */

  TWO_FORTSTRING_CREATE(arg1,arg2)
  TWO_FORTSTRING_PTR(farg1,farg2)

  /* Do some work with the strings */

  retval = strncpy(farg1,arg2,cctk_strlen1);

  /* Important, these must be freed after use */
  free(arg1);
  free(arg2);

  return retval;
}
```

Note that in the example above, two new variables, pointers to the Fortran strings, were created. These are just pointers and *should not be freed*. The example also illustrates the automatically-created variables, e.g. `cctk_strlen1`, which hold the sizes of original Fortran strings. When writing to a string its length should never be exceeded.

Calling Fortran Routines from C

To call a utility Fortran routine from C, use

```
void CCTK_FCALL CCTK_FNAME(<Fortran routine name>)(<argument list>)
```

Note that Fortran expects all arguments (apart from strings) to be pointers, so any non-array data should be passed by address.

Currently, we have no support for calling Fortran routines which expect strings from C. However, passing routines is supported when you use function aliasing, see Section C1.9.5.

C1.9.5 Function aliasing

Like calling functions in a different language, Cactus offers a mechanism for calling a function in a different thorn where you don 't need to know which thorn is actually providing the function, nor what language the function is provided in. The idea of *function aliasing* is similar to that of thorns; the routine that calls a function should not need to know anything about it, except that the function exists.

Function aliasing is quite restrictive, because of the problems involved in inter-language calling, as seen in the previous section. Function aliasing is also comparatively inefficient, and should not be used in a part of your code where efficiency is important.

Function aliasing is language-neutral, however, the syntax is strongly based on C. In the future, the function aliasing declarations may go into a new `functions.ccl` file, and will have a format more similar to that of variable group and parameter declarations.

Using an Aliased Function

To use an aliased function you must first declare it in your `interface.ccl` file. Declare the prototype as, for example,

```
CCTK_REAL FUNCTION SumStuff(CCTK_REAL IN x, CCTK_REAL IN y)
```

and that this function will be either required in your thorn by

```
REQUIRES FUNCTION SumStuff
```

or optionally used in your thorn by

```
USES FUNCTION SumStuff
```

A prototype of this function will be available to any C routine that includes the `cctk.h` header file. In a Fortran file, the declaration of the function will be included in the `DECLARE_CCTK_FUNCTIONS` macro, which is available after the statement `#include "cctk_Functions.h"`. The keywords `IN`, `OUT`, and `INOUT` work in the same fashion as `INTENT` statements in Fortran 90. That is, the C prototype will expect an argument with intent `IN` to be a value and one with intent `OUT` or `INOUT` to be a pointer. There also exists the `ARRAY` keyword for passing arrays of any dimension. Functions which are required by some thorn (which doesn't provide it itself) are checked at startup to be provided by some other thorn.

Providing a Function

To provide an aliased function you must again add the prototype to your `interface.ccl` file. A statement containing the name of the providing function and the language it is provided in, must also be given. For example,

```
CCTK_REAL FUNCTION SumStuff(CCTK_REAL IN x, CCTK_REAL IN y)
PROVIDES FUNCTION SumStuff WITH AddItUp LANGUAGE C
```

The appropriate function must then be provided somewhere in this thorn. Multiple thorns providing the same function can be compiled into the same configuration; however, only one providing thorn may be activated at runtime, otherwise, an error message is printed and the run is aborted.

It is necessary to specify the language of the providing function; no default will be assumed.

Conventions and Restrictions

Various restrictions are necessary to make function aliasing work. These are

- The return type of any function must be either `void` or one of the Cactus data types `CCTK_INT` or `CCTK_REAL`. Standard types such as `int` are not allowed.
- The type of an argument must be one of scalar types `CCTK_INT, CCTK_REAL, CCTK_COMPLEX, CCTK_STRING, CCTK_POINTER, CCTK_FPOINTER`, or an array or pointer type `CCTK_INT ARRAY, CCTK_REAL ARRAY, CCTK_COMPLEX ARRAY, CCTK_POINTER ARRAY`.[5] The scalar types are assumed to be not modifiable. Any changes made to a scalar argument by a providing function may be silently lost, or may not; it is dependent on the language of the providing and calling function. If you wish to modify an argument, then it must have intent `OUT` or `INOUT` (and hence must be either a `CCTK_INT`, a `CCTK_REAL`, or a `CCTK_COMPLEX`, or an array of one of these types).
- The name of both, the aliased and providing function, are restricted. They must follow the standard C semantics (start with a letter, contain only letters, numbers or underscores). Additionally, they must be mixed case (that is, contain at least one uppercase and one lowercase letter). The names of the aliased and providing functions must be distinct.
- If an argument is a function pointer, then the syntax looks like

  ```
  CCTK_REAL Integrate(CCTK_REAL CCTK_FPOINTER  func(CCTK_REAL IN x), \
                      CCTK_REAL IN xmin, CCTK_REAL IN xmax)
  ```

  It is assumed that the function pointer argument has the same language as the calling function. Function pointer arguments may not be nested.
- `CCTK_STRING` arguments follow the same conventions as in the previous section. That is, they must appear at the end of the argument list, there must be at most three, and a function with a `CCTK_STRING` argument can only be provided in C, not Fortran, although they may be called from either.

Examples

- A C function is provided to add together two real numbers. The `interface.ccl` should read

  ```
  CCTK_REAL FUNCTION SumStuff(CCTK_REAL IN x, CCTK_REAL IN y)
  PROVIDES FUNCTION SumStuff WITH AddItUp LANGUAGE C
  USES FUNCTION SumStuff
  ```

[5] Unfortunately, neither `CCTK_FPOINTER ARRAY`, nor `CCTK_STRING ARRAY` will work.

- A Fortran function is provided to invert a real number. The `interface.ccl` should read

```
SUBROUTINE Invert(CCTK_REAL INOUT x)
PROVIDES FUNCTION Invert WITH FindInverse LANGUAGE Fortran
USES FUNCTION Invert
```

  Note that `SUBROUTINE` has the same meaning as `void FUNCTION`.

- A Fortran function is provided to integrate any function over an interval. The `interface.ccl` should read

```
CCTK_REAL Integrate(CCTK_REAL CCTK_FPOINTER  func(CCTK_REAL IN x), \
                    CCTK_REAL IN xmin, CCTK_REAL IN xmax)
PROVIDES FUNCTION Integrate WITH SimpsonsRule LANGUAGE Fortran
USES FUNCTION Integrate
```

Testing Aliased Functions

The calling thorn does not know if an aliased function is even provided by another thorn. Calling an aliased function that has not been provided, will lead to a level 0 warning message, stopping the code. In order to check if a function has been provided by some thorn, use the `CCTK_IsFunctionAliased` function described in the function reference section.

C1.9.6 Naming Conventions

- Thorn names must not start with the word "Cactus" (in any case).
- Arrangements will be ignored if their names start with a hash mark '`#`' or dot '`.`', or end with a tilde '`~`', `.bak` or `.BAK`.
- Thorns will be ignored if they are called doc or start with a hash mark '`#`' or dot '`.`', or end with a tilde '`~`', `.bak` or `.BAK`.
- Routine names have to be unique among all thorns.
- Names of global variables have to be unique among all thorns.

C1.9.7 General Naming Conventions

The following naming conventions are followed by the flesh and the supported Cactus arrangements. They are not compulsory, but if followed, will allow for a homogeneous code.

- Parameters: lower case (except for acronyms) with words separated by an underscore. Examples: `my_first_parameter`, `solve_PDE_equation`.
- Routine names and names of global variables: Prefixed by thorn name with an underscore, then capitalised words, with no spaces. Fortran modules should be placed into source files that have the same name as the module (see section C1.6.2 for details). Examples: `MyThorn_StartUpRoutine`, `BestSolver_InitialDataForPDE`, `BestSolver_SpectralGrid`.

C1.9.8 Data Types and Sizes

Cactus knows about the following fixed size data types:

Data Type	Size (bytes)	Variable Type	Fortran Equivalent
CCTK_BYTE	1	CCTK_VARIABLE_BYTE	integer*1
CCTK_INT1	1	CCTK_VARIABLE_INT1	integer*1
CCTK_INT2	2	CCTK_VARIABLE_INT2	integer*2
CCTK_INT4	4	CCTK_VARIABLE_INT4	integer*4
CCTK_INT8	8	CCTK_VARIABLE_INT8	integer*8
CCTK_REAL4	4	CCTK_VARIABLE_REAL4	real*4
CCTK_REAL8	8	CCTK_VARIABLE_REAL8	real*8
CCTK_REAL16	16	CCTK_VARIABLE_REAL16	real*16
CCTK_COMPLEX8	8	CCTK_VARIABLE_COMPLEX8	complex*8
CCTK_COMPLEX16	16	CCTK_VARIABLE_COMPLEX16	complex*16
CCTK_COMPLEX32	32	CCTK_VARIABLE_COMPLEX32	complex*32

The availability of these types, and the corresponding C data types, are platform-dependent. For each fixed-size data type, there exists a corresponding preprocessor macro `HAVE_<data type>`, which should be used to check whether the given CCTK data type is supported, e.g.

```
/* declare variable with extended-precision complex data type if available,
   otherwise, with default CCTK precision */
#ifdef HAVE_CCTK_COMPLEX32
CCTK_COMPLEX32 var;
#else
CCTK_COMPLEX   var;
#endif
```

In addition, Cactus provides three generic numeric data types which map onto the compilers' native data types used to represent integer, real, and complex values. The size for these generic types can be chosen at configuration time (see Section B2.1.1). This is to allow the code to be run easily at different precisions. Note that the effectiveness of running the code, at a lower or higher precision, depends crucially on all thorns being used making consistent use of the these generic data types:

Data Type	Variable Type	Configuration Option
CCTK_INT	CCTK_VARIABLE_INT	INTEGER_PRECISION
CCTK_REAL	CCTK_VARIABLE_REAL	REAL_PRECISION
CCTK_COMPLEX	CCTK_VARIABLE_COMPLEX	Same as real precision

These variable types must be used by thorn writers to declare variables in the thorn interface files, and may be used to declare variables in the thorn routines. Note that variable declarations in thorns should obviously match the definitions in the interface files where appropriate.

A set of macros, which are interpreted by the preprocessor at compile time, to signify which data size is being used, are also provided:

Data Type	#define
CCTK_INT1	CCTK_INT_PRECISION_1
CCTK_INT2	CCTK_INT_PRECISION_2
CCTK_INT4	CCTK_INT_PRECISION_4
CCTK_INT8	CCTK_INT_PRECISION_8
CCTK_REAL4	CCTK_REAL_PRECISION_4
CCTK_REAL8	CCTK_REAL_PRECISION_8
CCTK_REAL16	CCTK_REAL_PRECISION_16
CCTK_COMPLEX8	CCTK_COMPLEX_PRECISION_8
CCTK_COMPLEX16	CCTK_COMPLEX_PRECISION_16
CCTK_COMPLEX32	CCTK_COMPLEX_PRECISION_32

Cactus also provides generic data and function pointers, which can be used from either C or Fortran:

Data Type	Variable Type	C equivalent
CCTK_POINTER	CCTK_VARIABLE_POINTER	void *data_ptr
CCTK_POINTER_TO_CONST	CCTK_VARIABLE_POINTER_TO_CONST	const void *data_ptr
CCTK_FPOINTER	CCTK_VARIABLE_FPOINTER	void (*fn_ptr)(void)

Fortran Thorn Writers

Cactus provides the data types CCTK_POINTER and CCTK_POINTER_TO_CONST for use in Fortran code to declare a pointer passed from C. For example, the variable cctkGH is of the type CCTK_POINTER. The data type CCTK_STRING is, in Fortran, also an opaque type; it corresponds to a C pointer, and one has to use the function CCTK_FortranString to convert it to a Fortran string, or the CCTK_Equals to compare it to a Fortran String.

Since the data types, integer in Fortran and int in C, may be different[6], many routines that can be called from both, C and Fortran, take arguments of the type CCTK_INT. This type can be different from the type integer. Fortran does not convert routine arguments automatically, and it is, therefore, necessary to pay attention to the exact argument types that a routine expects, and to convert between integer and CCTK_INT, accordingly. Currently, most flesh functions take integer arguments, while all aliased functions take CCTK_INT arguments.

> NOTE: If you make errors in passing Fortran arguments, and if there are no interfaces ("prototypes") available for the routines that are called, then the compiler cannot detect these errors. Be careful, when you write Fortran code yourself, consider placing routines in modules, which implicitly define interfaces for all contained routines.

There are two convenient ways to convert between these types. An easy way, is to define parameters or to declare variables of the desired type, assign a value to these parameters or variables, and then pass the parameter or value. This makes for very readable code, since the name of the parameter or variable serves as additional documentation:

```
CCTK_INT, parameter : jtwo = 2
integer  :: vindex_gxx, vindex_kxx
```

[6]This is only a theoretical possibility, in practice, they have to be the same type for Cactus to work at all.

```
CCTK_INT :: syncvars(jtwo)

call CCTK_VarIndex (vindex_gxx, "ADMBase::gxx")
call CCTK_VarIndex (vindex_kxx, "ADMBase::kxx")

syncvars(1) = vindex_gxx
syncvars(2) = vindex_kxx
call CCTK_SyncGroupsI (cctkGH, jtwo, syncvars)
```

(You have probably seen the strange Fortran convention, where people introduce constants named `zero` or `two`—it is a convenient way to make sure that the constant has the correct type.)

Another possibility are explicit type conversions. They can rather easily be added to existing code:

```
! Boilerplate code to determine type kinds
integer,  parameter :: izero = 0   ! A dummy variable of type integer
CCTK_INT, parameter :: jzero = 0   ! A dummy variable of type CCTK_ITN
integer,  parameter :: ik = kind (izero)   ! The kind of "integer"
integer,  parameter :: jk = kind (jzero)   ! The kind of "CCTK_INT"

integer :: syncvars(2)

call CCTK_VarIndex (syncvars(1), "ADMBase::gxx")
call CCTK_VarIndex (syncvars(2), "ADMBase::kxx")

call CCTK_SyncGroupsI (cctkGH, int(2,jk), int(syncvars,jk))
```

Fortran distinguishes between different integer *kinds*. These kinds are what is different between `integer` and `CCTK_INT`. The expression `int(EXPR,KIND)` converts `EXPR` to an integer of kind `KIND`. Above, we use the convention that the prefix `i` denotes things having to do with `integer`, and the prefix `j` denotes `CCTK_INT`.

Note that we declare the array `syncvars` with the type that is necessary to set its values. Type conversions are only possible if variables are read, not when they are written to.

C1.10 Telling the Make system What to Do

C1.10.1 Basic Recipe

C1.10.2 Make Concepts

C1.10.3 The Four Files

C1.10.4 How your code is built

Chapter C2

Infrastructure Thorns

- Concepts and terminology (Overloading and registration of functions)
- The cGH structure — what it is and how to use it
- Extending the cGH structure
- Querying group and variable information
- Providing an I/O layer
- Providing a communication layer
- Providing a reduction operator
- Providing an interpolation operator
- Overloadable functions

C2.1 Concepts and Terminology

C2.1.1 Overloading and Registration

The flesh defines a core API which guarantees the presence of a set of functions. Although the flesh guarantees the presence of these functions, they can be provided by thorns. Thorns do this by either the *overloading* or the *registration* of functions.

Overloading

Some functions can only be provided by one thorn. The first thorn to *overload* this function succeeds, and any later attempt to overload the function fails. For each overloadable function, there is a function with a name something like `CCTK_Overload...` which is passed the function pointer.

Registration

Some functions may be provided by several thorns. The thorns *register* their function with the flesh, and when the flesh-provided function is called, the flesh calls all the registered functions.

C2.1.2 GH Extensions

A GH extension is a way to associate data with each cGH. This data should be data that is required to be associated with a particular GH by a thorn.

Each GH extension is given a unique handle.

C2.1.3 I/O Methods

An I/O method is a distinct way to output data. Each I/O method has a unique name, and the flesh-provided I/O functions operate on all registered I/O methods.

C2.2 GH Extensions

A GH extension is created by calling `CCTK_RegisterGHExtension`, with the name of the extension. This returns a unique handle that identifies the extension. (This handle can be retrieved at any time by a call to `CCTK_GHExtensionHandle`.)

Associated with a GH extension are three functions

`SetupGH` this is used to actually create the data structure holding the extension. It is called when a new cGH is created.

`InitGH` this is used to initialise the extension. It is called after the scheduler has been initialised on the cGH.

`ScheduleTraverseGH`
this is called whenever the schedule tree is due to be traversed on the GH. It should initialise the data on the cGH and the call `CCTK_ScheduleTraverse` to traverse the schedule tree.

C2.3 Overloadable and Registerable Functions in Main

Function	Default
CCTK_Initialise	
CCTK_Evolve	
CCTK_Shutdown	

C2.4 Overloadable and Registerable Functions in Comm

Function	Default
CCTK_SyncGroup	
CCTK_SyncGroupsByDirI	
CCTK_EnableGroupStorage	
CCTK_DisableGroupStorage	
CCTK_QueryMaxTimeLevels	
CCTK_EnableGroupComm	
CCTK_DisableGroupComm	
CCTK_Barrier	
CCTK_Reduce	
CCTK_Interp	
CCTK_ParallelInit	

C2.5 Overloadable and Registerable Functions in I/O

Function	Default
CCTK_OutputGH	
CCTK_OutputVarAsByMethod	

C2.6 Drivers

The flesh does not know about memory allocation for grid variables, about how to communicate data when synchronisation is called for, or about multiple patches or adaptive mesh refinement. All this is the job of a driver.

This chapter describes how to add a driver to your code.

C2.6.1 Anatomy

A driver consists of a Startup routine which creates a GH extension, registers its associated functions, and overloads the communication functions. It may optionally register interpolation, reduction, and I/O methods.

A driver may also overload the default Initialisation and Evolution routines, although a simple unigrid evolver is supplied in the flesh.

C2.6.2 Startup

A driver consists of a GH extension, and the following overloaded functions.

1. CCTK_EnableGroupStorage

2. CCTK_DisableGroupStorage
3. CCTK_QueryMaxTimeLevels
4. CCTK_ArrayGroupSizeB
5. CCTK_QueryGroupStorageB
6. CCTK_SyncGroup
7. CCTK_SyncGroupsByDirI
8. CCTK_EnableGroupComm
9. CCTK_DisableGroupComm
10. CCTK_Barrier
11. CCTK_OverloadParallelInit
12. CCTK_OverloadExit
13. CCTK_OverloadAbort
14. CCTK_OverloadMyProc
15. CCTK_OverloadnProcs

The overloadable function CCTK_SyncGroup is deprecated, a driver should instead provide a routine to overload the more general function CCTK_SyncGroupsByDirI.

C2.6.3 The GH Extension

The GH extension is where the driver stores all its grid-dependent information. This is stuff like any data associated with a grid variable (e.g. storage and communication state), how many grids if it is AMR, ... It is very difficult to describe in general, but one simple example might be

```
struct SimpleExtension
{
  /* The data associated with each variable */
  /* data[var][timelevel][ijk]                */
  void ***data
} ;
```

with a SetupGH routine like

```
struct SimpleExtension *SimpleSetupGH(tFleshConfig *config, int conv_level, cGH *GH)
{
   struct SimpleExtension *extension;
```

```
   extension = NULL;

   if(conv_level < max_conv_level)
   {
      /* Create the extension */
      extension = malloc(sizeof(struct SimpleExtension));

      /* Allocate data for all the variables */
      extension->data = malloc(num_vars*sizeof(void**));

      for(var = 0 ; var < num_vars; var++)
      {
        /* Allocate the memory for the time levels */
        extension->data[var] = malloc(num_var_time_levels*sizeof(void *));

        for(time_level = 0; time_level < num_var_time_level; time_level++)
        {
          /* Initialise the data to NULL */
          extension->data[var][time_level] = NULL;
        }
      }
    }

   return extension;
}
```

Basically, what this example is doing is preparing a data array for use. The function can query the flesh for information on every variable. Note that scalars should always have memory actually assigned to them.

An `InitGH` function isn't strictly necessary, and in this case, it could just be a dummy function.

The `ScheduleTraverseGH` function needs to fill out the cGH data, and then call `CCTK_ScheduleTraverse` to have the functions scheduled at that point executed on the grid

```
int SimpleScheduleTraverseGH(cGH *GH, const char *where)
{
  int retcode;
  int  var;
  int  gtype;
  int  ntimelevels;
  int  level;
  int  idir;

  extension = (struct SimpleExtension *)GH->extensions[SimpleExtension];

  for (idir=0;idir<GH->cctk_dim;idir++)
  {
    GH->cctk_levfac[idir] = 1;
```

```
    GH->cctk_nghostzones[idir] = extension->nghostzones[idir];
    GH->cctk_lsh[idir]         = extension->lnsize[idir];
    GH->cctk_gsh[idir]         = extension->nsize[idir];
    GH->cctk_bbox[2*idir]      = extension->lb[extension->myproc][idir] == 0;
    GH->cctk_bbox[2*idir+1]    = extension->ub[extension->myproc][idir]
                               == extension->nsize[idir]-1;
    GH->cctk_lbnd[idir]        = extension->lb[extension->myproc][idir];
    GH->cctk_ubnd[idir]        = extension->ub[extension->myproc][idir];
#ifdef CCTK_HAVE_CGH_TILE
    GH->cctk_tile_min[idir]    = extension->tmin[extension->myproc][idir];
    GH->cctk_tile_max[idir]    = extension->tmax[extension->myproc][idir];
#endif

  }

  for(var = 0; var < extension->nvariables; var++)
  {
    gtype = CCTK_GroupTypeFromVarI(var);
    ntimelevels = CCTK_MaxTimeLevelsVI(var);

    for(level = 0; level < ntimelevels; level++)
    {
      switch(gtype)
      {
        case CCTK_SCALAR :
          GH->data[var][level] = extension->variables[var][level];
          break;
        case CCTK_GF     :
          GH->data[var][level] =
            ((pGF ***)(extension->variables))[var][level]->data;
          break;
        case CCTK_ARRAY :
          GH->data[var][level] =
            ((pGA ***)(extension->variables))[var][level]->data;
          break;
        default:
          CCTK_WARN(CCTK_WARN_ALERT,"Unknown group type in SimpleScheduleTraverse");
      }
    }
  }

  retcode = CCTK_ScheduleTraverse(where, GH, NULL);

  return retcode;

}
```

The third argument to `CCTK_ScheduleTraverse` is actually a function which will be called by the scheduler when it wants to call a function scheduled by a thorn. This function is given some information about the function to call, and is an alternative place where the cGH can be setup.

This function is optional, but a simple implementation might be

```
int SimpleCallFunction(void *function,
                       cFunctionData *fdata,
                       void *data)
{
  void (*standardfunc)(void *);

  int (*noargsfunc)(void);

  switch(fdata->type)
  {
    case FunctionNoArgs:
      noargsfunc = (int (*)(void))function;
      noargsfunc();
      break;
    case FunctionStandard:
      switch(fdata->language)
      {
        case LangC:
          standardfunc = (void (*)(void *))function;
          standardfunc(data);
          break;
        case LangFortran:
          fdata->FortranCaller(data, function);
          break;
        default :
          CCTK_WARN(CCTK_WARN_ALERT, "Unknown language.");
      }
      break;
    default :
      CCTK_WARN(CCTK_WARN_ALERT, "Unknown function type.");
  }

  /* Return 0, meaning didn't synchronise */
  return 0;
}
```

The return code of the function signifies whether or not the function synchronised the groups in this functions synchronisation list of not.

The flesh will synchronise them if the function returns false.

Providing this function is probably the easiest way to do multi-patch or AMR drivers.

C2.6.4 Memory Functions

These consist of

1. CCTK_EnableGroupStorage
2. CCTK_DisableGroupStorage
3. CCTK_QueryMaxTimeLevels
4. CCTK_QueryGroupStorageB
5. CCTK_ArrayGroupSizeB

En/Disable Group Storage

These are responsible for switching the memory for all variables in a group on or off. They should return the former state, e.g. if the group already has storage assigned, they should return 1.

In our simple example above, the enabling routine would look something like

```
int SimpleEnableGroupStorage(cGH *GH, const char *groupname)
{

  extension = (struct SimpleExtension *)GH->extensions[SimpleExtension];

  if(extension->data[first][0][0] == NULL)
  {
    for(var = first; var <= last; var++)
    {
      allocate memory for all time levels;
    }
    retcode = 0;
  }
  else
  {
    retcode = 1;
  }

  return retcode;
}
```

The disable function is basically the reverse of the enable one.

The `CCTK_QueryMaxTimeLevels` function returns the maximum number of timelevels ever activated for a given group ie. the size of the `data` member of `cGH`.

The `QueryGroupStorage` function basically returns true or false if there is storage for the group, and the `ArrayGroupSize` returns the size of the grid function or array group in a particular direction.

En/Disable Group Comm

These are the communication analogues to the storage functions. Basically, they flag that communication is to be done on that group or not, and may initialise data structures for the communication.

C2.7 I/O Methods

The flesh by itself does not provide output for grid variables or other data. Instead, it provides a mechanism for thorns to register their own routines as I/O methods, and to invoke these I/O methods by either the flesh scheduler or by other thorn routines.

This chapter explains how to implement your own I/O methods and register them with the flesh.

C2.7.1 I/O Method Registration

All I/O methods have to be registered with the flesh before they can be used.

The flesh I/O registration API provides a routine `CCTK_RegisterIOMethod()` to create a handle for a new I/O method which is identified by its name (this name must be unique for all I/O methods). With such a handle, a thorn can then register a set of functions (using the `CCTK_RegisterIOMethod*()` routines from the flesh) for doing periodic, triggered, and/or unconditional output.

The following example shows how a thorn would register an I/O method, `SimpleIO`, with routines to provide all these different types of output.

```
void SimpleIO_Startup (void)
{
  int handle = CCTK_RegisterIOMethod ("SimpleIO");
  if (handle >= 0)
  {
    CCTK_RegisterIOMethodOutputGH (handle, SimpleIO_OutputGH);

    CCTK_RegisterIOMethodTimeToOutput (handle, SimpleIO_TimeToOutput);
    CCTK_RegisterIOMethodTriggerOutput (handle, SimpleIO_TriggerOutput);

    CCTK_RegisterIOMethodOutputVarAs (handle, SimpleIO_OutputVarAs);
  }
}
```

C2.7.2 Periodic Output of Grid Variables

The flesh scheduler will automatically call `CCTK_OutputGH()` at every iteration after the `CCTK_ANALYSIS` time bin. This function loops over all I/O methods and calls their routines registered as `OutputGH()` (see also Section C1.2.3).

```
int SimpleIO_OutputGH (const cGH *GH);
```

The `OutputGH()` routine itself should loop over all variables living on the `GH` grid hierarchy, and do all necessary output if requested (this is usually determined by I/O parameter settings).

As its return code, it should pass back the number of variables which were output at the current iteration, or zero if no output was done by this I/O method.

C2.7.3 Triggered Output of Grid Variables

Besides the periodic output at every so many iterations using `OutputGH()`, analysis and output of grid variables can also be done via triggers. For this, a `TimeToOutput()` routine is registered with an I/O method. This routine will be called by the flesh scheduler at every iteration before `CCTK_ANALYSIS` with the triggering variable(s) as defined in the schedule block for all `CCTK_ANALYSIS` routines (see Section C1.5.4).

If the `TimeToOutput()` routine decides that it is now time to do output, the flesh scheduler will invoke the corresponding analysis routine and also request output of the triggering variable(s) using `TriggerOutput()`.

```
int SimpleIO_TimeToOutput (const cGH *GH, int varindex);
int SimpleIO_TriggerOutput (const cGH *GH, int varindex);
```

Both routines get passed the index of a possible triggering grid variable.

`TimeToOutput()` should return a non-zero value if analysis and output for *`varindex`* should take place at the current iteration, and zero otherwise.

`TriggerOutput()` should return zero for successful output of variable *`varindex`*, and a negative value in case of an error.

C2.7.4 Unconditional Output of Grid Variables

An I/O method's `OutputVarAs()` routine is supposed to do output for a specific grid variable if ever possible. It will be invoked by the flesh I/O API routines `CCTK_OutputVar*()` for unconditional, non-triggered output of grid variables (see also Section C1.7.3).

A function registered as an `OutputVarAs()` routine has the following prototype:

```
int SimpleIO_OutputVarAs (const cGH *GH, const char *varname, const char *alias);
```

The variable to output, *`varname`*, is given by its full name. The full name may have appended an optional I/O options string enclosed in curly braces (with no space between the full name and the opening curly brace). In addition to that, an *`alias`* string can be passed which then serves the purpose of constructing a unique name for the output file.

The `OutputVarAs()` routine should return zero if output for *`varname`* was done successfully, or a negative error code otherwise.

C2.8 Checkpointing/Recovery Methods

Like for I/O methods, checkpointing/recovery functionality must be implemented by thorns. The flesh only provides specific time bins at which the scheduler will call thorns' routines, in order to perform checkpointing and/or recovery.

This chapter explains how to implement checkpointing and recovery methods in your thorn. For further information, see the documentation for thorn `CactusBase/IOUtil`.

C2.8.1 Checkpointing Invocation

Thorns register their checkpointing routines at `CCTK_CPINITIAL` and/or `CCTK_CHECKPOINT` and/or `CCTK_TERMINATE`. These time bins are scheduled right after all initial data has been set up, after every evolution timestep, and after the last time step of a simulation, respectively. (See Section C1.2.3 for a description of all timebins). Depending on parameter settings, the checkpoint routines decide whether to write an initial data checkpoint, and when to write an evolution checkpoint.

Each checkpoint routine should save all information to persistent storage, which is necessary to restart the simulation at a later time from exactly the same state. Such information would include

- the current settings of all parameters
- the contents of all grid variables which have global storage assigned and are not tagged with `checkpoint="no"` (see also Section D2.2.4 on page D10 for a list of possible tags)
  Note that grid variables should be synced before writing them to disk.
- other relevant information such as the current iteration number and physical time, the number of processors, etc.

C2.8.2 Recovery Invocation

Recovering from a checkpoint is a two-phase operation for which the flesh provides distinct timebins for recovery routines to be scheduled at:

`CCTK_RECOVER_PARAMETERS`
: This time bin is executed before `CCTK_STARTUP`, in which the parameter file is parsed. From these parameter settings, the recovery routines should decide whether recovery was requested, and if so, restore all parameters from the checkpoint file and overwrite those which aren't steerable.
  The flesh loops over all registered recovery routines to find out whether recovery was requested. Each recovery routine should, therefore, return a positive integer value for successful parameter recovery (causing a shortcut of the loop over all registered recovery routines), zero for no recovery, or a negative value to flag an error. If recovery was requested, but no routine could successfully recover parameters, the flesh will abort the run with an error message. If no routine recovered any parameters, i.e. if all parameter recovery routines returned zero, then this indicates that this run is not a recovery run.
  If parameter recovery was performed successfully, the scheduler will set the `recovered`

flag which—in combination with the setting of the `Cactus::recovery_mode` parameter—decides whether any thorn routine scheduled for `CCTK_INITIAL` and `CCTK_POSTINITIAL` will be called. The default is to not execute these initial time bins during recovery, because the initial data will be set up from the checkpoint file during the following `CCTK_RECOVER_VARIABLES` time bin.

`CCTK_RECOVER_VARIABLES`
: Recovery routines scheduled for this time bin are responsible for restoring the contents of all grid variables with storage assigned from the checkpoint.
Depending on the setting of `Cactus::recovery_mode`, they should also decide how to treat errors in recovering individual grid variables. Strict recovery (which is the default) requires all variables to be restored successfully (and will stop the code if not), whereas a relaxed mode could, e.g. allow for grid variables, which are not found in the checkpoint file, to be gracefully ignored during recovery.

C2.9 Clocks for Timing

To add a Cactus clock, you need to write several functions to provide the timer functionality (see Section C1.9.1), and then register these functions with the flesh as a named clock.

The function pointers are placed in function pointer fields of a `cClockFuncs` structure. The fields of this structure are are:

```
create          void *(*create)(int)
destroy         void (*destroy)(int, void *)
start           void (*start)(int, void *)
stop            void (*stop)(int, void *)
reset           void (*reset)(int, void *)
get             void (*get)(int, void *, cTimerVal *)
set             void (*set)(int, void *, cTimerVal *)
n_vals          int
```

The first `int` argument of the functions may be used in any way you see fit.

The `n_vals` field holds the number of elements in the `vals` array field of the `cTimerVal` structure used by your timer (usually 1).

The return value of the `create` function will be a pointer to a new structure representing your clock.

The second `void*` argument of all the other functions will be the pointer returned from the `create` function.

The `get` and `set` functions should write to and read from (respectively) a structure pointed to by the `cTimerVal*` argument:

```
typedef enum {val_none, val_int, val_long, val_double} cTimerValType;

typedef struct
{
  cTimerValType type;
  const char *heading;
  const char *units;
  union
 {
    int        i;
    long int   l;
    double     d;
  } val;
  double seconds;
  double resolution;
} cTimerVal;
```

The `heading` field is the name of the clock, the `units` field holds a string describing the type held in the `val` field, and the `seconds` field is the time elapsed in seconds. The `resolution` field is the smallest non-zero difference in values of two calls to the timer, in seconds. For example, it could reflect that the clock saves the time value internally as an integer value representing milliseconds.

To name and register the clock with the flesh, call the function

`CCTK_ClockRegister( "`*`my_clock_name`*`", &`*`clock_func`*` ).`

Part D

Appendices

Chapter D1

Glossary

alias function	See *function aliasing*.
AMR	*Automatic Mesh Refinement*
analysis	
API	*Applications Programming Interface*, the interface provided by some software component to programmers who use the component. An API usually consists of subroutine/function calls, but may also include structure definitions and definition of constant values. The Cactus Reference Manual documents most of the Cactus flesh APIs.
arrangement	A collection of thorns, stored in a subdirectory of the Cactus `arrangements` directory. See Section C1.1.2.
autoconf	A GNU program which builds a configuration script which can be used to make a Makefile.
boundary zone	A boundary zone is a set of points at the edge of a grid, interpreted as the boundary of the physical problem, and which contains boundary data, e.g. Dirichlet conditions or von Neumann conditions. (See also *symmetry zone*, *ghost zone*.)
Cactus	Distinctive and unusual plant, which is adapted to extremely arid and hot environments, showing a wide range of anatomical and physiological features which conserve water. Cacti stems have expanded into green succulent structures containing the chlorophyll necessary for life and growth, while the leaves have become the spines for which cacti are so well known.[1]
CCTK	*Cactus Computational Tool Kit* (The Cactus flesh and computational thorns).
CCL	The *Cactus Configuration Language*, this is the language that the thorn configuration files are written in. See Section D2.
configuration	The combination of a set of thorns, and all the Cactus configure options which affect what binary will be produced when compiling Cactus. For example, the choice of compilers (Cactus `CC`, `CXX`, `CUCC`, and `F90` configure options) and the compiler

[1] http://en.wikipedia.org/wiki/Cactus

optimization settings (`OPTIMISE`/`OPTIMIZE` and `*_OPTIMISE_FLAGS` configure options) are part of a configuration (these flags change what binary is produced), but the Cactus `VERBOSE` and `WARN` configure options aren't part of a configuration (they don't change what binary will be produced). See Section A1.2.1.

checkout Get a copy of source code from git. See Section A1.1.

checkpoint Save the entire state of a Cactus run to a file, so that the run can be restarted at a later time. See Sections A3, C2.8.

computational grid A discrete finite set of spatial points in $\Re^n$ (typically, $1 \leq n \leq 3$). Historically, Cactus has required these points to be uniformly spaced (uniformly spaced grid), but now, Cactus supports non-uniform spacings (non-uniformly spaced grid), and mesh refinement.

The grid consists of the physical domain and the boundary and symmetry points.

See *grid functions* for the typical use of grid points.

convergence Important, but often neglected.

CST The *Cactus Specification Tool*, which is the set of Perl scripts which parse the thorns' `.ccl` files, and generates the code that binds the thorn source files with the flesh.

git *git* is the favoured code distribution system for Cactus. See Section A1.1.

domain decomposition The technique of breaking up a large computational problem into parts that are easier to solve. In Cactus, it refers especially to a decomposition wherein the parts are solved in parallel on separate computer processors.

driver A special kind of thorn which creates and handles grid hierarchies and grid variables. Drivers are responsible for memory management for grid variables, and for all parallel operations, in response to requests from the scheduler. See Section C1.6.3.

evolution An iteration interpreted as a step through time. Also, a particular Cactus schedule bin for executing routines when evolution occurs.

flesh The Cactus routines which hold the thorns together, allowing them to communicate and scheduling things to happen with them. This is what you get if you check out Cactus from our git repository.

friend Interfaces that are *friends*, share their collective set of protected grid variables. See Section D2.2 C1.2.3.

function aliasing The process of referring to a function to be provided by an interface independently of which thorn actually contains the function, or what language the function is written in. The function is called an *alias function*. See Section C1.9.5, D2.2.3.

GA Shorthand for a *grid array*.

GF Shorthand for a *grid function*.

gmake GNU version of the `make` utility.

ghost zone	A set of points added for parallelisation purposes to a block of a grid lying on one processor, corresponding to points on the boundary of an adjoining block of the grid lying on another processor. Points from the boundary of the one block are copied (via an inter-processor communication mechanism) during synchronisation to the corresponding ghost zone of the other block, and vice versa. In single processor runs there are no ghost zones. Contrast with symmetry or boundary zones. See Section C1.3.5.
grid	Short for *computational grid*.
grid array	A *grid variable* whose global size need not be that of the computational grid; instead, the size is declared explicitly in an `interface.ccl` file.
grid function	A *grid variable* whose global size is the size of the computational grid. (See also *local array*.) From another perspective, *grid functions* are functions (of any of the Cactus data types (see Section C1.9.8) defined on the domain of grid points. Typically, grid functions are used to discretely approximate functions defined on the domain $\Re^n$, with *finite differencing* used to approximate partial derivatives.
grid hierarchy	A *computational grid*, and the *grid variables* associated with it.
grid point	A spatial point in the *computational grid*.
grid scalar	A *grid variable* with index zero, i.e. just a number on each processor.
grid variable	A variable which is passed through the flesh interface, either between thorns or between routines of the same thorn. This implies the variable is related to the computational grid, as opposed to being an internal variable of the thorn or one of its routines. *grid scalar*, *grid function*, and *grid array* are all examples of *grid variables*. See Sections C1.3.2, D2.2.4
GNATS	The GNU program we use for reporting and tracking bugs, comments and suggestions.
GNU	*GNU's Not Unix*: a freely-distributable code project. See `http://www.gnu.org/`.
GV	Shorthand for *grid variable*.
handle	A signed integer value >= 0 passed by many Cactus routines and used to represent a dynamic data or code object.
HDF5	*Hierarchical Data Format* version 5, an API, subroutine library, and file format for storing structured data. An HDF5 file can store both data (for example, Cactus grid variables), and meta data (data describing the other data, for example, Cactus coordinate systems). See Section B2.2, also `https://www.hdfgroup.org/HDF5/`.
implementation	Defines the interface that a thorn presents to the rest of a Cactus program. See Section C1.1.3.
inherit	A thorn that *inherits* from another implementation can access all the other implementation's public variables. See Section D2.2, C1.2.3.
interface	
interpolation	Given a set of grid variables and interpolation points (points in the grid coordinate space, which are typically distinct from the grid points), interpolation is the act of producing values for the grid variables at each interpolation point over the entire grid hierarchy. (Contrast with *local interpolation*.)

local array An array that is declared in thorn code, but not declared in the thorn's `interface.ccl`, as opposed to a *grid array*.

local interpolation Given a set of grid variables and interpolation points (points in the grid coordinate space ,which are typically distinct from the grid points), interpolation is the act of producing values for the grid variables at each interpolation point on a particular grid. (Contrast with *interpolation*.)

Makefile The default input file for `make` (or `gmake`). Includes rules for building targets.

make A system for building software. It uses rules involving dependencies of one part of software on another, and information of what has changed since the last build, to determine what parts need to be built.

MPI *Message Passing Interface*, an API and software library for sending messages between processors in a multiprocessor system. See Section B2.2.

multi-patch

mutual recursion See *recursion, mutual*.

NUL character The C programming language uses a "NUL character" to terminate character strings. A NUL character has the integer value zero, but it's useful to write it as `'\0'`, to emphasize to human readers that this has type `char` rather than `int`.

null pointer, NULL pointer
C defines a "null pointer", often (slightly incorrectly) called a "NULL pointer", which is guaranteed not to point to any object. You get a null pointer by converting the integer constant 0 to a pointer type, e.g. `int* ptr = 0;`.[2]

Many programmers prefer to use the predefined macro `NULL` (defined in `<stdlib.h>`, `<stdio.h>`, and possibly other system header files) to create null pointers, e.g. `int* ptr = NULL;`, to emphasize to human readers that this is a null *pointer* rather than "just" the integer zero.

Note that it is explicitly *not* defined whether a null pointer is represented by a bit pattern of all zero bits—this varies from system to system, and there are real-world systems where null pointers are, in fact, *not* represented this way.

For further information, see the section "Null pointers" in the (excellent) `comp.lang.c FAQ`, available online at `http://www.eskimo.com/~scs/C-faq/top.html`.

parallelisation The process of utilising multiple computer processors to work on different parts of a computational problem at the same time, in order to obtain a solution of the problem more quickly. Cactus achieves parallelisation by means of *domain decomposition*.

parameter A variable that controls the run time behaviour of the Cactus executable. Parameters have default values which can be set in a *parameter file*. (See Chapter C1.4). The flesh has parameters; thorn parameters are made available to the rest of Cactus by describing them in the thorn's `param.ccl` file (See Appendix D2.3).

parameter file (Also called *par file*.) A text file used as the input of a Cactus program, specifying initial values of thorn parameters. See Section B3.2.

[2] Note that if you have an expression which has the value zero, but which isn't an integer constant, converting this to a pointer type is *not* guaranteed to give a NULL pointer, e.g.:

```
int i = 0;
int* ptr = i; /* ptr is NOT guaranteed to be a NULL pointer!  */
```

processor topology

PUGH — The default driver thorn for Cactus which uses MPI. See Section B1.1.

PVM — *Parallel Virtual Machine*, provides interprocessor communication. See Section B1.1.

recursion, mutual — See *mutual recursion*.

reduction — Given a set of grid variables on a computational grid, *reduction* is the process of producing values for the variables on a proper subset of points from the grid.

scheduler — The part of the Cactus flesh that determines the order and circumstances in which to execute Cactus routines. Thorn functions and schedule groups are registered with the flesh via the thorn's `schedule.ccl` file to be executed in a certain schedule bin, before or after another function or group executes, and so forth. See section D2.4 C1.5,

schedule bin — One of a set of special timebins pre-defined by Cactus. See Section D4 for a list.

schedule group — A timebin defined by a thorn, in its `schedule.ccl` file (see Appendix D2.4). Each schedule group must be defined to occur in a Cactus schedule bin or another schedule group. See Chapter C1.5, C1.5.1.

shares — An implementation may *share* restricted parameters with another implementation, which means the other implementation can get the parameter values, and if the parameters are steerable, it can change them. See Section D2.3 C1.2.3.

steerable parameter — A parameter which can be changed at any time after the program has been initialised. See Section C1.4.3.

symmetry operation — A grid operation that is a manifestation of a geometrical symmetry, especially rotation or reflection.

symmetry zone — A set of points laying at the edge of the computational grid and containing data obtained by some symmetry operation from another part of the same grid. (Contrast with *boundary zone*, *ghost zone*.)

synchronisation — The process of copying information from the outer part of a computational interior on one processor to the corresponding ghost zone (see) on another processor. Also refers to a special Cactus timebin corresponding to the occurrence of this process. See Section C1.3.5.

TAGS — See Section D7.

target — A *make target* is the name of a set of rules for `make` (or `gmake`). When the target is included in the command line for `make`, the rules are executed, usually to build some software.

test suite — See Sections B2.6, C1.8.5.

thorn — A collection of subroutines defining a Cactus interface. See Chapters C1.1, C1.2.

ThornList — A file used by the Cactus CST to determine which thorns to compile into a Cactus executable (see Section B2.4.1, B2.4.2). Can also be used to determine which thorns to check out from git. (see Section A1.1). A ThornList for each Cactus configuration lies in the configuration subdirectory of the Cactus `configs` directory.

time bin — A time interval in the duration of a Cactus run wherein the flesh runs specified routines. See *scheduler*, *schedule bin*.

time level

timer — A Cactus API for reporting time. See Section C1.9.1.

trigger

unigrid

wrapper

Chapter D2

Configuration File Syntax

D2.1 General Concepts

Each thorn is configured by three compulsory and one optional files in the top level thorn directory:

- `interface.ccl`
- `param.ccl`
- `schedule.ccl`
- `configuration.ccl` (optional)

These files are written in the *Cactus Configuration Language* which is case insensitive.

D2.2 interface.ccl

The interface configuration file consists of:

- A header block giving details of the thorn's relationship with other thorns.
- A block detailing which include files are used from other thorns, and which include files are provided by this thorn.
- Blocks detailing aliased functions provided or used by this thorn.
- A series of blocks listing the thorn's global variables.

(For a more extensive discussion of Cactus variables, see Chapter C1.3.)

D2.2.1 Header Block

The header block has the form:

```
implements: <implementation>
inherits: <implementation>, <implementation>
friend: <implementation>, <implementation>
```

where

- The implementation name must be unique among all thorns, except between thorns which have the same public and protected variables and global and restricted parameters.
- Inheriting from another implementation makes all that implementation's public variables available to your thorn. At least one thorn providing any inherited implementation must be present at compile time. A thorn cannot inherit from itself. Inheritance is transitive (if A inherits from B, and B inherits from C, then A also implicitly inherits from C), but not commutative.
- Being a friend of another implementation makes all that implementation's protected variables available to your thorn. At least one thorn providing an implementation for each friend must be present at compile time. A thorn cannot be its own friend. Friendship is associative, commutative and transitive (i.e. if A is a friend of B, and B is a friend of C, then A is implicitly a friend of C).

D2.2.2 Include Files

The include file section has the form:

```
USES INCLUDE [SOURCE|HEADER]: <file_name>
INCLUDE[S] [SOURCE|HEADER]: <file_to_include> in <file_name>
```

The former is used when a thorn wishes to use an include file from another thorn. The latter indicates that this thorn adds the code in `<file_to_include>` to the include file `<file_name>`. If the include file is described as `SOURCE`, the included code is only executed if the providing thorn is active. Both default to `HEADER`.

D2.2.3 Function Aliasing

If any aliased function is to be used or provided by the thorn, then the prototype must be declared with the form:

```
<return_type> FUNCTION <alias>(<arg1_type> <intent1> [ARRAY] <arg1>, ...)
```

The `<return_type>` must be either `void`, `CCTK_INT`, `CCTK_REAL`, `CCTK_COMPLEX`, `CCTK_POINTER`, or `CCTK_POINTER_TO_CONST`. The keyword `SUBROUTINE` is equivalent to `void FUNCTION`. The name of the aliased function `<alias>` must contain at least one uppercase and one lowercase letter and follow the C standard for function names. The type of each argument, `<arg*_type>`, must be either `CCTK_INT`,

CCTK_REAL, CCTK_COMPLEX, CCTK_POINTER, CCTK_POINTER_TO_CONST, or STRING. All string arguments must be the last arguments in the list. The intent of each argument, <*intent**>, must be either IN, OUT, or INOUT. An argument may only be modified if it is declared to have intent OUT or INOUT. If the argument is an array then the prefix ARRAY must also be given.

If the argument <*arg**> is a function pointer, then the argument itself (which will preceded by the return type) should be

```
CCTK_FPOINTER <function_arg1>(<arg1_type> <intent1> <arg1>, ...)
```

Function pointers may not be nested.

If an aliased function is to be required, then the block

```
REQUIRES FUNCTION <alias>
```

is required.

If an aliased function is to be (optionally) used, then the block

```
USES FUNCTION <alias>
```

is required.

If a function is provided, then the block

```
PROVIDES FUNCTION <alias> WITH <provider> LANGUAGE <providing_language>
```

is required. As with the alias name, <*provider*> must contain at least one uppercase and one lowercase letter, and follow the C standard for function names. Currently, the only supported values of <*providing_language*> are C and Fortran.

D2.2.4 Variable Blocks

The thorn's variables are collected into groups. This is not only for convenience, but for collecting like variables together. Storage assignment, communication assignment, and ghostzone synchronization take place for groups only.

The thorn's variables are defined by:

```
[<access>:]

<data_type> <group_name>[[<number>]] [TYPE=<group_type>] [DIM=<dim>]
[TIMELEVELS=<num>]
[SIZE=<size in each direction>] [DISTRIB=<distribution_type>]
[GHOSTSIZE=<ghostsize>]
[TAGS=<string>]  ["<group_description>"]
```

```
[{
 [ <variable_name>[,]<variable_name>
   <variable_name> ]
} ["<group_description>"] ]
```

(The options `TYPE`, `DIM`, etc., following `<group_name>` must all appear on one line.) Note that the beginning brace ({) must sit on a line by itself; the ending brace (}) must be preceded by a carriage return.

- *access* defines which thorns can use the following groups of variables. *access* can be either `public`, `protected` or `private`.
- *data_type* defines the data type of the variables in the group. Supported data types are `CHAR`, `BYTE`, `INT`, `REAL`, and `COMPLEX`.
- *group_name* must be an alphanumeric name (which may also contain underscores) which is unique across group and variable names within the scope of the thorn. A group name is compulsory.
- `[`*number*`]`, if present, indicates that this is a *vector* group. The number can be any valid arithmetical expression consisting of integers or integer-valued parameters. Each variable in that group appears as a one-dimensional array of grid variables. When the variable is accessed in the code, then the last index is the member-index, and any other indices are the normal spatial indices for a group of this type and dimension.
- `TYPE` designates the kind of variables held by the group. The choices are `GF`, `ARRAY` or `SCALAR`. This field is optional, with the default variable type being `SCALAR`.
- `DIM` defines the spatial dimension of the `ARRAY` or `GF`. The default value is `DIM=3`.
- `TIMELEVELS` defines the number of timelevels a group has if the group is of type `ARRAY` or `GF`, and can take any positive value. The default is one timelevel.
- `SIZE` defines the number grid-points an `ARRAY` has in each direction. This should be a comma-separated list of valid arithmetical expressions consisting of integers or integer-valued parameters.
- `DISTRIB` defines the processor decomposition of an `ARRAY`. `DISTRIB=DEFAULT` distributes `SIZE` grid-points across all processors. `DISTRIB=CONSTANT` implies that `SIZE` grid-points should be allocated on each processor. The default value is `DISTRIB=DEFAULT`.
- `GHOSTSIZE` defines number of ghost zones in each dimension of an `ARRAY`. It defaults to zero.
- `TAGS` defines an optional string which is used to create a set of key-value pairs associated with the group. The keys are case independent. The string (which must be deliminated by single or double quotes) is interpreted by the function `Util_TableSetFromString()`, which is described in the Reference Manual.
  Currently the CST parser and the flesh do not evaluate any information passed in an optional `TAGS` string. Thorns may do so by querying the key/value table information for a group by using `CCTK_GroupTagsTable()` and the appropriate `Util_TableGet*()` utility functions (see the ReferenceManual for detailed descriptions).
  For a list of currently supported `TAGS` key-value table information, please refer to the corresponding chapter in the documentation of the `CactusDoc` arrangement. (Section B2.5 on page B15 explains how to build this documentation).

- The (optional) block following the group declaration line, contains a list of variables contained in the group. All variables in a group have the same data type, variable type, dimension and distribution. The list can be separated by spaces, commas, or new lines. The variable names must be unique within the scope of the thorn. A variable can only be a member of one group. The block must be delimited by brackets on new lines. If no block is given after a group declaration line, a variable with the same name as the group is created. Apart from this case, a group name cannot be the same as the name of any variable seen by this thorn.
- An optional description of the group can be given on the last line. If the variable block is omitted, this description can be given at the end of the declaration line.

The process of sharing code among thorns using include files is discussed in Section C1.9.2.

D2.3 param.ccl

The parameter configuration file consists of a list of *parameter object specification items* (OSIs) giving the type and range of the parameter separated by optional *parameter data scoping items* (DSIs), which detail access to the parameter. (For a more extensive discussion of Cactus parameters, see Chapter C1.4.)

D2.3.1 Parameter Data Scoping Items

`<access>:`

The keyword `access` designates that all parameter object specification items, up to the next parameter data scoping item, are in the same protection or scoping class. `access` can take the values:

`global`	all thorns have access to global parameters
`restricted`	other thorns can have access to these parameters, if they specifically request it in their own param.ccl
`private`	only your thorn has access to private parameters
`shares`	in this case, an `implementation` name must follow the colon. It declares that all the parameters in the following scoping block are restricted variables from the specified `implementation`. (Note: only one implementation can be specified on this line.)

D2.3.2 Parameter Object Specification Items

```
[EXTENDS|USES] <parameter type> <parameter name>[[<len>]] "<parameter description>"
[AS <alias>] [STEERABLE=<NEVER|ALWAYS|RECOVER>]
[ACCUMULATOR=<expression>] [ACCUMULATOR-BASE=<parameter name>]
{
  <parameter values>
} <default value>
```

where the options `AS`, `STEERABLE`, etc., following *<parameter description>*, must all appear in one line. Note that the beginning brace ({) must sit on a line by itself; the ending brace (}) must be at the beginning of a line followed by *<default value>* on that same line.

- The *parameter values* depend on the *parameter type*, which may be one of the following:

`INT` The specification of *parameter value*s takes the form of one or more lines, each of the form

```
<range description> [::"<comment describing this range>"]
```

Here, a *<range description>* specifies a set of integers, and has one of the following forms:

```
*                            # means any integer
<integer>                    # means only <integer>
<lower bound>:<upper bound>  # means all integers in the range
                             # from <lower bound> to <upper bound>
<lower bound>:<upper bound>:<positive step>
                             # means all integers in the range
                             # from <lower bound> to <upper bound>
                             # in steps of <positive step>
```

where *<lower bound>* has one of the forms

```
<empty field>   # means no lower limit
*               # means no lower limit
<integer>       # means a closed interval starting at <integer>
[<integer>      # also means a closed interval starting at <integer>
(<integer>      # means an open interval starting at <integer>
```

and *<upper bound>* has one of the forms

```
<empty field>   # means no upper limit
*               # means no upper limit
<integer>       # means a closed interval ending at <integer>
<integer>]      # also means a closed interval ending at <integer>
<integer>)      # means an open interval ending at <integer>
```

`REAL` The range specification is the same as with integers, except that here, no *step* implies a continuum of values. Note that numeric constants should be expressed as in C (e.g. `1e-10`). Note also that one cannot use the Cactus types such as `CCTK_REAL4` to specify the precision of the parameter; parameters always have the default precision.

`KEYWORD` Each entry in the list of acceptable values for a keyword has the form

```
<keyword value>, <keyword value> :: "<description>"
```

Keyword values should be enclosed in double quotes. The double quotes are mandatory if the keyword contains spaces.

`STRING` Allowed values for strings should be specified using regular expressions. To allow any string, the regular expression `""` should be used. (An empty regular expression matches anything.) Regular expressions and string values should be enclosed in double quotes. The double quotes are mandatory if the regular expression or the string value is empty or contains spaces.

`BOOLEAN` No *`parameter values`* should be specified for a boolean parameter. The default value for a boolean can be

- True: `1`, `yes`, `y`, `t`, `true`
- False: `0`, `no`, `n`, `f`, `false`

Boolean values may optionally be enclosed in double quotes.

- The *`parameter name`* must be unique within the scope of the thorn.
- The *`default value`* must match one of the ranges given in the *`parameter type`*
- A thorn can declare that it `EXTENDS` a parameter of another thorn. This allows it to declare additional acceptable values. By default, it is acceptable for two thorns to declare the same value as acceptable.
- If the thorn wants to simply use a parameter from another thorn, without declaring additional values, use `USES` instead.
- `[`*`len`*`]` (where *`len`* is an integer), if present, indicates that this is an *array* parameter of *`len`* values of the specified type. (Note that the notation used above for the parameter specification breaks down here, as there must be square brackets around the length).
- *`alias`* allows a parameter to appear under a different name in this thorn, other than its original name in another thorn. The name, as seen in the parameter file, is unchanged.
- `STEERABLE` specifies when a parameter value may be changed. By default, parameters may not be changed after the parameter file has been read, or on restarting from checkpoint. This option relaxes this restriction, specifying that the parameter may be changed at recovery time from a parameter file or at any time using the flesh routine `CCTK_ParameterSet`—see the Reference Guide.

  The value `RECOVERY` is used in checkpoint/recovery situations, and indicates that the parameter may be altered until the value is read in from a recovery par file, but not after.
- `ACCUMULATOR` specifies that this is an *accumulator* parameter. Such parameters cannot be set directly, but are set by other parameters who specify this one as an `ACCUMULATOR-BASE`. The expression is a two-parameter arithmetical expression of x and y. Setting the parameter consists of evaluating this expression successively, with x being the current value of the parameter (at the first iteration this is the default value), and y the value of the setting parameter. This procedure is repeated, starting from the default value of the parameter, each time one of the setting parameters changes.
- `ACCUMULATOR-BASE` specifies the name of an `ACCUMULATOR` parameter which this parameter sets.

D2.4 schedule.ccl

(A more extensive discussion of Cactus scheduling is provided in Chapter C1.5.) A schedule configuration file consists of:

- *Assignment statements* to switch on storage for grid variables for the entire duration of program execution.
- *Schedule blocks* to schedule a subroutine from a thorn to be called at specific times during program execution in a given manner.
- *Conditional statements* for both assignment statements and schedule blocks to allow them to be processed depending on parameter values.

D2.4.1 Assignment Statements

Assignment statements, currently only assign storage.

These lines have the form:

```
[STORAGE: <group>[timelevels], <group>[timelevels]]
```

If the thorn is active, storage will be allocated, for the given groups, for the duration of program execution (unless storage is explicitly switched off by some call to `CCTK_DisableGroupStorage` within a thorn).

The storage line includes the number of timelevels to activate storage for, this number can be from 1 up to the maximum number or timelevels for the group, as specified in the defining `interface.ccl` file. If the maximum number of timelevels is 1 (the default), this number may be omitted. Alternatively *`timelevels`* can be the name of a parameter accessible to the thorn. The parameter name is the same as used in C routines of the thorn, fully qualified parameter names of the form *`thorn::parameter`* are not allowed. In this case 0 (zero) *`timelevels`* can be requested, which is equivalent to the `STORAGE` statement being absent.

The behaviour of an assignment statement is independent of its position in the schedule file (so long as it is outside a schedule block).

D2.4.2 Schedule Blocks

Each *schedule block* in the file `schedule.ccl` must have the syntax

```
schedule [GROUP] <function name|group name> AT|IN <time> \
     [AS <alias>] \
     [WHILE <variable>] [IF <variable>] \
     [BEFORE|AFTER <function name>|(<function name> <function name> ...)] \
{
  [LANG: <language>]
  [OPTIONS:       <option>,<option>...]
  [TAGS:          <keyword=value>,<keyword=value>...]
  [STORAGE:       <group>[timelevels],<group>[timelevels]...]
  [READS:         <group-or-variable>[(region)],[<group-or-variable>[(region)]]...]
  [WRITES:        <group-or-variable>[(region)],[<group-or-variable>[(region)]]...]
  [TRIGGER:       <group>,<group>...]
  [SYNCHRONISE:   <group>,<group>...]
  [OPTIONS:       <option>,<option>...]
} "Description of function"
```

`GROUP` Schedule a schedule group with the same options as a schedule function. The schedule group will be created if it doesn't exist.

`<function name|group name>` The name of a function or a schedule group to be scheduled. Function and schedule group names are case sensitive.

`<group>` A group of grid variables. Variable groups inherited from other thorns may be used, but they must then be fully qualified with the implementation name.

`AT` Functions can be scheduled to run at the Cactus schedule bins, for example, `CCTK_EVOL`, and `CCTK_STARTUP`. A complete list and description of these is provided in Appendix D4. The initial letters `CCTK_` are optional. Grid variables cannot be used in the `CCTK_STARTUP` and `CCTK_SHUTDOWN` timebins.

`IN` Schedules a function or schedule group to run in a schedule group, rather than in a Cactus timebin.

`AS` Provides an alias for a function or schedule group which should be used for scheduling before, after or in. This can be used to provide thorn independence for other thorns scheduling functions, or schedule groups relative to this one.

`WHILE` Executes a function or schedule group until the given variable (which must be a fully qualified integer grid scalar) has the value zero.

`IF` Executes a function or schedule group only if the given variable (which must be a fully qualified integer grid scalar) has a non-zero value.

`BEFORE/AFTER` Takes a function name, a function alias, a schedule group name, or a parentheses-enclosed whitespace-separated list of these. (Any names that are not provided by an active thorn are ignored.) Note that a single schedule block may have multiple `BEFORE/AFTER` clauses. See Section C1.5 ("Scheduling") in the Cactus Users' Guide for more information about `BEFORE/AFTER` clauses.

`LANG` The code language for the function (either `C` or `FORTRAN`). No language should be specified for a schedule group.

`OPTIONS` Schedule options are used for mesh refinement and multi-block simulations, and they determine "where" a routine executes. Possible options are:

`meta`

`meta_early`

`meta_late`

`global`

`global_early`

`global_late`

`level`

`singlemap`

`local` (default, may be omitted)

(Only one of these options may be used.) These options can be combined with the following:

`loop_meta`

`loop_global`

`loop_level`

`loop_singlemap`

`loop_local`

(At most one of the `loop_...` options may be used.)

`TAGS` Schedule tags. These tags must have the form `keyword=value`, and must be in a syntax accepted by `Util_TableCreateFromString`.

`STORAGE` List of variable groups which should have storage switched on for the duration of the function or schedule group. Each group must specify how many timelevels to activate storage for, from 1 up to the maximum number for the group as specified in the defining `interface.ccl` file. If the maximum is 1 (the default) this number may be omitted. Alternatively *`timelevels`* can be the name of a parameter accessible to the thorn. The parameter name is the same as used in C routines of the thorn, fully qualified parameter names of the form *`thorn::parameter`* are not allowed. In this case 0 (zero) *`timelevels`* can be requested, which is equivalent to the `STORAGE` statement being absent.

`READS/WRITES` `READS/WRITES` are used to declare which grid variables are read/written by the routine. This information is used e.g. to determine which variables need to be synchronized, copied between host and device for OpenCL or CUDA kernel, or poisoned as part of error checking.

`READS/WRITE` directives can be applied to either a variable or a group of variables and may also contain a region specification. The region specification can be EVERYWHERE, or ALL, INTERIOR, or IN, INTERIOWITHBOUNDARY, or BOUNDARY. A missing region defaults to EVERYWHERE for `READS` and INTERIOR for `WRITES`. If a function needs to read a given grid function everywhere and only the interior is valid (i.e. has been written to), then a synchronization is required and will happen automatically if enabled with the Cactus parameter `presync_mode`.

When grid functions are updated in this way, not only ghost zones, but boundary zones will be updated by the drier. Use either the function `Driver_SelectGroupForBC` or `Driver_SelectVarForBC` to tell the driver how it should update a group or variable. The arguments to both functions are the same as those found in the Boundary thorn, but with the prefix `Boundary_` instead of `Driver_`:

```
Driver_SelectGroupForBC(CCTK_ARGUMENTS,
                        CCTK_INT faces,
                        CCTK_INT width,
                        CCTK_INT table_handle,
                        CCTK_STRING group_name,
                        CCTK_STRING bc_name)

Driver_SelectVarForBC(CCTK_ARGUMENTS,
                      CCTK_INT faces,
                      CCTK_INT width,
                      CCTK_INT table_handle,
                      CCTK_STRING var_name,
                      CCTK_STRING bc_name)
```

Note that the above two functions, unlike their similarly named components in the Boundary thorn, only need to be called once. The information they provide will then be used each time the driver synchronizes the named grid function or group.

It is possible, however, that the thorn you are creating does not know the correct READS/WRITES information at compile time. I/O thorns, for example, do not know which grid functions they will access until run time. Likewise, the Method of Lines thorn does not know which grid functions it is operating on until run time.

For situations like these, we have the following functions which can be used to check READS/WRITES data and perform synchronization at run time: `Driver_RequireValidData`, `Driver_NotifyDataModified`, `Driver_GetValidRegion`, and `Driver_SetValidRegion`. For details on these functions, please consult the Reference Manual.

`TRIGGER` List of grid variables or groups to be used as triggers for causing an `ANALYSIS` function or group to be executed. Any schedule block for an analysis function or analysis group may contain a `TRIGGER` line.

`SYNCHRONISE` List of groups to be synchronised, as soon as the function or schedule group is exited.

`OPTIONS` List of additional options (see below) for the scheduled function or group of functions

Allowed Options

Cactus understands the following options. These options are interpreted by the driver, not by Cactus. The current set of options is useful for Berger-Oliger mesh refinement which has subcycling in time, and for multi-patch simulations in which the domain is split into several distinct patches. Given this, the meanings of the options below is only tentative, and their exact meaning needs to be obtained from the driver documentation. The standard driver PUGH ignores all options.

Option names are case-insensitive. There can be several options given at the same time.

`META` This routine will only be called once, even if several simulations are performed at the same time. This can be used, for example, to initialise external libraries, or to set up data structures that live in global variables.

`META-EARLY` This option is identical to to `META` option with the exception that the routine will be called together with the routines on the first subgrid.

`META-LATE` This option is identical to to `META` option with the exception that the routine will be called together with the routines on the last subgrid.

`GLOBAL` This routine will only be called once on a grid hierarchy, not for all subgrids making up the hierarchy. This can be used, for example, for analysis routines which use global reduction or interpolation routines, rather than the local subgrid passed to them, and hence should only be called once.

`GLOBAL-EARLY` This option is identical to to `GLOBAL` option with the exception that the routine will be called together with the routines on the first subgrid.

`GLOBAL-LATE` This option is identical to to `GLOBAL` option with the exception that the routine will be called together with the routines on the last subgrid.

`LEVEL` This routine will only be called once on any "level" of the grid hierarchy. That is, it will only be called once for any set of sub-grids which have the same `cctk_levfac` numbers.

`SINGLEMAP` This routine will only be called once on any of the "patches" that form a "level" of the grid hierarchy.

`LOCAL` (this is the default)
This routine will be called on every "component".

When the above options are used, it is often the case that a certain routine should, e.g. be called at the time for a `GLOBAL` routine, but should actually loop over all "components". The following set of options allows this:

`LOOP-META` Loop once.

`LOOP-GLOBAL` Loop over all simulations.

`LOOP-LEVEL` Loop over all "levels".

`LOOP-SINGLEMAP` Loop over all "patches".

`LOOP-LOCAL` Loop over all "components".

For example, the specification

```
OPTIONS: global loop-local
```

schedules a routine at the time when a `GLOBAL` routine is scheduled, and then calls the routine in a loop over all "components".

D2.4.3 Conditional Statements

Any schedule block or assignment statements can be optionally surrounded by conditional `if-elseif-else` constructs using the parameter data base. These can be nested, and have the general form:

```
if (<conditional-expression>)
{
  [<assignments>]
  [<schedule blocks>]
}
```

¡*conditional-expression*¿ can be any valid C construct evaluating to a truth value. Such conditionals are evaluated only at program startup, and are used to pick between different static schedule options. For dynamic scheduling, the `SCHEDULE WHILE` construction should be used.

Conditional constructs cannot be used inside a schedule block.

D2.5 configuration.ccl

[**NOTE:** The configuration.ccl is still relatively new, and not all features listed below may be fully implemented or functional.]

A configuration.ccl file defines **capabilities** which a thorn either provides or requires, or may use if available. Unlike **implementations**, only one thorn providing a particular capability may be compiled into a configuration at one time. Thus, this mechanism may be used to, for example: provide access to external libraries; provide access to functions which other thorns must call, but are too complex for function aliasing; or to split a thorn into several thorns, all of which require some common (not aliased) functions.

A configuration options file can contain any number of the following sections:

- `PROVIDES <Capability>`
  `{`
  `  SCRIPT <Configuration script>`
  `  [VERSION <Version String>]`
  `  LANG <Language>`
  `  [OPTIONS [<option>[,<option>]...]]`
  `}`

  Informs the CST that this thorn provides a given capability, and that this capability has a given detection script which may be used to configure it (e.g. running an autoconf script or detecting an external library's location). The script should output configuration information on its standard output—the syntax is described below in Section D2.5.1. The script may also indicate the failure to detect a capability by returning a non-zero exit code; this will stop the build after the CST stage.

  All capabilities have an optional version attached to them, so that other thorns can depend on such a specific version. Version strings can only contain letters, numbers, or ".+-:" (dot, plus, minus, colon), and have to start with a number. Specifying a version number is optional. If none is given, "0.0.1" is assumed.

  Scripts can be in any language. If an interpreter is needed to run the script, for example `Perl`, this should be indicated by the `LANG` option.

  The specified options are checked for in the original configuration, and any options passed on the command line (including an 'options' file) at compile time when the thorn is added, or if the CST is rerun. These options need be set only once, and will be remembered between builds.

- `REQUIRES <Capability> [(Comparison operator Version string)]`

  Informs the CST that this thorn requires a certain capability to be present. If no thorn providing the capability is in the ThornList, the build will stop after the CST stage.

  Optionally, thorns can depend on a capability version. This has to be enclosed in parentheses, following the capability name. Inside the parenthesis, first a comparison operator determines the kind of dependency on the requested capability version, which has to be attached without spaces. Valid operators and their meaning are:

  `<<` Requesting capability strictly older than given version

  `<=` Requesting capability older or equal to given version

  `=` Requesting capability exactly equal to given version

  `>=` Requesting capability newer or equal to given version

  `>>` Requesting capability strictly newer than given version.

Version strings are compared from left to right. First the initial part of each string consisting entirely of digits is determined. The integer values of these two parts are compared. If a difference is found it is returned. Then the initial part of the remainder of each string which consists entirely of non-digit characters is determined. The two parts are compared lexically, and any difference found is returned as the result of the comparison. The lexical comparison is a comparison of ASCII values. These two steps (comparing and removing initial non-digit strings and initial digit strings) are repeated until a difference is found or both strings are exhausted. Spaces inside the parentheses are allowed, except inside the operator or version string.

Example:

```
REQUIRES ExampleCapability (>=2.43.2dev-1)
```

- `OPTIONAL <Capability>`
  `{`
  `  DEFINE <macro>`
  `}`

  Informs the CST that this thorn may use a certain capability, if a thorn providing it is in the ThornList. If present, the preprocessor macro, `macro`, will be defined and given the value "1".

D2.5.1 Configuration Scripts

The configuration script may tell the CST to add certain features to the Cactus environment—either to the make system or to header files included by thorns. It does this by outputting lines to its standard output:

- `BEGIN DEFINE`
  `<text>`
  `END DEFINE`

  Places a set of definitions in a header file which will be included by all thorns using this capability (either through an OPTIONAL or REQUIRES entry in their configuration.ccl files).
- `INCLUDE_DIRECTORY <directory>`

  Adds a directory to the include path used for compiling files in thorns using this capability.
- `BEGIN MAKE_DEFINITION`
  `<text>`
  `END MAKE_DEFINITION`

  Adds a makefile definition into the compilation of all thorns using this capability.
- `BEGIN MAKE_DEPENDENCY`
  `<text>`
  `END MAKE_DEPENDENCY`

  Adds makefile dependency information into the compilation of all thorns using this capability.
- `LIBRARY <library>`

  Adds a library to the final cactus link.
- `LIBRARY_DIRECTORY <library>`

  Adds a directory to the list of directories searched for libraries at link time.

No other lines should be output by the script.

Chapter D3

Utility Routines

D3.1 Introduction

As well as the high-level `CCTK_*` routines, Cactus also provides a set of lower-level `Util_*` utility routines, which are mostly independent of the rest of Cactus. This chapter gives a general overview of programming with these utility routines.

D3.2 Key/Value Tables

D3.2.1 Motivation

Cactus functions may need to pass information through a generic interface. In the past, we have used various ad hoc means to do this, and we often had trouble passing "extra" information that wasn't anticipated in the original design. For example, for periodic output of grid variables, `CCTK_OutputVarAsByMethod()` requires that any parameters (such as hyperslabbing parameters) be appended as an option string to the variable's character string name. Similarly, elliptic solvers often need to pass various parameters, but we haven't had a good way to do this.

Key/value tables (*tables* for short) provide a clean solution to these problems. They're implemented by the `Util_Table*` functions (described in detail in the Reference Manual).

D3.2.2 The Basic Idea

Basically, a table is an object which maps strings to almost arbitrary user-defined data. (If you know Perl, a table is very much like a Perl hash table. Alternatively, if you know Unix shells, a table is like the set of all environment variables. As yet another analogy, if you know Awk, a table is like an Awk associative array.)[1]

[1] However, the present Cactus tables implementation is optimized for a relatively small number of distinct keys in any one table. It will still work OK for huge numbers of keys, but it will be slow.

More formally, a table is an object which stores a set of *keys* and a corresponding set of *values*. We refer to a (key,value) pair as a table *entry*.

Keys are C-style null-terminated character strings, with the slash character '`/`' reserved for future expansion.[2]

Values are 1-dimensional arrays of any of the usual Cactus data types, described in Section C1.9.8. A string can be stored by treating it as a 1-dimensional array of `CCTK_CHAR` (there's an example of this below).

The basic "life cycle" of a table looks like this:

1. Some code creates it with `Util_TableCreate()` or `Util_TableClone()`.
2. Some code (often the same piece of code, but maybe some other piece) sets entries in it using one or more of the `Util_TableSet*()`, `Util_TableSet*Array()`, `Util_TableSetGeneric()`, `Util_TableSetGenericArray()`, and/or `Util_TableSetString()` functions.
3. Some other piece or pieces of code can get (copies of) the values which were set, using one or more of the `Util_TableGet*()`, `Util_TableGet*Array()`, `Util_TableGetGeneric()`, `Util_TableGetGenericArray()`, and/or `Util_TableGetString()` functions.
4. When everyone is through with a table, some (single) piece of code should destroy it with `Util_TableDestroy()`.

There are also convenience functions `Util_TableSetFromString()` to set entries in a table based on a parameter-file-style string, and `Util_TableCreateFromString()` to create a table and then set entries in it based on a parameter-file-style string.

As well, there are "table iterator" functions `Util_TableIt*()` to allow manipulation of a table even if you don't know its keys.

A table has an integer "flags word" which may be used to specify various options, via bit flags defined in `util_Table.h`. For example, the flags word can be used to control whether keys should be compared as case sensitive or case insensitive strings. See the detailed function description of `Util_TableCreate()` in the Reference Manual for a list of the possible bit flags and their semantics.

D3.2.3 A Simple Example

Here's a simple example (in C)[3] of how to use a table:

```
#include "util_Table.h"
#include "cctk.h"

/* create a table and set some entries in it */
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);
Util_TableSetInt(handle, 2, "two");
Util_TableSetReal(handle, 3.14, "pi");
```

[2] Think of hierarchical tables for storing tree-like data structures.

[3] All (or almost all) of the table routines are also usable from Fortran. See the full descriptions in the Reference Manual for details.

```
...

/* get the values from the table */
CCTK_INT two_value;
CCTK_REAL pi_value;
Util_TableGetInt(handle, &two_value, "two");    /* sets two_value = 2 */
Util_TableGetReal(handle, &pi_value, "pi");     /* sets pi_value = 3.14 */
```

Actually, you shouldn't write code like this—in the real world errors sometimes happen, and it's much better to catch them close to their point of occurrence, rather than silently produce garbage results or crash your program. So, the *right* thing to do is to always check for errors. To allow this, all the table routines return a status, which is zero or positive for a successful return, but negative if and only if some sort of error has occurred.[4] So, the above example should be rewritten like this:

```
#include "util_Table.h"

/* create a table and set some entries in it */
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);
if (handle < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't create table!");

/* try to set some table entries */
if (Util_TableSetInt(handle, 2, "two") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't set integer value in table!");
if (Util_TableSetReal(handle, 3.14, "pi") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't set real value in table!");

...

/* try to get the values from the table */
CCTK_INT two_value;
CCTK_REAL pi_value;
if (Util_TableGetInt(handle, &two_value, "two") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get integer value from table!");
if (Util_TableGetReal(handle, &pi_value, "pi") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get integer value from table!");

/* if we get to here, then two_value = 2 and pi_value = 3.14 */
```

D3.2.4 Arrays as Table Values

As well as a single numbers (or characters or pointers), tables can also store 1-dimensional arrays of numbers (or characters or pointers).[5]

For example (continuing the previous example):

[4] Often (as in the examples here) you don't care about the details of which error occurred. But if you do, there are various error codes defined in `util_Table.h` and `util_ErrorCodes.h`; the detailed function descriptions in the Reference Manual say which error codes each function can return.

[5] Note that the table makes (stores) a *copy* of the array you pass in, so it's somewhat inefficient to store a large array (e.g. a grid function) this way. If this is a problem, consider storing a `CCTK_POINTER` (pointing to the array) in the table instead. (Of course, this requires that you ensure that the array still exists whenever that `CCTK_POINTER` is used.)

```
static const CCTK_INT a[3] = { 42, 69, 105 };
if (Util_TableSetIntArray(handle, 3, a, "my array") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't set integer array value in table!");

...

CCTK_INT blah[10];
int count = Util_TableGetIntArray(handle, 10, blah, "my array");
if (count < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get integer array value from table!");
/* now count = 3, blah[0] = 42, blah[1] = 69, blah[2] = 105, */
/*     and all remaining elements of blah[] are unchanged */
```

As you can see, a table entry remembers the length of any array value that has been stored in it.[6]

If you only want the first few values of a larger array, just pass in the appropriate length of your array, that's OK:

```
CCTK_INT blah2[2];
int count = Util_TableGetIntArray(handle, 2, blah2, "my array");
if (count < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get integer array value from table!");
/* now count = 3, blah2[0] = 42, blah2[1] = 69 */
```

You can even ask for just the first value:

```
CCTK_INT blah1;
int count = Util_TableGetInt(handle, &blah1, "my array");
if (count < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get integer array value from table!");
/* now count = 3, blah1 = 42 */
```

D3.2.5 Character Strings

One very common thing you might want to store in a table is a character string. While you could do this by explicitly storing an array of `CCTK_CHAR`, there are also routines specially for conveniently setting and getting strings:

```
if (Util_TableSetString(handle, "black holes are fun", "bh") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't set string value in table!");

...
char buffer[50];
if (Util_TableGetString(handle, 50, buffer, "bh") < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't get string value from table!");

/* now buffer[] contains the string "black holes are fun" */
```

[6] In fact, actually *all* table values are arrays—setting or getting a single value is just the special case where the array length is 1.

`Util_TableGetString()` guarantees that the string is terminated by a null character (‘`\0`’), and also returns an error if the string is too long for the buffer.

D3.2.6 Convenience Routines

There are also convenience routines for the common case of setting values in a table based on a string.

For example, the following code sets up exactly the same table as the example in Section D3.2.3:

```
#include <util_Table.h>

/* create a table and set some values in it */
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);
if (handle < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't create table!");

/* try to set some table entries */
if (Util_TableSetFromString(handle, "two=2 pi=3.14") != 2)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't set values in table!");
```

There is also an even higher-level convenience function `Util_TableCreateFromString()`: this creates a table with the case insensitive flag set (to match Cactus parameter file semantics), then (assuming no errors occurred) calls `Util_TableSetFromString()` to set values in the table.

For example, the following code sets up a table (with the case insensitive flag set) with four entries: an integer number (`two`), a real number (`pi`), a string (`buffer`), and an integer array with three elements (`array`):

```
#include <util_Table.h>

int handle = Util_TableCreateFromString(" two    = 2 "
                                        " pi     = 3.14 "
                                        " buffer = 'Hello World' "
                                        " array  = { 1 2 3 }");
if (handle < 0)
        CCTK_WARN(CCTK_WARN_ABORT, "couldn't create table from string!");
```

Note that this code passes a single string to `Util_TableCreateFromString()`[7], which then gets parsed into key/value pairs, with the key separated from its corresponding value by an equals sign.

Values for numbers are converted into integers (`CCTK_INT`) if possible (no decimal point appears in the value), otherwise into reals (`CCTK_REAL`). Strings must be enclosed in either single or double quotes. String values in single quotes are interpreted literally, strings in double quotes may contain character escape codes which then will be interpreted as in C. Arrays must be enclosed in curly braces, array elements must be single numbers of the same type (either all integer or all real).

[7] C automatically concatenates adjacent character string constants separated only by whitespace.

D3.2.7 Table Iterators

In the examples up to now, the code, which wanted to get values from the table, knew what the keys were. It's also useful to be able to write generic code which can operate on a table without knowing the keys. "Table iterators" ("iterators", for short) are used for this.

An iterator is an abstraction of a pointer to a particular table entry. Iterators are analogous to the `DIR *` pointers used by the POSIX `opendir()`, `readdir()`, `closedir()`, and similar functions, to Perl hash tables' `each()`, `keys()`, and `values()`, and to the C++ Standard Template Library's forward iterators.

At any time, the entries in a table may be considered to be in some arbitrary (implementation-defined) order; an iterator may be used to walk through some or all of the table entries in this order. This order is guaranteed to remain unchanged for any given table, so long as no changes are made to that table, i.e. so long as no `Util_TableSet*()`, `Util_TableSet*Array()`, `Util_TableSetGeneric()`, `Util_TableSetGenericArray()`, `Util_TableSetString()`, or `Util_TableDeleteKey()` calls are made on that table (making such calls on other tables doesn't matter). The order may change if there is any change in the table, and it may differ even between different tables with identical key/value contents (including those produced by `Util_TableClone()`).[8]

Any change in the table also invalidates all iterators pointing anywhere in the table; using any such iterator is an error. Multiple iterators may point into the same table; they all use the same order, and (unlike in Perl) they're all independent.

The detailed function description in the Reference Manual for `Util_TableItQueryKeyValueInfo()` has an example of using an iterator to print out all the entries in a table.

D3.2.8 Multithreading and Multiprocessor Issues

At the moment, the table functions are *not* thread-safe in a multithreaded environment.

Note that tables and iterators are process-wide, i.e. all threads see the same tables and iterators (think of them as like the Unix current working directory, with the various routines which modify the table or change iterators acting like a Unix `chdir()` system call).

In a multiprocessor environment, tables are always processor-local.

D3.2.9 Metadata about All Tables

Tables do not *themselves* have names or other attributes. However, we may add some special "system tables" to be used by Cactus itself to store this sort of information for those cases where it's needed. For example, we may add support for a "checkpoint me" bit in a table's flags word, so that if you want a table to be checkpointed, you just need to set this bit. In this case, the table will probably get a system generated name in the checkpoint dump file. But if you want the table to have some other name in the dump file, then you need to tell the checkpointing code that, by setting an appropriate entry in a checkpoint table. (You would find the checkpoint table by looking in a special "master system table" that records handles of other interesting tables.)

[8] For example, if tables were implemented by hashing, the internal order could be that of the hash buckets, and the hash function could depend on the internal table address.

Chapter D4

Schedule Bins

Using the `schedule.ccl` files, thorn functions can be scheduled to run in the different timebins which are executed by the Cactus flesh. This chapter describes these standard timebins, and shows the flow of program execution through them.

Scheduled functions must be declared as

In C:
```
#include "cctk_Arguments.h"
void MyFunction (CCTK_ARGUMENTS);
```

In C++:
```
#include "cctk_Arguments.h"
extern "C" void MyFunction (CCTK_ARGUMENTS);
```

In Fortran:
```
#include "cctk_Arguments.h"
subroutine MyFunction (CCTK_ARGUMENTS)
   DECLARE_CCTK_ARGUMENTS
end
```

Exceptions are the functions that are scheduled in the bins `CCTK_STARTUP`, `CCTK_RECOVER_PARAMETERS`, and `CCTK_SHUTDOWN`. They do not take arguments, and they return an integer. They must be declared as

In C:
```
int MyFunction (void);
```

In C++
```
extern "C" int MyFunction ();
```

In Fortran:
```
integer function MyFunction ()
end
```

The return value in `CCTK_STARTUP` and `CCTK_SHUTDOWN` is unused, and might in the future be used to indicate whether an error occurred. You should return 0.

The return value in `CCTK_RECOVER_PARAMETERS` should be zero, positive, or negative, indicating that no parameters were recovered, that parameters were recovered successfully, or that an error occurred, respectively. Routines in this bin are executed in alphabetical order, according to the owning thorn's name, until one returns a positive value. All later routines are ignored. Schedule clauses `BEFORE`, `AFTER`, `WHILE`, `IF`, etc., are ignored.

CCTK_RECOVER_PARAMETERS
: Used by thorns with relevant I/O methods as the point to read parameters when recovering from checkpoint files. Grid variables are not available in this timebin. Scheduling in this timebin is special (see above).

CCTK_STARTUP
: Run before any grids are constructed, this is the timebin, for example, where grid independent information (e.g. output methods, reduction operators) is registered. Note that since no grids are setup at this point, grid variables cannot be used in routines scheduled here.

CCTK_WRAGH
: This timebin is executed when all parameters are known, but before the driver thorn constructs the grid. It should only be used to set up information that is needed by the driver.

CCTK_PARAMCHECK
: This timebin is for thorns to check the validity of parameter combinations. This bin is also executed before the grid hierarchy is made, so that routines scheduled here only have access to the global grid size and the parameters.

CCTK_PREREGRIDINITIAL
: This timebin is used in mesh refinement settings. It is ignored for unigrid runs. This bin is executed whenever the grid hierarchy is about to change during evolution; compare CCTK_PREREGRID. Routines that decide the new grid structure should be scheduled in this bin.

CCTK_POSTREGRIDINITIAL
: This timebin is used in mesh refinement settings. It is ignored for unigrid runs. This bin is executed whenever the grid hierarchy or patch setup has changed during evolution; see CCTK_POSTREGRID. It is, e.g. necessary to re-apply the boundary conditions or recalculate the grid points' coordinates after every changing the grid hierarchy.

CCTK_BASEGRID
: This timebin is executed very early after a driver thorn constructs grid; this bin should only be used to set up coordinate systems on the newly created grids.

CCTK_INITIAL
: This is the place to set up any required initial data. This timebin is not run when recovering from a checkpoint file.

CCTK_POSTINITIAL
: This is the place to modify initial data, or to calculate data that depend on the initial data. This timebin is also not run when recovering from a checkpoint file.

CCTK_POSTRESTRICTINITIAL
: This timebin is used only in mesh refinement settings. It is ignored for unigrid runs. This bin is executed after each restriction operation while initial data are set up; compare CCTK_POSTRESTRICT. It is, e.g. necessary to re-apply the boundary conditions after every restriction operation.

CCTK_POSTPOSTINITIAL
: This is the place to modify initial data, or to calculate data that depend on the initial data. This timebin is executed after the recursive initialisation of finer grids if there is a mesh refinement hierarchy, and it is also not run when recovering from a checkpoint file.

CCTK_RECOVER_VARIABLES
: Used by thorns with relevant I/O methods as the point to read in all the grid variables when recovering from checkpoint files.

CCTK_POST_RECOVER_VARIABLES
: This timebin exists for scheduling any functions which need to modify grid variables after recovery.

CCTK_CPINITIAL	Used by thorns with relevant I/O methods as the point to checkpoint initial data if required.
CCTK_CHECKPOINT	Used by thorns with relevant I/O methods as the point to checkpoint data during the iterative loop when required.
CCTK_PREREGRID	This timebin is used in mesh refinement settings. It is ignored for unigrid runs. This bin is executed whenever the grid hierarchy is about to change during evolution; compare CCTK_PREREGRIDINITIAL. Routines that decide the new grid structure should be scheduled in this bin.
CCTK_POSTREGRID	This timebin is used in mesh refinement settings. It is ignored for unigrid runs. This bin is executed whenever the grid hierarchy or patch setup has changed during evolution; see CCTK_POSTREGRIDINITIAL. It is, e.g. necessary to re-apply the boundary conditions or recalculate the grid points' coordinates after every changing the grid hierarchy.
CCTK_PRESTEP	The timebin for scheduling any routines which need to be executed before any routines in the main evolution step. This timebin exists for thorn writers convenience, the BEFORE, AFTER, etc., functionality of the schedule.ccl file should allow all functions to be scheduled in the main CCTK_EVOL timebin.
CCTK_EVOL	The timebin for the main evolution step.
CCTK_POSTRESTRICT	This timebin is used only in mesh refinement settings. It is ignored for unigrid runs. This bin is executed after each restriction operation during evolution; compare CCTK_POSTRESTRICTINITIAL. It is, e.g. necessary to re-apply the boundary conditions after every restriction operation.
CCTK_POSTSTEP	The timebin for scheduling any routines which need to be executed after all the routines in the main evolution step. This timebin exists for thorn writers convenience, the BEFORE, AFTER, etc., functionality of the schedule.ccl file should allow all functions to be scheduled in the main CCTK_EVOL timebin.
CCTK_ANALYSIS	The ANALYSIS timebin is special, in that it is closely coupled with output, and routines which are scheduled here are typically only executed if output of analysis variables is required. Routines which perform analysis should be independent of the main evolution loop (that is, it should not matter for the results of a simulation whether routines in this timebin are executed or not).
CCTK_TERMINATE	Called after the main iteration loop when Cactus terminates. Note that sometime, in this timebin, a driver thorn should be destroying the grid hierarchy and removing grid variables.
CCTK_SHUTDOWN	Cactus final shutdown routines, after the grid hierarchy has been destroyed. Grid variables are no longer available.

Chapter D5

Flesh Parameters

The flesh parameters are defined in the file `src/param.ccl`.

D5.1 Private Parameters

Here, the default value is shown in square brackets, while curly braces show allowed parameter values.

`allow_mixeddim_gfs`
: Allow use of GFs from different dimensions [`no`]

`cctk_brief_output`
: Give only brief output [`no`]

`cctk_full_warnings`
: Give detailed information for each warning statement [`yes`]

`cctk_run_title`
: Description of this simulation [`""`]

`cctk_show_banners`
: Show any registered banners for the different thorns [`yes`]

`cctk_show_schedule`
: Print the scheduling tree to standard output [`yes`]

`cctk_strong_param_check`
: Die on parameter errors in `CCTK_PARAMCHECK` [`yes`]

`cctk_timer_output`
: Give timing information [`off`] {`off, full`}

`recovery_mode`
: How to behave when recovering from a checkpoint [`strict`] {`strict, relaxed`}

`highlight_warning_messages`
: Highlight CCTK warning messages [`yes`]

`info_format`
: Specifies the content and format of `CCTK_INFO()`/`CCTK_VINFO` messages. [`basic`] {`"basic"`, `"numeric time stamp"`, `"human-readable time stamp"`, `"full time stamp"`}

D5.2 Restricted Parameters

`cctk_final_time` Final time for evolution, overridden by `cctk_itlast` unless it is positive [`-1.0`]

`cctk_initial_time` Initial time for evolution [`0.0`]

`cctk_itlast` Final iteration number [`10`]

`max_runtime` Terminate evolution loop after a certain elapsed runtime (in minutes); set to zero to disable this termination condition [`0`]

`presync_mode` The `presync_mode` parameter defines the behavior of the automatic pre-synchronization of grid functions. A summary of the different modes available is available in Table D5.1.

The meaning of modes is as follows:

off : This is the default. Cactus behaves as it did prior to the inclusion of PreSync. READS and WRITES declarations in the schedule have no effect. Synchronization of ghost zones only happens when a SYNC statement is encountered in the schedule.

warn-only The same as `presync_off`, except that now, if the Driver believes that a SYNC is required but not executed, it issues a warning.

mixed-warn In this mode, Cactus tries to honor both SYNC statements in the schedule as well as providing automatic synchronization based on the READS/WRITES declarations. It may, however, issue warnings about possible problems.

mixed-error In this mode, Cactus tries to honor both SYNC statements in the schedule as well as providing automatic synchronization based on the READS/WRITES declarations. It may, however, issue errors when encountering possible problems.

presync-only In this mode, Cactus will only synchronize ghost zones based on READS/WRITES declarations and will ingore SYNC statements in the schedule.

Sync triggered by describes what triggers the synchronization of ghost zones for a grid function. "Schedule only," in this context, means an explicit declaration of `SYNC` in the schedule. "Driver," in this context, means that a synchronization is triggered because the schedule requires the variable to be valid everywhere, but it is only valid in the interior. "sync if driver" indicates that a physical boundary condition is registered for the grid function, and thus the driver can update the exterior of the grid function.

Can sync describes the condition in which the interior of a grid function contains valid data, but the exterior does not. This will produce a warning in `warn-only` mode, as it indicates a possibly incorrect synchronization.

Can't sync describes a situation where a read of the interior is needed (e.g. to perform a synchronization), but (as far as the driver knows) the interior of the grid function is not valid.

IO Thorns describes what the various IO thorns should do if they are asked to write out a grid function if they find invalid data in the interior and/or exterior of a grid function.

Old Macro refers to the familiar `DECLARE_CCTK_ARGUMENTS`, which declares all the grid functions visible to a thorn. The `DECLARE_CCTK_ARGUMENTS_CHECKED(func_name)` macros only declare arguments visible based on the READS/WRITES declarations.

Driver BC refers to boundary condition update functions registered via `Driver_SelectVarForBC` or `Driver_SelectGroupForBC`. Once registered, this function will be called whenever data is needed in the boundary for the grid function or group.

Scheduled SYNC refers to the explicit SYNC declaration in the `schedule.ccl`.

presync_mode	Synch triggered by	Can sync	Can't sync	IO Thorns	Old Macro	Driver BC	Scheduled SYNC
off	schedule only	nothing	nothing	nothing	allowed	ignored	honored
warn-only	schedule only	warn	warn	nothing	allowed	ignored	honored
mixed-warn	driver where possible, schedule otherwise	sync if driver, warn otherwise	warn	warn for driver bc, nothing otherwise	allowed	allowed	ignored for driver bc
mixed-error	driver where possible, schedule otherwise	sync if driver, error otherwise	error	error for driver bc, nothing otherwise	allowed	allowed	ignored for driver bc
presync-only	driver only	driver only	error	error	error	required	ignored

Table D5.1: Summary of modes and meanings for the `presync_mode` parameter.

`terminate` Condition on which to terminate evolution loop [`iteration`] {`never, iteration, time, runtime, any, all`}

`terminate_next` Terminate on next iteration ? [`no`]

Chapter D6

Using the Einstein Toolkit issue tracker

Bitbucket offers a web-based tool for tracking bug reports and feature requests. Cactus bugs and feature requests are handled using the Bitbucket issue tracker at https://bitbucket.org/einsteintoolkit/tickets/. Click on *Create* to create a new ticket in the system.

Here, we briefly describe the main categories when creating a Cactus problem report.

Title A brief and informative subject line.

Description Describe your problem precisely, if you get a core dump, include the stack trace, and if possible give the minimal number of thorns, this problems occurs with. Describe how to reproduce the problem if it is not clear. Note that the description field (and the comments) allow markdown syntax. This means that blocks of code or error messages should be surrounded by 3 backticks ``` ... ``` in order to avoid the text being interpreted as markdown commands. Click on the question mark link to learn more about the available markdown commands.

Kind Choose *bug* for cases where there is clearly something wrong and *enhancement* for a feature request.

Priority Pick whichever level is appropriate. *blocker* for issues that stop you using the code, *critical* for very serious problems, *major* for things which should definitely be addressed, *minor* for things which would be good to fix but not essential, and *trivial* for very low priority items. If in doubt, choose either *major* or *minor*.

Component Use *Cactus* for problems related to the Cactus flesh or one of the thorns in one of the Cactus arrangements (those in arrangements with names starting "Cactus").

Milestone This is used by the maintainers to indicate an intention to fix the problem before a particular release of Cactus.

Version The Cactus release you are using. You can find this out, for example, from an executable by typing `cactus_<config> -v`.

Keyword There is no explicit field to add keywords. It is however a good idea to add a line "Keyword: keyword1 keyword2" at the end to the Description to indicate that a

backport to the release version is required. For example, you could also enter the thorn name if the problem is with a specific thorn, the keyword *testsuite* if the ticket is related to a test failure, or the keyword *documentation* if the problem is related to the documentation.

Chapter D7

Using Tags

Finding your way around in the Cactus structure can be pretty difficult to handle. To make life easier there is support for *tags*, which lets you browse the code easily from within Emacs/XEmacs or `vi`. A tags database can be generated with `gmake`:

D7.1 Tags with Emacs

The command `gmake TAGS` will create a database for a routine reference table to be used within Emacs. This database can be accessed within Emacs if you add either of the following lines to your `.emacs` file:
`(setq tags-file-name "CACTUS_HOME/TAGS")` XOR
`(setq tag-table-alist '(("CACTUS_HOME" . "CACTUS_HOME/TAGS")))`
where `CACTUS_HOME` is your Cactus directory.

You can now easily navigate your Cactus flesh and Toolkits by searching for functions or "tags":

1. **Alt.** will find a tag
2. **Alt,** will find the next matching tag
3. **Alt*** will go back to the last matched tag

If you add the following lines to your `.emacs` file, the files found with tags will opened in *read-only* mode:

```
(defun find-tag-readonly (&rest a)
  (interactive)
  (call-interactively `find-tag a)
  (if (eq nil buffer-read-only) (setq buffer-read-only t))  )

(defun find-tag-readonly-next (&rest a)
  (interactive)
  (call-interactively `tags-loop-continue a)
  (if (eq nil buffer-read-only) (setq buffer-read-only t))  )
```

```
(global-set-key [(control meta \.)] 'find-tag-readonly)
(global-set-key [(control meta \,)] 'find-tag-readonly-next)
```

The key strokes to use when you want to browse in read-only mode are:

1. **CTRL Alt.** will find a tag and open the file in read-only mode
2. **CTRL Alt,** will find the next matching tag in read-only mode

D7.2 Tags with `vi`

The commands available are highly dependent upon the version of `vi`, but the following is a selection of commands which may work.

1. **vi -t tag** Start `vi` and position the cursor at the file and line where 'tag' is defined.
2. **Control-**] Find the tag under the cursor.
3. **:ta tag** Find a tag.
4. **:tnext** Find the next matching tag.
5. **:pop** Return to previous location before jump to tag.
6. **Control-T** Return to previous location before jump to tag (not widely implemented).

Note: Currently some of the `CCTK_FILEVERSION()` macros at the top of every source file don't have a trailing semicolon, which confuses the `etags` and `ctags` programs, so tags does not find the first subroutine in any file with this problem.