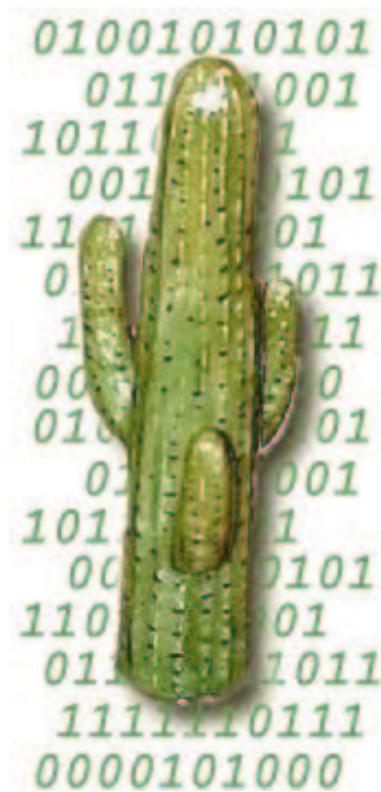


---

# Cactus 4.0

## Reference Manual

---



Revision: 4621

Date: 2010-06-03 16:31:49 -0400 (Thu, 03 Jun 2010)

# Contents

<b>A</b>	<b>CCTK_* Functions Reference</b>	<b>A1</b>
<b>A1</b>	<b>Functions Alphabetically</b>	<b>A3</b>
<b>A2</b>	<b>Full Description of Functions</b>	<b>A14</b>
<b>B</b>	<b>Util_* Functions Reference</b>	<b>B1</b>
<b>B1</b>	<b>Functions Alphabetically</b>	<b>B3</b>
	B1.1 Miscellaneous Functions . . . . .	B3
	B1.2 String Functions . . . . .	B3
	B1.3 Table Functions . . . . .	B3
<b>B2</b>	<b>Full Descriptions of Miscellaneous Functions</b>	<b>B6</b>
<b>B3</b>	<b>Full Descriptions of String Functions</b>	<b>B13</b>
<b>B4</b>	<b>Full Descriptions of Table Functions</b>	<b>B25</b>
<b>C</b>	<b>Appendices</b>	<b>C1</b>
<b>C1</b>	<b>Glossary</b>	<b>C3</b>
<b>C2</b>	<b>Configuration File Syntax</b>	<b>C8</b>
	C2.1 General Concepts . . . . .	C8
	C2.2 interface.ccl . . . . .	C8
	C2.2.1 Header Block . . . . .	C8
	C2.2.2 Include Files . . . . .	C9
	C2.2.3 Function Aliasing . . . . .	C9
	C2.2.4 Variable Blocks . . . . .	C10
	C2.3 param.ccl . . . . .	C12
	C2.3.1 Parameter Data Scoping Items . . . . .	C12
	C2.3.2 Parameter Object Specification Items . . . . .	C12
	C2.4 schedule.ccl . . . . .	C14
	C2.4.1 Assignment Statements . . . . .	C14
	C2.4.2 Schedule Blocks . . . . .	C15
	C2.4.3 Conditional Statements . . . . .	C17
	C2.5 configuration.ccl . . . . .	C18
	C2.5.1 Configuration Scripts . . . . .	C19
<b>C3</b>	<b>Utility Routines</b>	<b>C20</b>
	C3.1 Introduction . . . . .	C20

---

C3.2	Key/Value Tables . . . . .	C20
C3.2.1	Motivation . . . . .	C20
C3.2.2	The Basic Idea . . . . .	C20
C3.2.3	A Simple Example . . . . .	C21
C3.2.4	Arrays as Table Values . . . . .	C22
C3.2.5	Character Strings . . . . .	C23
C3.2.6	Convenience Routines . . . . .	C24
C3.2.7	Table Iterators . . . . .	C25
C3.2.8	Multithreading and Multiprocessor Issues . . . . .	C25
C3.2.9	Metadata about All Tables . . . . .	C25
C4	Schedule Bins . . . . .	C26
C5	Flesh Parameters . . . . .	C29
C5.1	Private Parameters . . . . .	C29
C5.2	Restricted Parameters . . . . .	C30
C6	Using GNATS . . . . .	C31
C7	Using CVS . . . . .	C32
C7.1	Essential CVS Commands . . . . .	C32
C7.2	CVS Options . . . . .	C33
C7.3	CVS Examples . . . . .	C34
C7.4	Checking Out Flesh and Thorns with CVS . . . . .	C36
C8	Using Tags . . . . .	C37
C8.1	Tags with Emacs . . . . .	C37
C8.2	Tags with vi . . . . .	C38
C9	ThornLists . . . . .	C39

---

## Preface

This document will eventually be a complete reference manual for the Cactus Code. However, it is currently under development, so please be patient if you can't find what you need. Please report omissions, errors, or suggestions to and of our contact addresses below, and we will try and fix them as soon as possible.

### Overview of documentation

This guide covers the following topics

**Part A: CCTK\_\* Function Reference.**

Here all the CCTK\_\*() Cactus flesh functions which are available to thorn writers are described.

**Part B: Util\_\* Function Reference.**

Here all the Util\_\*() Cactus flesh functions which are available to thorn writers are described.

**Part C: Appendices.**

These contain a description of the Cactus Configuration Language, a glossary, and other odds and ends, such as how to use GNATS or TAGS.

Other topics to be discussed in separate documents include:

**Users' Guide**

This gives a general overview of the Cactus Computational Tool Kit, including overall design/architecture, how to get/configure/compile/run it, and general discussions of the how to program in Cactus.

**Relativity Thorn Guide**

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

**Flesh Maintainers Guide**

This will contain all the gruesome details about the inner workings of Cactus, for all those who want or need to expand or maintain the core of Cactus.

### 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 our bug tracking system at <http://www.cactuscode.org>, or via email to [cactusmaint@cactuscode.org](mailto:cactusmaint@cactuscode.org). Alternatively, write to us at

The Cactus Team  
Center for Computation & Technology  
216 Johnston Hall  
Louisiana State University  
Baton Rouge, LA 70803  
USA

## 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

# CCTK\_\* Functions Reference

---

In this chapter all CCTK\_\* Cactus functions are described. These functions are callable from Fortran or C thorns. Note that whereas all functions are available from C, not all are currently available from Fortran.

# Chapter A1

## Functions Alphabetically

<code>CCTK_Abort</code>	<a href="#">[A15]</a> Causes abnormal Cactus termination
<code>CCTK_ActivatingThorn</code>	<a href="#">[A16]</a> Finds the thorn which activated a particular implementation
<code>CCTK_ActiveTimeLevels</code>	<a href="#">[A17]</a> Returns the number of active timelevels from a group name
<code>CCTK_ActiveTimeLevelsGI</code>	<a href="#">[A17]</a> Returns the number of active timelevels from a group index
<code>CCTK_ActiveTimeLevelsGN</code>	<a href="#">[A17]</a> Returns the number of active timelevels from a group name
<code>CCTK_ActiveTimeLevelsVI</code>	<a href="#">[A17]</a> Returns the number of active timelevels from a variable index
<code>CCTK_ActiveTimeLevelsVN</code>	<a href="#">[A17]</a> Returns the number of active timelevels from a variable name
<code>CCTK_ArrayGroupSize</code>	<a href="#">[A19]</a> Returns a pointer to the local size for a group, given by its group name
<code>CCTK_ArrayGroupSizeI</code>	<a href="#">[A20]</a> Returns a pointer to the local size for a group, given by its group index
<code>CCTK_Barrier</code>	<a href="#">[A21]</a> Synchronizes all processors
<code>CCTK_ClockRegister</code>	<a href="#">[A22]</a> Registers a new named clock with the Flesh.
<code>CCTK_Cmplx</code>	<a href="#">[A23]</a> Turns two real numbers into a complex number (only C)
<code>CCTK_CmplxAbs</code>	<a href="#">[A24]</a> Returns the absolute value of a complex number (only C)
<code>CCTK_CmplxAdd</code>	<a href="#">[A25]</a> Returns the sum of two complex numbers (only C)
<code>CCTK_CmplxConjg</code>	<a href="#">[A26]</a> Returns the complex conjugate of a complex number (only C)
<code>CCTK_CmplxCos</code>	<a href="#">[A27]</a> Returns the Cosine of a complex number (only C) [not yet available]
<code>CCTK_CmplxDiv</code>	<a href="#">[A28]</a> Returns the division of two complex numbers (only C)



CCTK_CmplxExp	[A29] Returns the Exponentiation of a complex number (only C) [not yet available]
CCTK_CmplxImag	[A30] Returns the imaginary part of a complex number (only C)
CCTK_CmplxLog	[A31] Returns the Logarithm of a complex number (only C) [not yet available]
CCTK_CmplxMul	[A32] Returns the multiplication of two complex numbers (only C)
CCTK_CmplxReal	[A33] Returns the real part of a complex number (only C)
CCTK_CmplxSin	[A34] Returns the Sine of a complex number (only C) [not yet available]
CCTK_CmplxSqrt	[A35] Returns the square root of a complex number (only C) [not yet available]
CCTK_CmplxSub	[A36] Returns the subtraction of two complex numbers (only C)
CCTK_CompileDate	[A37] Returns a formatted string containing the date stamp when Cactus was compiled
CCTK_CompileDateTime	[A38] Returns a formatted string containing the datetime stamp when Cactus was compiled
CCTK_CompileTime	[A39] Returns a formatted string containing the time stamp when Cactus was compiled
CCTK_CompiledImplementation	[A40] Return the name of the compiled implementation with given index
CCTK_CompiledThorn	[A41] Return the name of the compiled thorn with given index
CCTK_CoordDir	[A42] Give the direction for a given coordinate name ( <b>deprecated</b> )
CCTK_CoordIndex	[A43] Give the grid variable index for a given coordinate ( <b>deprecated</b> )
CCTK_CoordRange	[A44] Return the global upper and lower bounds for a given coordinate name on a cctkGH ( <b>deprecated</b> )
CCTK_CoordRegisterData	[A45] Register a coordinate as belonging to a coordinate system, with a given name and direction, and optionally with a grid variable ( <b>deprecated</b> )
CCTK_CoordRegisterRange	[A46] Saves the global upper and lower bounds for a given coordinate name on a cctkGH ( <b>deprecated</b> )
CCTK_CoordRegisterSystem	[A47] Registers a coordinate system with a given dimension ( <b>deprecated</b> )
CCTK_CoordSystemDim	[A48] Provides the dimension of a given coordinate system ( <b>deprecated</b> )
CCTK_CoordSystemHandle	[A49] Get the handle associated with a registered coordinate system ( <b>deprecated</b> )
CCTK_CoordSystemName	[A50] Provides the name of the coordinate system identified by its handle ( <b>deprecated</b> )
CCTK_CreateDirectory	[A51] Creates a directory

<code>CCTK_DecomposeName</code>	[A52] Given the full name of a variable/group, separates the name returning both the implementation and the variable/group
<code>CCTK_DisableGroupComm</code>	[A53] Disable the communication for a group
<code>CCTK_DisableGroupCommI</code>	[A54] Disable the communication for a group
<code>CCTK_DisableGroupStorage</code>	[A55] Disable the storage for a group
<code>CCTK_DisableGroupStorageI</code>	[A56] Disable the storage for a group
<code>CCTK_EnableGroupComm</code>	[A57] Enable the communication for a group
<code>CCTK_EnableGroupCommI</code>	[A58] Enable the communication for a group
<code>CCTK_EnableGroupStorage</code>	[A59] Enable the storage for a group
<code>CCTK_EnableGroupStorageI</code>	[A60] Enable the storage for a group
<code>CCTK_Equals</code>	[A61] Check a <code>STRING</code> or <code>KEYWORD</code> parameter for equality equality with a given string
<code>CCTK_Exit</code>	[A63] Causes normal Cactus termination
<code>CCTK_FirstVarIndex</code>	[A64] Given a group name returns the first variable index in the group
<code>CCTK_FirstVarIndexI</code>	[A65] Given a group index returns the first variable index in the group
<code>CCTK_FortranString</code>	[A66] Copy the contents of a C string into a Fortran string variable
<code>CCTK_FullName</code>	[A68] Given a variable index, returns the full name of the variable
<code>CCTK_GetClockName</code>	[A69] Given a pointer to a clock <code>cTimerVal</code> structure, returns the name of the clock.
<code>CCTK_GetClockResolution</code>	[A70] Given a pointer to a clock <code>cTimerVal</code> structure, returns the resolution of the clock.
<code>CCTK_GetClockSeconds</code>	[A71] Given a pointer to a clock <code>cTimerVal</code> structure, returns the elapsed time.
<code>CCTK_GetClockValue</code>	[A72] Given the name of a clock, returns a pointer to the corresponding <code>cTimerVal</code> structure within the <code>cTimerData</code> structure.
<code>CCTK_GetClockValueI</code>	[A73] Given the index of a clock, returns a pointer to the corresponding <code>cTimerVal</code> structure within the <code>cTimerData</code> structure.
<code>CCTK_GHExtension</code>	[A74] Get the pointer to a registered extension to the Cactus GH structure

<code>CCTK_GHExtensionHandle</code>	[A75] Get the handle associated with a extension to the Cactus GH structure
<code>CCTK_GridArrayReductionOperator</code>	[A76] The name of the implementation of a grid array reduction operator, or NULL if the handle is invalid
<code>CCTK_GroupbboxGI</code>	[A77] Given a group index, return an array of the bounding box of the group for each face
<code>CCTK_GroupbboxGN</code>	[A77] Given a group name, return an array of the bounding box of the group for each face
<code>CCTK_GroupbboxVI</code>	[A79] Given a variable index, return an array of the bounding box of the variable for each face
<code>CCTK_GroupbboxVN</code>	[A79] Given a variable name, return an array of the bounding box of the variable for each face
<code>CCTK_GroupData</code>	[A81] Given a group index, returns information about the variables held in the group
<code>CCTK_GroupDimFromVarI</code>	[A83] Given a variable index, returns the dimension of all variables in the group associated with this variable
<code>CCTK_GroupDimI</code>	[A84] Given a group index, returns the dimension of variables in that group
<code>CCTK_GroupDynamicData</code>	[A85] Given a group index, returns information about the variables held in the group
<code>CCTK_GroupGhostsizesI</code>	[A86] Given a group index, returns the ghost size array of that group
<code>CCTK_GroupgshGI</code>	[A87] Given a group index, return an array of the global size of the group in each dimension
<code>CCTK_GroupgshGN</code>	[A87] Given a group name, return an array of the global size of the group in each dimension
<code>CCTK_GroupgshVI</code>	[A89] Given a variable index, return an array of the global size of the variable in each dimension
<code>CCTK_GroupgshVN</code>	[A89] Given a variable name, return an array of the global size of the variable in each dimension
<code>CCTK_GroupIndex</code>	[A91] Get the index number for a group name
<code>CCTK_GroupIndexFromVar</code>	[A92] Given a variable name, returns the index of the associated group
<code>CCTK_GroupIndexFromVarI</code>	[A93] Given a variable index, returns the index of the associated group
<code>CCTK_GrouplbndGI</code>	[A94] Given a group index, return an array of the lower bounds of the group in each dimension
<code>CCTK_GrouplbndGN</code>	[A94] Given a group name, return an array of the lower bounds of the group in each dimension

CCTK_Group1bndVI	[A96] Given a variable index, return an array of the lower bounds of the variable in each dimension
CCTK_Group1bndVN	[A96] Given a variable name, return an array of the lower bounds of the variable in each dimension
CCTK_Group1shGI	[A98] Given a group index, return an array of the local size of the group in each dimension
CCTK_Group1shGN	[A98] Given a group name, return an array of the local size of the group in each dimension
CCTK_Group1shVI	[A100] Given a variable index, return an array of the local size of the variable in each dimension
CCTK_Group1shVN	[A100] Given a variable name, return an array of the local size of the variable in each dimension
CCTK_GroupName	[A102] Given a group index, returns the group name
CCTK_GroupNameFromVarI	[A103] Given a variable index, return the name of the associated group
CCTK_GroupnghostzonesGI	[A104] Given a group index, return an array with the number of ghostzones in each dimension of the group
CCTK_GroupnghostzonesGN	[A104] Given a group name, return an array with the number of ghostzones in each dimension of the group
CCTK_GroupnghostzonesVI	[A106] Given a variable index, return an array with the number of ghostzones in each dimension of the variable's group
CCTK_GroupnghostzonesVN	[A106] Given a group variable, return an array with the number of ghostzones in each dimension of the variable's group
CCTK_GroupSizesI	[A108] Given a group index, returns the size array of that group
CCTK_GroupStorageDecrease	[A109] Decrease the active number of timelevels for a list of groups
CCTK_GroupStorageIncrease	[A110] Increase the active number of timelevels for a list of groups
CCTK_GroupTagsTable	[A111] Given a group name, return the table handle of the group's tags table.
CCTK_GroupTagsTableI	[A112] Given a group index, return the table handle of the group's tags table.
CCTK_GroupTypeFromVarI	[A113] Provides a group's group type index given a variable index
CCTK_GroupTypeI	[A114] Provides a group's group type index given a group index
CCTK_GroupubndGI	[A115] Given a group index, return an array of the upper bounds of the group in each dimension

<code>CCTK_GroupubndGN</code>	<a href="#">[A115]</a> Given a group name, return an array of the upper bounds of the group in each dimension
<code>CCTK_GroupubndVI</code>	<a href="#">[A117]</a> Given a variable index, return an array of the upper bounds of the variable in each dimension
<code>CCTK_GroupubndVN</code>	<a href="#">[A117]</a> Given a variable name, return an array of the upper bounds of the variable in each dimension
<code>CCTK_ImpFromVarI</code>	<a href="#">[A119]</a> Given a variable index, returns the implementation name
<code>CCTK_ImplementationRequires</code>	<a href="#">[A120]</a> Return the ancestors for an implementation
<code>CCTK_ImplementationThorn</code>	<a href="#">[A121]</a> Returns the name of one thorn providing an implementation
<code>CCTK_ImpThornList</code>	<a href="#">[A122]</a> Return the thorns for an implementation
<code>CCTK_INFO</code>	<a href="#">[A123]</a> Macro to print a single string as an information message to screen
<code>CCTK_InfoCallbackRegister</code>	<a href="#">[A124]</a> Register one or more routines for dealing with information messages in addition to printing them to screen
<code>CCTK_InterpGridArrays</code>	<a href="#">[A126]</a> Performs an interpolation on a list of CCTK grid variables, using a chosen external local interpolation operator
<code>CCTK_InterpHandle</code>	<a href="#">[A132]</a> Returns the handle for a given interpolation operator
<code>CCTK_InterpLocalUniform</code>	<a href="#">[A133]</a> Interpolate a list of processor-local arrays which define a uniformly-spaced data grid
<code>CCTK_InterpRegisterOpLocalUniform</code>	<a href="#">[A139]</a> Registers a routine as a <code>CCTK_InterpLocalUniform</code> interpolation operator
<code>CCTK_IsFunctionAliased</code>	<a href="#">[A141]</a> Reports whether an aliased function has been provided
<code>CCTK_IsImplementationActive</code>	<a href="#">[A142]</a> Reports whether an implementation was activated in a parameter file
<code>CCTK_IsImplementationCompiled</code>	<a href="#">[A143]</a> Reports whether an implementation was compiled into a configuration
<code>CCTK_IsThornActive</code>	<a href="#">[A144]</a> Reports whether a thorn was activated in a parameter file
<code>CCTK_IsThornCompiled</code>	<a href="#">[A145]</a> Reports whether a thorn was compiled into a configuration
<code>CCTK_LocalArrayReduceOperator</code>	<a href="#">[A146]</a> Returns the name of a registered reduction operator
<code>CCTK_LocalArrayReduceOperatorImplementation</code>	<a href="#">[A147]</a> Provide the implementation which provides an local array reduction operator
<code>CCTK_LocalArrayReductionHandle</code>	<a href="#">[A148]</a> Returns the handle of a given local array reduction operator

<code>CCTK_MaxDim</code>	[A149] Get the maximum dimension of any grid variable
<code>CCTK_MaxGFDim</code>	[A150] Get the maximum dimension of all grid functions
<code>CCTK_MaxTimeLevels</code>	[A151] Gives the maximum number of timelevels for a group
<code>CCTK_MaxTimeLevelsGI</code>	[A152] Gives the maximum number of timelevels for a group
<code>CCTK_MaxTimeLevelsGN</code>	[A153] Gives the maximum number of timelevels for a group
<code>CCTK_MaxTimeLevelsVI</code>	[A154] Gives the maximum number of timelevels for a variable
<code>CCTK_MaxTimeLevelsVN</code>	[A155] Gives the maximum number of timelevels for a variable
<code>CCTK_MyProc</code>	[A156] Get the local processor number
<code>CCTK_nProcs</code>	[A157] Get the total number of processors used
<code>CCTK_NullPointer</code>	[A158] Returns a C-style NULL pointer value
<code>CCTK_NumCompiledImplementations</code>	[A159] Return the number of implementations compiled in
<code>CCTK_NumCompiledThorns</code>	[A160] Return the number of thorns compiled in
<code>CCTK_NumGridArrayReductionOperators</code>	[A161] The number of grid array reduction operators registered
<code>CCTK_NumGroups</code>	[A162] Get the number of groups of variables compiled in the code
<code>CCTK_NumIOMethods</code>	[A163] Returns the total number of I/O methods registered with the flesh
<code>CCTK_NumLocalArrayReduceOperators</code>	[A164] The number of local reduction operators registered
<code>CCTK_NumReductionArraysGloballyOperators</code>	[A165] The number of global array reduction operators registered
<code>CCTK_NumTimeLevels</code>	[A166] Returns the number of active timelevels from a group name ( <b>deprecated</b> )
<code>CCTK_NumTimeLevelsGI</code>	[A166] Returns the number of active timelevels from a group index ( <b>deprecated</b> )
<code>CCTK_NumTimeLevelsGN</code>	[A166] Returns the number of active timelevels from a group name ( <b>deprecated</b> )
<code>CCTK_NumTimeLevelsVI</code>	[A166] Returns the number of active timelevels from a variable index ( <b>deprecated</b> )
<code>CCTK_NumTimeLevelsVN</code>	[A166] Returns the number of active timelevels from a variable name ( <b>deprecated</b> )
<code>CCTK_NumTimerClocks</code>	[A168] Returns the number of clocks in a <code>cTimerData</code> structure.
<code>CCTK_NumVars</code>	[A169] Get the number of grid variables compiled in the code

<code>CCTK_NumVarsInGroup</code>	<a href="#">[A170]</a> Provides the number of variables in a group from the group name
<code>CCTK_NumVarsInGroupI</code>	<a href="#">[A171]</a> Provides the number of variables in a group from the group index
<code>CCTK_OutputGH</code>	<a href="#">[A172]</a> Conditional output of all variables on a GH by all I/O methods
<code>CCTK_OutputVar</code>	<a href="#">[A173]</a> Output of a single variable by all I/O methods
<code>CCTK_OutputVarAs</code>	<a href="#">[A174]</a> Output of a single variable as an alias by all I/O methods
<code>CCTK_OutputVarAsByMethod</code>	<a href="#">[A175]</a> Output of a single variable as an alias by a single I/O method
<code>CCTK_OutputVarByMethod</code>	<a href="#">[A176]</a> Output of a single variable by a single I/O method
<code>CCTK_ParallelInit</code>	<a href="#">[A177]</a> Initializes the parallel subsystem
<code>CCTK_ParameterData</code>	<a href="#">[A178]</a> Get parameter properties for given parameter/thorn pair
<code>CCTK_ParameterGet</code>	<a href="#">[A179]</a> Get the data pointer to and type of a parameter's value
<code>CCTK_ParameterLevel</code>	<a href="#">[A180]</a> Return the parameter checking level
<code>CCTK_ParameterQueryTimesSet</code>	<a href="#">[A181]</a> Return number of times a parameter has been set
<code>CCTK_ParameterSet</code>	<a href="#">[A182]</a> Sets the value of a parameter
<code>CCTK_ParameterSetNotifyRegister</code>	<a href="#">[A184]</a> Registers a parameter set operation notify callback
<code>CCTK_ParameterSetNotifyUnregister</code>	<a href="#">[A186]</a> Unregisters a parameter set operation notify callback
<code>CCTK_ParameterValString</code>	<a href="#">[A187]</a> Get the string representation of a parameter's value
<code>CCTK_ParameterWalk</code>	<a href="#">[A189]</a> Walk through the list of parameters
<code>CCTK_PARAMWARN</code>	<a href="#">[A190]</a> Prints a warning from parameter checking, and possibly stops the code
<code>CCTK_PointerTo</code>	<a href="#">[A191]</a> Returns a pointer to a Fortran variable.
<code>CCTK_PrintGroup</code>	<a href="#">[A192]</a> Prints a group name from its index
<code>CCTK_PrintString</code>	<a href="#">[A193]</a> Prints a Cactus string to screen (from Fortran)
<code>CCTK_PrintVar</code>	<a href="#">[A194]</a> Prints a variable name from its index
<code>CCTK_QueryGroupStorage</code>	<a href="#">[A195]</a> Queries storage for a group given by its group name
<code>CCTK_QueryGroupStorageB</code>	<a href="#">[A196]</a> Queries storage for a group given by its group name or index
<code>CCTK_QueryGroupStorageI</code>	<a href="#">[A197]</a> Queries storage for a group given by its group index

<code>CCTK_ReduceArraysGlobally</code>	[A198] Reduces a list of local arrays globally
<code>CCTK_ReduceGridArrays</code>	[A202] Reduces a list of local arrays (new grid array reduction API)
<code>CCTK_ReduceLocalArrays</code>	[A206] Reduces a list of local arrays (new local array reduction API) Returns the address of a variable passed in by reference from a Fortran routine
<code>CCTK_ReductionHandle</code>	[A210] Get the handle for a registered reduction operator
<code>CCTK_RegisterBanner</code>	[A211] Register a banner for a thorn
<code>CCTK_RegisterGHExtension</code>	[A212] Register the name of an extension to the Cactus GH
<code>CCTK_RegisterGHExtensionInitGH</code>	[A213] Register a routine for providing initialisation for an extension to the Cactus GH
<code>CCTK_RegisterGHExtensionScheduleTraverseGH</code>	[A214] Register a GH extension schedule traversal routine
<code>CCTK_RegisterGridArrayReductionOperator</code>	[A216] Registers a function as a grid array reduction operator of a certain name
<code>CCTK_RegisterGHExtensionSetupGH</code>	[A215] Register a routine for setting up an extension to the Cactus GH
<code>CCTK_RegisterIOMethod</code>	[A217] Registers a new I/O method
<code>CCTK_RegisterIOMethodOutputGH</code>	[A218] Registers an I/O method's routine for conditional output
<code>CCTK_RegisterIOMethodOutputVarAs</code>	[A219] Registers an I/O method's routine for unconditional output
<code>CCTK_RegisterIOMethodTimeToOutput</code>	[A220] Register a routine for deciding if it is time to output for an IO method
<code>CCTK_RegisterIOMethodTriggerOutput</code>	[A221] Register a routine for dealing with trigger output for an IO method
<code>CCTK_RegisterLocalArrayReductionOperator</code>	[A222] Registers a function as a reduction operator of a certain name
<code>CCTK_RegisterReduceArraysGloballyOperator</code>	[A223] Register a function as providing a global array reduction operation
<code>CCTK_RegisterReductionOperator</code>	[A224] Register a function as providing a reduction operation
<code>CCTK_SchedulePrintTimes</code>	[A225] Output the timing results for a certain schedule item to stdout
<code>CCTK_SchedulePrintTimesToFile</code>	[A226] Output the timing results for a certain schedule item to a file
<code>CCTK_SetupGH</code>	[A227] Sets up a CCTK grid hierarchy



<code>CCTK_SyncGroup</code>	<a href="#">[A228]</a> Synchronize the ghost zones for a group of variables (identified by the group name)
<code>CCTK_SyncGroupI</code>	<a href="#">[A230]</a> Synchronize the ghost zones for a group of variables (identified by the group index)
<code>CCTK_SyncGroupsI</code>	<a href="#">[A232]</a> Synchronize the ghost zones for a list of groups of variables (identified by their group indices)
<code>CCTK_TerminateNext</code>	<a href="#">[A234]</a> Causes a Cactus simulation to terminate after the next iteration
<code>CCTK_TerminationReached</code>	<a href="#">[A235]</a> Returns true if <code>CCTK_TerminateNext</code> has been called.
<code>CCTK_ThornImplementation</code>	<a href="#">[A236]</a> Returns the implementation provided by the thorn
<code>CCTK_Timer</code>	<a href="#">[A237]</a> Fills a timer <code>cTimerData</code> structure with current values of all clocks of a timer with a given name.
<code>CCTK_TimerCreate</code>	<a href="#">[A238]</a> Create a timer with a given name, returns a timer index.
<code>CCTK_TimerCreateData</code>	<a href="#">[A239]</a> Allocates a timer <code>cTimerData</code> structure.
<code>CCTK_TimerCreateI</code>	<a href="#">[A240]</a> Create an unnamed timer, returns a timer index.
<code>CCTK_TimerDestroy</code>	<a href="#">[A241]</a> Reclaims resources for a timer with a given name.
<code>CCTK_TimerDestroyData</code>	<a href="#">[A242]</a> Reclaims resources of a timer <code>cTimerData</code> structure.
<code>CCTK_TimerDestroyI</code>	<a href="#">[A243]</a> Reclaims resources for a timer with a given index.
<code>CCTK_TimerI</code>	<a href="#">[A244]</a> Fills a timer <code>cTimerData</code> structure with current values of all clocks of a timer with a given index.
<code>CCTK_TimerReset</code>	<a href="#">[A245]</a> Initialises the timer with a given name.
<code>CCTK_TimerResetI</code>	<a href="#">[A246]</a> Initialises the timer with a given index.
<code>CCTK_TimerStart</code>	<a href="#">[A247]</a> Initialises the timer with a given name.
<code>CCTK_TimerStartI</code>	<a href="#">[A248]</a> Initialises the timer with a given index.
<code>CCTK_TimerStop</code>	<a href="#">[A249]</a> Gets current values for all clocks of the timer with a given name.
<code>CCTK_TimerStopI</code>	<a href="#">[A250]</a> Gets current values for all clocks of the timer with a given index.
<code>CCTK_VarDataPtr</code>	<a href="#">[A251]</a> Returns the data pointer for a grid variable
<code>CCTK_VarDataPtrB</code>	<a href="#">[A252]</a> Returns the data pointer for a grid variable from the variable index or name
<code>CCTK_VarDataPtrI</code>	<a href="#">[A253]</a> Returns the data pointer for a grid variable from the variable index
<code>CCTK_VarIndex</code>	<a href="#">[A254]</a> Get the index for a variable
<code>CCTK_VarName</code>	<a href="#">[A255]</a> Given a variable index, returns the variable name
<code>CCTK_VarTypeI</code>	<a href="#">[A256]</a> Provides variable type index from the variable index

<code>CCTK_VarTypeSize</code>	<a href="#">[A257]</a> Provides variable type size in bytes from the variable type index
<code>CCTK_VInfo</code>	<a href="#">[A258]</a> Prints a formatted string with a variable argument list as an information message to screen
<code>CCTK_VWarn</code>	<a href="#">[A259]</a> Prints a formatted string with a variable argument list as a warning message to standard error and possibly stops the code
<code>CCTK_WARN</code>	<a href="#">[A261]</a> Macro to print a single string as a warning message to standard error and possibly stop the code
<code>CCTK_WarnCallbackRegister</code>	<a href="#">[A263]</a> Register one or more routines for dealing with warning messages in addition to printing them to standard error

## Chapter A2

# Full Description of Functions

## CCTK\_Abort

---

Abnormal Cactus termination.

### Synopsis

```
C          #include "cctk.h"

          int dummy = CCTK_Abort(const cGH *cctkGH);

Fortran    #include "cctk.h"

          subroutine CCTK_Abort (dummy, cctkGH)
             integer      dummy
             CCTK_POINTER cctkGH
          end subroutine CCTK_Abort
```

### Result

The function never returns, and hence never produces a result.

### Parameters

GH ( $\neq$  NULL)     Pointer to a valid Cactus grid hierarchy.

### Discussion

This routine causes an immediate, abnormal Cactus termination. It never returns to the caller.

### See Also

CCTK_Exit <a href="#">[A63]</a>	Exit the code cleanly
CCTK_WARN <a href="#">[A261]</a>	Macro to print a single string as a warning message and possibly stop the code
CCTK_Warn <a href="#">[A261]</a>	Prints a single string as a warning message and possibly stops the code
CCTK_VWarn <a href="#">[A259]</a>	Prints a formatted string with a variable argument list as a warning message to standard error and possibly stops the code

### Errors

The function never returns, and hence never reports an error.

### Examples

```
C          #include "cctk.h"
          CCTK_Abort (cctkGH);

Fortran    #include "cctk.h"
          integer dummy
          call CCTK_Abort (dummy, cctkGH)
```

**CCTK\_ActivatingThorn**

---

Finds the thorn which activated a particular implementation.

**Synopsis**

```
C          #include "cctk.h"

          const char *thorn = CCTK_ActivatingThorn(const char *name);
```

**Result**

thorn Name of activating thorn, or NULL if inactive

**Parameters**

name Implementation name

**See Also**

CCTK\_CompiledImplementation [\[A40\]](#) Return the name of the compiled implementation with given index

CCTK\_CompiledThorn [\[A41\]](#) Return the name of the compiled thorn with given index

CCTK\_ImplementationRequires [\[A120\]](#) Return the ancestors for an implementation

CCTK\_ImplementationThorn [\[A121\]](#) Returns the name of one thorn providing an implementation.

CCTK\_ImpThornList [\[A122\]](#) Return the thorns for an implementation

CCTK\_IsImplementationActive [\[A142\]](#) Reports whether an implementation was activated in a parameter file

CCTK\_IsImplementationCompiled [\[A143\]](#) Reports whether an implementation was compiled into a configuration

CCTK\_IsThornActive [\[A144\]](#) Reports whether a thorn was activated in a parameter file

CCTK\_IsThornCompiled [\[A145\]](#) Reports whether a thorn was compiled into a configuration

CCTK\_NumCompiledImplementations [\[A159\]](#) Return the number of implementations compiled in

CCTK\_NumCompiledThorns [\[A160\]](#) Return the number of thorns compiled in

CCTK\_ThornImplementation [\[A236\]](#) Returns the implementation provided by the thorn

**Errors**

NULL The implementation is inactive, or an error occurred.

**CCTK\_ActiveTimeLevels**

---

Returns the number of active time levels for a group.

**Synopsis**

```
C          #include "cctk.h"

          int timelevels = CCTK_ActiveTimeLevels(const cGH *cctkGH,
                                                  const char *groupname);

          int timelevels = CCTK_ActiveTimeLevelsGI(const cGH *cctkGH,
                                                  int groupindex);

          int timelevels = CCTK_ActiveTimeLevelsGN(const cGH *cctkGH,
                                                  const char *groupname);

          int timelevels = CCTK_ActiveTimeLevelsVI(const cGH *cctkGH,
                                                  int varindex);

          int timelevels = CCTK_ActiveTimeLevelsVN(const cGH *cctkGH,
                                                  const char *varname);

Fortran    #include "cctk.h"

          subroutine CCTK_ActiveTimeLevels(timelevels, cctkGH, groupname)
             integer      timelevels
             CCTK_POINTER  cctkGH
             character*(*) groupname
          end subroutine CCTK_ActiveTimeLevels

          subroutine CCTK_ActiveTimeLevelsGI(timelevels, cctkGH, groupindex)
             integer      timelevels
             CCTK_POINTER  cctkGH
             integer      groupindex
          end subroutine CCTK_ActiveTimeLevelsGI

          subroutine CCTK_ActiveTimeLevelsGN(timelevels, cctkGH, groupname)
             integer      timelevels
             CCTK_POINTER  cctkGH
             character*(*) groupname
          end subroutine CCTK_ActiveTimeLevelsGN

          subroutine CCTK_ActiveTimeLevelsVI(timelevels, cctkGH, varindex)
             integer      timelevels
             CCTK_POINTER  cctkGH
             integer      varindex
          end subroutine CCTK_ActiveTimeLevelsVI

          subroutine CCTK_ActiveTimeLevelsVN(timelevels, cctkGH, varname)
             integer      timelevels
             CCTK_POINTER  cctkGH
             character*(*) varname
```

```
end subroutine CCTK_ActiveTimeLevelsVN
```

**Result**

`timelevels`      The currently active number of timelevels for the group.

**Parameters**

`GH` ( $\neq$  NULL)      Pointer to a valid Cactus grid hierarchy.  
`groupname`      Name of the group.  
`groupindex`      Index of the group.  
`varname`      Name of a variable in the group.  
`varindex`      Index of a variable in the group.

**Discussion**

This function returns the number of timelevels for which storage has been activated, which is always equal to or less than the maximum number of timelevels which may have storage provided by `CCTK_MaxTimeLevels`.

**See Also**

`CCTK_MaxTimeLevels` [\[A151\]](#)      Return the maximum number of active timelevels.  
`CCTK_NumTimeLevels` [\[A166\]](#)      Deprecated; same as `CCTK_ActiveTimeLevels`.  
`CCTK_GroupStorageDecrease` [\[A109\]](#)      Base function, overloaded by the driver, which decreases the number of active timelevels, and also returns the number of active timelevels.  
`CCTK_GroupStorageIncrease` [\[A110\]](#)      Base function, overloaded by the driver, which increases the number of active timelevels, and also returns the number of active timelevels.

**Errors**

`timelevels < 0`      Illegal arguments given.

**CCTK\_ArrayGroupSize**

---

Returns a pointer to the processor-local size for variables in a group, specified by its name, in a given dimension.

**Synopsis**

```
C          #include "cctk.h"
          int *size = CCTK_ArrayGroupSize(const cGH *cctkGH,
                                          int dir,
                                          const char *groupname);
```

**Result**

NULL                    A NULL pointer is returned if the group index or the dimension given are invalid.

**Parameters**

GH ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dir ( $\geq 0$ )	Which dimension of array to query.
groupname	Name of the group.

**Discussion**

For a CCTK\_ARRAY or CCTK\_GF group, this routine returns a pointer to the processor-local size for variables in that group in a given direction. The direction is counted in C order (zero being the lowest dimension).

This function returns a pointer to the result for technical reasons; so that it will efficiently interface with Fortran. This may change in the future. Consider using CCTK\_GroupgshGN instead.

**See Also**

CCTK_GroupgshGN <a href="#">[A87]</a>	Returns an array with the array size in all dimensions.
...	There are many related functions which grab information from the GH, but many are not yet documented.



**CCTK\_ArrayGroupSizeI**

---

Returns a pointer to the processor-local size for variables in a group, specified by its index, in a given dimension.

**Synopsis**

```
C          #include "cctk.h"
          int *size = CCTK_ArrayGroupSizeI(const cGH *cctkGH,
                                           int dir,
                                           int groupi);
```

**Result**

NULL                    A NULL pointer is returned if the group index or the dimension given are invalid.

**Parameters**

GH ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dir ( $\geq 0$ )	Which dimension of array to query.
groupi	The group index.

**Discussion**

For a CCTK\_ARRAY or CCTK\_GF group, this routine returns a pointer to the processor-local size for variables in that group in a given direction. The direction is counted in C order (zero being the lowest dimension).

This function returns a pointer to the result for technical reasons; so that it will efficiently interface with Fortran. This may change in the future. Consider using CCTK\_GroupgshGI instead.

**See Also**

CCTK_GroupgshGI	<a href="#">[A87]</a>	Returns an array with the array size in all dimensions.
...		There are many related functions which grab information from the GH, but many are not yet documented.

## CCTK\_Barrier

---

Synchronizes all processors at a given execution point. This routine synchronizes all processors in a parallel job at a given point of execution. No processor will continue execution until all other processors have called `CCTK_Barrier`. Note that this is a collective operation – it must be called by all processors otherwise the code will hang.

### Synopsis

<b>C</b>	<code>int istat = CCTK_Barrier(int cGH, const cGH *cctkGH)</code>
<b>Fortran</b>	<code>subroutine CCTK_Barrier(cGH, cctkGH)</code> <code>integer          cGH</code> <code>CCTK_Pointer  cctkGH</code>

**CCTK\_ClockRegister**

---

Registers a named timer clock with the Flesh.

**Synopsis**

```
C          int err = CCTK_ClockRegister(name, functions)
```

**Parameters**

```
const char * name
```

The name the clock will be given

```
const cClockFuncs * functions
```

The structure holding the function pointers that define the clock

**Discussion**

The `cClockFuncs` structure contains function pointers defined by the clock module to be registered.

**Errors**

A negative return value indicates an error.

**CCTK\_Cmplx**

---

Turns two real numbers into a complex number

**Synopsis**

```
C          CCTK_COMPLEX cmpno = CCTK_Cmplx( CCTK_REAL realpart, CCTK_REAL imagpart)
```

**Parameters**

<code>cmpno</code>	The complex number
<code>realpart</code>	The real part of the complex number
<code>imagpart</code>	The imaginary part of the complex number

**Examples**

```
C          cmpno = CCTK_Cmplx(re,im);
```

**CCTK\_CmplxAbs**

---

Absolute value of a complex number

**Synopsis**

```
C          CCTK_COMPLEX absval = CCTK_CmplxAbs( CCTK_COMPLEX inval)
```

**Parameters**

<code>absval</code>	The computed absolute value
<code>realpart</code>	The complex number who absolute value is to be returned

**Examples**

```
C          absval = CCTK_CmplxAbs(inval);
```

**CCTK\_CmplxAdd**

---

Sum of two complex numbers

**Synopsis**

```
C          CCTK_COMPLEX addval = CCTK_CmplxAdd( CCTK_COMPLEX inval1, CCTK_COMPLEX inval2)
```

**Parameters**

<code>addval</code>	The computed added value
<code>inval1</code>	The first complex number to be summed
<code>inval2</code>	The second complex number to be summed

**Examples**

```
C          addval = CCTK_CmplxAdd(inval1,inval2);
```

**CCTK\_CmplxConjg**

---

Complex conjugate of a complex number

**Synopsis**

```
C          CCTK_COMPLEX conjgval = CCTK_CmplxConjg( CCTK_COMPLEX inval)
```

**Parameters**

<code>conjval</code>	The computed conjugate
<code>inval</code>	The complex number to be conjugated

**Examples**

```
C          conjgval = CCTK_CmplxConjg(inval);
```

**CCTK\_CmplxCos**

---

Cosine of a complex number

**Synopsis**

```
C          CCTK_COMPLEX cosval = CCTK_CmplxCos( CCTK_COMPLEX inval)
```

**Parameters**

<code>cosval</code>	The computed cosine
<code>inval</code>	The complex number to be cosined

**Discussion**

**NOT YET AVAILABLE**

**Examples**

```
C          cosval = CCTK_CmplxCos(inval);
```



**CCTK\_CmplxDiv**

---

Division of two complex numbers

**Synopsis**

```
C          CCTK_COMPLEX divval = CCTK_CmplxDiv( CCTK_COMPLEX inval1, CCTK_COMPLEX inval2)
```

**Parameters**

<code>divval</code>	The divided value
<code>inval1</code>	The enumerator
<code>inval1</code>	The denominator

**Examples**

```
C          divval = CCTK_CmplxDiv(inval1,inval2);
```

**CCTK\_CmplxExp**

---

Exponent of a complex number

**Synopsis**

```
C          CCTK_COMPLEX expval = CCTK_CmplxExp( CCTK_COMPLEX inval)
```

**Parameters**

<code>expval</code>	The computed exponent
<code>inval</code>	The complex number to be exponented

**Discussion**

**NOT YET AVAILABLE**

**Examples**

```
C          expval = CCTK_CmplxExp(inval);
```

**CCTK\_CmplxImag**

---

Imaginary part of a complex number

**Synopsis**

```
C          CCTK_REAL imval = CCTK_CmplxImag( CCTK_COMPLEX inval)
```

**Parameters**

<code>imval</code>	The imaginary part
<code>inval</code>	The complex number

**Discussion**

The imaginary part of a complex number  $z = a + bi$  is  $b$ .

**Examples**

```
C          imval = CCTK_CmplxImag(inval);
```

**CCTK\_CmplxLog**

---

Logarithm of a complex number

**Synopsis**

```
C          CCTK_COMPLEX logval = CCTK_CmplxLog( CCTK_COMPLEX inval)
```

**Parameters**

logval	The computed logarithm
inval	The complex number

**Discussion**

**NOT YET AVAILABLE**

**Examples**

```
C          logval = CCTK_CmplxLog(inval);
```

**CCTK\_CmplxMul**

---

Multiplication of two complex numbers

**Synopsis**

**C**                    `CCTK_COMPLEX mulval = CCTK_CmplxMul( CCTK_COMPLEX inval1, CCTK_COMPLEX inval2)`

**Parameters**

<code>mulval</code>	The product
<code>inval1</code>	First complex number to be multiplied
<code>inval2</code>	Second complex number to be multiplied

**Discussion**

The product of two complex numbers  $z_1 = a_1 + b_1i$  and  $z_2 = a_2 + b_2i$  is  $z = (a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i$ .

**Examples**

**C**                    `mulval = CCTK_CmplxMul(inval1,inval2);`

**CCTK\_CmplxReal**

---

Real part of a complex number

**Synopsis**

```
C          CCTK_REAL reval = CCTK_CmplxReal( CCTK_COMPLEX inval)
```

**Parameters**

<code>reval</code>	The real part
<code>inval</code>	The complex number

**Discussion**

The real part of a complex number  $z = a + bi$  is  $a$ .

**Examples**

```
C          reval = CCTK_CmplxReal(inval);
```

**CCTK\_CmplxSin**

---

Sine of a complex number

**Synopsis**

```
C          CCTK_COMPLEX sinval = CCTK_CmplxSin( CCTK_COMPLEX inval)
```

**Parameters**

<code>sinval</code>	The computed sine
<code>inval</code>	The complex number to be Sined

**Discussion**

**NOT YET AVAILABLE**

**Examples**

```
C          sinval = CCTK_CmplxSin(inval);
```

**CCTK\_CmplxSqrt**

---

Square root of a complex number

**Synopsis**

```
C          CCTK_COMPLEX sqrtval = CCTK_CmplxSqrt( CCTK_COMPLEX inval)
```

**Parameters**

expval	The computed square root
inval	The complex number to be square rooted

**Discussion**

**NOT YET AVAILABLE**

**Examples**

```
C          sqrtval = CCTK_CmplxSqrt(inval);
```



**CCTK\_CmplxSub**

---

Subtraction of two complex numbers

**Synopsis**

```
C          CCTK_COMPLEX subval = CCTK_CmplxSub( CCTK_COMPLEX inval1, CCTK_COMPLEX inval2)
```

**Parameters**

<code>addval</code>	The computed subtracted value
<code>inval1</code>	The complex number to be subtracted from
<code>inval2</code>	The complex number to subtract

**Discussion**

If  $z_1 = a_1 + b_1i$  and  $z_2 = a_2 + b_2i$  then

$$z_1 - z_2 = (a_1 - a_2) + (b_1 - b_2)i$$

**Examples**

```
C          subval = CCTK_CmplxSub(inval1,inval2);
```

**CCTK\_CompileDate**

---

Returns a formatted string containing the date stamp when Cactus was compiled

**Synopsis**

```
C          #include "cctk.h"

          const char *compile_date = CCTK_CompileDate ();
```

**Result**

`compile_date`      formatted string containing the date stamp

**See Also**

`CCTK_CompileTime` [[A39](#)]      Returns a formatted string containing the time stamp when Cactus was compiled

`CCTK_CompileDateTime` [[A38](#)]      Returns a formatted string containing the datetime stamp when Cactus was compiled

**CCTK\_CompileDateTime**

---

Returns a formatted string containing the datetime stamp when Cactus was compiled

**Synopsis**

```
C          #include "cctk.h"

          const char *compile_datetime = CCTK_CompileDateTime ();
```

**Result**

`compile_datetime`  
formatted string containing the datetime stamp

**Discussion**

If possible, the formatted string returned contains the datetime in a machine-processable format as defined in ISO 8601 chapter 5.4.

**See Also**

<code>CCTK_CompileDate</code> <a href="#">[A37]</a>	Returns a formatted string containing the date stamp when Cactus was compiled
<code>CCTK_CompileTime</code> <a href="#">[A39]</a>	Returns a formatted string containing the time stamp when Cactus was compiled

**CCTK\_CompileTime**

---

Returns a formatted string containing the time stamp when Cactus was compiled

**Synopsis**

```
C          #include "cctk.h"

          const char *compile_time = CCTK_CompileTime ();
```

**Result**

`compile_time`      formatted string containing the time stamp

**See Also**

`CCTK_CompileDate` [\[A37\]](#)      Returns a formatted string containing the date stamp when Cactus was compiled

`CCTK_CompileDateTime` [\[A38\]](#)      Returns a formatted string containing the datetime stamp when Cactus was compiled

**CCTK\_CompiledImplementation**

---

Return the name of the compiled implementation with given index.

**Synopsis**

```
C          #include "cctk.h"

          const char *imp = CCTK_CompiledImplementation(int index);
```

**Result**

imp                      Name of the implementation

**Parameters**

index                    Implementation index, with  $0 \leq \text{index} < \text{numimpls}$ , where `numimpls` is returned by `CCTK_NumCompiledImplementations`.

**See Also**

<code>CCTK_ActivatingThorn</code> <a href="#">[A16]</a>	Finds the thorn which activated a particular implementation
<code>CCTK_CompiledThorn</code> <a href="#">[A41]</a>	Return the name of the compiled thorn with given index
<code>CCTK_ImplementationRequires</code> <a href="#">[A120]</a>	Return the ancestors for an implementation
<code>CCTK_ImplementationThorn</code> <a href="#">[A121]</a>	Returns the name of one thorn providing an implementation.
<code>CCTK_ImpThornList</code> <a href="#">[A122]</a>	Return the thorns for an implementation
<code>CCTK_IsImplementationActive</code> <a href="#">[A142]</a>	Reports whether an implementation was activated in a parameter file
<code>CCTK_IsImplementationCompiled</code> <a href="#">[A143]</a>	Reports whether an implementation was compiled into a configuration
<code>CCTK_IsThornActive</code> <a href="#">[A144]</a>	Reports whether a thorn was activated in a parameter file
<code>CCTK_IsThornCompiled</code> <a href="#">[A145]</a>	Reports whether a thorn was compiled into a configuration
<code>CCTK_NumCompiledImplementations</code> <a href="#">[A159]</a>	Return the number of implementations compiled in
<code>CCTK_NumCompiledThorns</code> <a href="#">[A160]</a>	Return the number of thorns compiled in
<code>CCTK_ThornImplementation</code> <a href="#">[A236]</a>	Returns the implementation provided by the thorn

**Errors**

NULL                    Error.

**CCTK\_CompiledThorn**

---

Return the name of the compiled thorn with given index.

**Synopsis**

```
C          #include "cctk.h"

          const char *thorn = CCTK_CompiledThorn(int index);
```

**Result**

thorn            Name of the thorn

**Parameters**

index            Thorn index, with  $0 \leq \text{index} < \text{numthorns}$ , where `numthorns` is returned by `CCTK_NumCompiledThorns`.

**See Also**

`CCTK_ActivatingThorn` [\[A16\]](#)            Finds the thorn which activated a particular implementation

`CCTK_CompiledImplementation` [\[A40\]](#)            Return the name of the compiled implementation with given index

`CCTK_ImplementationRequires` [\[A120\]](#)            Return the ancestors for an implementation

`CCTK_ImplementationThorn` [\[A121\]](#)            Returns the name of one thorn providing an implementation.

`CCTK_ImpThornList` [\[A122\]](#)            Return the thorns for an implementation

`CCTK_IsImplementationActive` [\[A142\]](#)            Reports whether an implementation was activated in a parameter file

`CCTK_IsImplementationCompiled` [\[A143\]](#)            Reports whether an implementation was compiled into a configuration

`CCTK_IsThornActive` [\[A144\]](#)            Reports whether a thorn was activated in a parameter file

`CCTK_IsThornCompiled` [\[A145\]](#)            Reports whether a thorn was compiled into a configuration

`CCTK_NumCompiledImplementations` [\[A159\]](#)            Return the number of implementations compiled in

`CCTK_NumCompiledThorns` [\[A160\]](#)            Return the number of thorns compiled in

`CCTK_ThornImplementation` [\[A236\]](#)            Returns the implementation provided by the thorn

**Errors**

NULL            Error.

## CCTK\_CoordDir

---

Give the direction for a given coordinate.

**All the CCTK\_Coord\* APIs 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).**

### Synopsis

**C**                    `int dir = CCTK_CoordDir( const char * coordname, const char * systemname)`  
**Fortran**           `call CCTK_CoordDir(dir , coordname, systemname )`

integer dir  
character\*(\*) coordname  
character\*(\*) systemname

### Parameters

**dir**                    The direction of the coordinate  
**coordname**           The name assigned to this coordinate  
**systemname**          The name of the coordinate system

### Discussion

The coordinate name is independent of the grid function name.

### Examples

**C**                    `direction = CCTK_CoordDir("xdir","cart3d");`  
**Fortran**           `call CCTK_COORDDIR(direction,"radius","spher3d")`

## CCTK\_CoordIndex

---

Give the grid variable index for a given coordinate.

**All the CCTK\_Coord\* APIs 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).**

### Synopsis

**C** `int index = CCTK_CoordIndex( int direction, const char * coordname, const char * systemname );`

**Fortran** `call CCTK_CoordIndex(index , direction, coordname, systemname )`

```
integer index
integer direction
character(*) coordname
character(*) systemname
```

### Parameters

<code>index</code>	The coordinates associated grid variable index
<code>direction</code>	The direction of the coordinate in this coordinate system
<code>coordname</code>	The name assigned to this coordinate
<code>systemname</code>	The coordinate system for this coordinate

### Discussion

The coordinate name is independent of the grid variable name. To find the index, the coordinate system name must be given, and either the coordinate direction or the coordinate name. The coordinate name will be used if the coordinate direction is given as less than or equal to zero, otherwise the coordinate name will be used.

### Examples

**C** `index = CCTK_CoordIndex(-1,"xdir","cart3d");`

**C** `call CCTK_COORDINDEX(index,one,"radius","spher2d")`



## CCTK\_CoordRange

---

Return the global upper and lower bounds for a given coordinate.

All the CCTK\_Coord\* APIs 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).

### Synopsis

**C**                    `int ierr = CCTK_CoordRange( const cGH * cctkGH, CCTK_REAL * lower, CCTK_REAL * upper, i`

**Fortran**            `call CCTK_CoordRange(ierr , cctkGH, lower, upper, direction, coordname, systemname )`

```
integer ierr
CCTK_POINTER cctkGH
CCTK_REAL lower
CCTK_REAL upper
integer direction
character(*) coordname
character(*) systemname
```

### Parameters

<code>ierr</code>	Error code
<code>cctkGH</code>	pointer to CCTK grid hierarchy
<code>lower</code>	Global lower bound of the coordinate (POINTER in C)
<code>upper</code>	Global upper bound of the coordinate (POINTER in C)
<code>direction</code>	Direction of coordinate in coordinate system
<code>coordname</code>	Coordinate name
<code>systemname</code>	Coordinate system name

### Discussion

The coordinate name is independent of the grid function name. The coordinate range is registered by `CCTK_CoordRegisterRange`. To find the range, the coordinate system name must be given, and either the coordinate direction or the coordinate name. The coordinate direction will be used if is given as a positive value, otherwise the coordinate name will be used.

### Examples

**C**                    `ierr = CCTK_CoordRange(cctkGH, &xmin, &xmax, -1, "xdir", "mysystem");`

**Fortran**            `call CCTK_COORDRANGE(ierr, cctkGH, Rmin, Rmax, -1, "radius", "sphersystem")`

**CCTK\_CoordRegisterData**

---

Define a coordinate in a given coordinate system.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

**C**                    `int ierr = CCTK_CoordRegisterData( int direction, const char * gvname, const char * coordname, const char * systemname );`

**Fortran**            `call CCTK_CoordRegisterData(ierr , direction, gvname, coordname, systemname )`

```
integer ierr
integer direction
character(*) gvname
character(*) coordname
character(*) systemname
```

**Parameters**

<b>ierr</b>	Error code
<b>direction</b>	Direction of coordinate in coordinate system
<b>gvname</b>	Name of grid variable associated with coordinate
<b>coordname</b>	Name of this coordinate
<b>systemname</b>	Name of this coordinate system

**Discussion**

There must already be a coordinate system registered, using CCTK\_CoordRegisterSystem.

**Examples**

**C**                    `ierr = CCTK_CoordRegisterData(1,"coordthorn::myx","x2d","cart2d");`

**Fortran**            `two = 2`  
`call CCTK_COORDREGISTERDATA(ierr,two,"coordthorn::mytheta","spher3d")`

**CCTK\_CoordRegisterRange**

---

Assign the global maximum and minimum values of a coordinate on a given grid hierarchy.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

<b>C</b>	<code>int ierr = CCTK_CoordRegisterRange( const cGH * cctkGH, CCTK_REAL min, CCTK_REAL max, i</code>
<b>Fortran</b>	<code>call CCTK_CoordRegisterRange(ierr , cctkGH, min, max, direction, coordname, systemname</code>
	<code>integer ierr</code>
	<code>CCTK_POINTER cctkGH</code>
	<code>CCTK_REAL min</code>
	<code>CCTK_REAL max</code>
	<code>integer direction</code>
	<code>character*(*) coordname</code>
	<code>character*(*) systemname</code>

**Parameters**

<code>ierr</code>	Error code
<code>dimension</code>	Pointer to CCTK grid hierarchy
<code>min</code>	Global minimum of coordinate
<code>max</code>	Global maximum of coordinate
<code>direction</code>	Direction of coordinate in coordinate system
<code>coordname</code>	Name of coordinate in coordinate system
<code>systemname</code>	Name of this coordinate system

**Discussion**

There must already be a coordinate registered with the given name, with CCTK\_CoordRegisterData. The coordinate range can be accessed by CCTK\_CoordRange.

**Examples**

<b>C</b>	<code>ierr = CCTK_CoordRegisterRange(cctkGH,-1.0,1.0,1,"x2d","cart2d");</code>
<b>Fortran</b>	<code>min = 0</code>
	<code>max = 3.1415d0/2.0d0</code>
	<code>two = 2</code>
	<code>call CCTK_COORDREGISTERRANGE(ierr,min,max,two,"coordthorn::mytheta","spher3d")</code>

**CCTK\_CoordRegisterSystem**

---

Assigns a coordinate system with a chosen name and dimension.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

<b>C</b>	<code>int ierr = CCTK_CoordRegisterSystem( int dimension, const char * systemname)</code>
<b>Fortran</b>	<code>call CCTK_CoordRegisterSystem(ierr , dimension, systemname )</code>
	<code>integer ierr</code>
	<code>integer dimension</code>
	<code>character*(*) systemname</code>

**Parameters**

<code>ierr</code>	Error code
<code>dimension</code>	Dimension of coordinate system
<code>systemname</code>	Unique name assigned to coordinate system

**Examples**

<b>C</b>	<code>ierr = CCTK_CoordRegisterSystem(3,"cart3d");</code>
<b>Fortran</b>	<code>three = 3</code> <code>call CCTK_COORDREGISTERSYSTEM(ierr,three,"sphersystem")</code>

**CCTK\_CoordSystemDim**

---

Give the dimension for a given coordinate system.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

<b>C</b>	<code>int dim = CCTK_CoordSystemDim( const char * systemname)</code>
<b>Fortran</b>	<code>call CCTK_CoordSystemDim(dim , systemname )</code>
	<code>integer dim</code>
	<code>character*(*) systemname</code>

**Parameters**

<code>dim</code>	The dimension of the coordinate system
<code>systemname</code>	The name of the coordinate system

**Examples**

<b>C</b>	<code>dim = CCTK_CoordSystemDim("cart3d");</code>
<b>Fortran</b>	<code>call CCTK_COORDSYSTEMDIM(dim,"spher3d")</code>

**CCTK\_CoordSystemHandle**

---

Returns the handle associated with a registered coordinate system.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

<b>C</b>	<code>int handle = CCTK_CoordSystemHandle( const char * systemname)</code>
<b>Fortran</b>	<code>call CCTK_CoordSystemHandle(handle , systemname )</code>
	<code>integer handle</code>
	<code>character*(*) systemname</code>

**Parameters**

<code>handle</code>	The coordinate system handle
<code>systemname</code>	Name of the coordinate system

**Examples**

<b>C</b>	<code>handle = CCTK_CoordSystemHandle("my coordinate system");</code>
<b>Fortran</b>	<code>call CCTK_CoordSystemHandle(handle,"my coordinate system")</code>

**Errors**

<b>negative</b>	A negative return code indicates an invalid coordinate system name.
-----------------	---

**CCTK\_CoordSystemName**

---

Returns the name of a registered coordinate system.

All the CCTK\_Coord\* APIs 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).

**Synopsis**

```
C          const char * systemname = CCTK_CoordSystemName( int handle)
```

**Parameters**

handle	The coordinate system handle
systemname	The coordinate system name

**Discussion**

No Fortran routine exists at the moment.

**Examples**

```
C          systemname = CCTK_CoordSystemName(handle);  
          handle = CCTK_CoordSystemHandle(systemname);
```

**Errors**

NULL	A NULL pointer is returned if an invalid handle was given.
------	--

## CCTK\_CreateDirectory

---

Create a directory with required permissions

### Synopsis

**C**                    `int ierr = CCTK_CreateDirectory( int mode, const char * pathname)`

**Fortran**           `call CCTK_CreateDirectory(ierr , mode, pathname )`

integer ierr  
integer mode  
character\*(\*) pathname

### Parameters

**ierr**                Error code

**mode**              Permission mode for new directory as an octal number

**pathname**         Directory to create

### Discussion

To create a directory readable by everyone, but writeable only by the user running the code, the permission mode would be 0755. Alternatively, a permission mode of 0777 gives everyone unlimited access; the user's `umask` setting should cut this down to whatever the user's normal default permissions are anyway.

Note that (partly for historical reasons and partly for Fortran 77 compatability) the order of the arguments is the opposite of that of the usual Unix `mkdir(2)` system call.

### Examples

**C**                    `ierr = CCTK_CreateDirectory(0755, "Results/New");`

**Fortran**           `call CCTK_CREATEDIRECTORY(ierr,0755, "Results/New")`

### Errors

1	Directory already exists
0	Directory successfully created
-1	Memory allocation failed
-2	Failed to create directory
-3	Some component of <b>pathname</b> already exists but is not a directory



**CCTK-DecomposeName**

---

Given the full name of a variable/group, separates the name returning both the implementation and the variable/group

**Synopsis**

```
C          int istat = CCTK-DecomposeName( const char * fullname, char ** imp, char ** name)
```

**Parameters**

<code>istat</code>	Status flag returned by routine
<code>fullname</code>	The full name of the group/variable
<code>imp</code>	The implementation name
<code>name</code>	The group/variable name

**Discussion**

The implementation name and the group/variable name must be explicitly freed after they have been used.

No Fortran routine exists at the moment.

**Examples**

```
C          istat = CCTK-DecomposeName("evolve::scalars",imp,name)
```

**CCTK\_DisableGroupComm**

---

Turn communications off for a group of grid variables

**Synopsis**

```
C          int istat = CCTK_DisableGroupComm( cGH * cctkGH, const char * group)
```

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy

**Discussion**

Turning off communications means that ghost zones will not be communicated during a call to `CCTK_SyncGroup`. By default communications are all off.

**CCTK\_DisableGroupCommI**

---

Turn communications off for a group of grid variables.

**Synopsis**

```
C          int istat = CCTK_DisableGroupCommI(cGH * cctkGH, int group);
```

**Result**

0 The Group has been disabled.

**Parameters**

cctkGH pointer to CCTK grid hierarchy  
group number of group of grid variables to turn off

**Discussion**

Turning off communications means that ghost zones will not be communicated during a call to `CCTK_SyncGroup`. By default communications are all off.

**See Also**

CCTK\_DisableGroupComm [\[A53\]](#) Turn communications off for a group of grid variables.  
CCTK\_EnableGroupCommI [\[A58\]](#) Turn communications on for a group of grid variables.  
CCTK\_EnableGroupComm [\[A57\]](#) Turn communications on for a group of grid variables.

### CCTK\_DisableGroupStorage

---

Free the storage associated with a group of grid variables

#### Synopsis

```
C          int istat = CCTK_DisableGroupStorage( cGH * cctkGH, const char * group)
```

#### Parameters

cctkGH            pointer to CCTK grid hierarchy

**CCTK\_DisableGroupStorageI**

---

Deallocates memory for a group based upon its index

**Synopsis**

```
C          int  CCTK_DisableGroupStorageI(const cGH *GH, int group);
```

**Result**

0	The group previously had storage
1	The group did not have storage to disable storage
-1	The decrease storage routine was not overloaded

**Parameters**

GH	pointer to grid hierarchy
group	index of the group to deallocate storage for

**Discussion**

The disable group storage routine should deallocate memory for a group and return the previous status of that memory. This default function checks for the presence of the newer GroupStorageDecrease function, and if that is not available it flags an error. If it is available it makes a call to it, passing -1 as the timelevel argument, which is supposed to mean disable all timelevels, i.e. preserving this obsolete behaviour.

**CCTK\_EnableGroupComm**

---

Turn communications on for a group of grid variables

**Synopsis**

```
C          int istat = CCTK_EnableGroupComm( cGH * cctkGH, const char * group)
```

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy

**Discussion**

Grid variables with communication enabled will have their ghost zones communicated during a call to `CCTK_SyncGroup`. In general, this function does not need to be used, since communication is automatically enabled for grid variables who have assigned storage via the `schedule.ccl` file.

**CCTK\_EnableGroupCommI**

---

Turn communications on for a group of grid variables.

**Synopsis**

```
C          int istat = CCTK_EnableGroupCommI(cGH * cctkGH, int group);
```

**Result**

0 The Group has been enabled.

**Parameters**

`cctkGH` pointer to CCTK grid hierarchy  
`group` number of the group of grid variables to turn on

**Discussion**

Grid variables with communication enabled will have their ghost zones communicated during a call to `CCTK_SyncGroup`. In general, this function does not need to be used, since communication is automatically enabled for grid variables who have assigned storage via the `schedule.ccl` file.

**See Also**

`CCTK_DisableGroupComm` [\[A53\]](#) Turn communications off for a group of grid variables.  
`CCTK_DisableGroupCommI` [\[A54\]](#) Turn communications off for a group of grid variables.  
`CCTK_EnableGroupComm` [\[A58\]](#) Turn communications on for a group of grid variables.

**CCTK\_EnableGroupStorage**

---

Assign the storage for a group of grid variables

**Synopsis**

```
C          int istat = CCTK_EnableGroupStorage(cGH * cctkGH, const char * group);
```

**Result**

0 The Storage has been enabled.

**Parameters**

`cctkGH` pointer to CCTK grid hierarchy  
`group` name of the group to allocate storage for

**Discussion**

In general this function does not need to be used, since storage assignment is best handled by the Cactus scheduler via a thorn's `schedule.ccl` file.



**CCTK\_EnableGroupStorageI**

---

Assign the storage for a group of grid variables

**Synopsis**

**C**                    `int istat = CCTK_EnableGroupStorageI(cGH * cctkGH, int group);`

**Result**

0                    The Storage has been enabled.

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy  
`group`            Index of the group to allocate storage for

**Discussion**

In general this function does not need to be used, since storage assignment is best handled by the Cactus scheduler via a thorn's `schedule.ccl` file.

## CCTK\_Equals

---

Checks a STRING or KEYWORD parameter for equality with a given string

### Synopsis

**C** `#include "cctk.h"`  
`int status = CCTK_Equals(const char* parameter, const char* value)`

**Fortran** `integer status`  
`CCTK_POINTER parameter`  
`character*(*) value`  
`status = CCTK_Equals(parameter, value)`

### Result

1 if the parameter is (case-independently) equal to the specified value  
0 if the parameter is (case-independently) not equal to the specified value

### Parameters

**parameter** The string or keyword parameter to compare; Cactus represents this as a `CCTK_POINTER` pointing to the string value.  
**value** The value against which to compare the string or keyword parameter. This is typically a string literal (see the examples below).

### Discussion

This function compares a Cactus parameter of type STRING or KEYWORD against a given string value. The comparison is performed case-independently, returning a 1 if the strings are equal, and zero if they differ.

Note that in Fortran code, STRING or KEYWORD parameters are passed as C pointers, and can not be treated as normal Fortran strings. Thus `CCTK_Equals` should be used to check the value of such a parameter. See the examples below for typical usage.

### See Also

`Util.StrCmpi` [\[B14\]](#) compare two C-style strings case-independently

### Errors

**null pointer** If either argument is passed as a null pointer, `CCTK_Equals()` aborts the Cactus run with an error message. Otherwise, there are no error returns from this function.

### Examples

**C** `#include "cctk.h"`  
`#include "cctk_Arguments.h"`  
`#include "cctk_Parameters.h"`  
  
`/*`  
 `* assume this thorn has a string or keyword parameter my_parameter`

**Fortran**

```

    */
void MyThorn_some_function(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_PARAMETERS

    if (CCTK_Equals(my_parameter, "option A"))
    {
        CCTK_VInfo(CCTK_THORNSTRING, "using option A");
    }
}

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

c
c assume this thorn has a string or keyword parameter  my_parameter
c
subroutine MyThorn_some_function(CCTK_ARGUMENTS)
implicit none
DECLARE_CCTK_PARAMETERS

if (CCTK_Equals(my_parameter, "option A") .eq. 1) then
    call CCTK_INFO("using option A");
end if

return
end subroutine MyThorn_some_function

```

**CCTK\_Exit**

---

Exit the code cleanly

**Synopsis**

**C**                    `int istat = CCTK_Exit( cGH * cctkGH, int value)`

**Fortran**            `call CCTK_Exit(istat , cctkGH, value )`

`integer istat`  
                  `CCTK_POINTER cctkGH`  
                  `integer value`

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy

`value`            the return code to abort with

**Discussion**

This routine causes an immediate, regular termination of Cactus. It never returns to the caller.

**CCTK\_FirstVarIndex**

---

Given a group name, returns the first variable index in the group.

**Synopsis**

```
C          #include "cctk.h"
          int first_varindex = CCTK_FirstVarIndex(const char* group_name);

Fortran    #include "cctk.h"
          integer first_varindex
          character*(*) group_name
          call CCTK_FirstVarIndex(first_varindex, group_name)
```

**Result**

`first_varindex` ( $\geq 0$ )  
The first variable index in the group.

**Parameters**

`group_name` ( $\neq$  NULL in C)  
For C, this is a non-NULL pointer to the character-string name of the group. For Fortran, this is the character-string name of the group. In both cases this should be of the form "implementation::group".

**Discussion**

If the group contains  $N > 0$  variables, and  $V$  is the value of `first_varindex` returned by this function, then the group's variables have variable indices  $V, V + 1, V + 2, \dots, V + N - 1$ .

**See Also**

<code>CCTK_FirstVarIndexI()</code>	Given a group index, returns the first variable index in the group.
<code>CCTK_GroupData()</code>	Get “static” information about a group (including the number of variables in the group).
<code>CCTK_GroupDynamicData()</code>	Get “dynamic” information about a group.

**Errors**

-1	Group name is invalid.
-2	Group has no members.

**CCTK\_FirstVarIndexI**

---

Given a group index, returns the first variable index in the group.

**Synopsis**

```
C          #include "cctk.h"
            int first_varindex = CCTK_FirstVarIndexI(int group_index)

Fortran    #include "cctk.h"
            integer first_varindex, group_index
            call CCTK_FirstVarIndexI(first_varindex, group_index)
```

**Result**

`first_varindex` ( $\geq 0$ )  
The first variable index in the group.

**Parameters**

`group_index` ( $\geq 0$ )  
The group index, e.g. as returned by `CCTK_GroupIndex()`.

**Discussion**

If the group contains  $N > 0$  variables, and  $V$  is the value of `first_varindex` returned by this function, then the group’s variables have variable indices  $V, V + 1, V + 2, \dots, V + N - 1$ .

**See Also**

<code>CCTK_FirstVarIndex()</code>	Given a group name, returns the first variable index in the group.
<code>CCTK_GroupData()</code>	Get “static” information about a group (including the number of variables in the group).
<code>CCTK_GroupDynamicData()</code>	Get “dynamic” information about a group.

**Errors**

-1	Group index is invalid.
-2	Group has no members.

**CCTK\_FortranString**

---

Copy the contents of a C string into a Fortran string variable

**Synopsis**

```
C          #include "cctk.h"
          int CCTK_FortranString (char const * c_string,
                                char          * fortran_string,
                                int           fortran_length);

Fortran    #include "cctk.h"
          subroutine CCTK_FortranString (string_length, c_string, fortran_string)
             CCTK_INT                string_length
             CCTK_POINTER_TO_CONST    c_string
             character*(*)            fortran_string
          end subroutine
```

**Parameters**

**c\_string** This is (a pointer to) a standard C-style (NUL-terminated) string. Typically this argument is the name of a Cactus keyword or string parameter.

**fortran\_string** [This is an output argument] A Fortran character variable into which this function copies the C string (or as much of it as will fit).

**fortran\_length** The length of the Fortran character variable.

**Result**

**string\_length** This function sets this variable to the number of characters in the C string (not counting the terminating NUL character). If this is larger than the declared length of **fortran\_string** then the string was truncated. If this is negative, then an error occurred.

**Discussion**

String or keyword parameters in Cactus are passed into Fortran routines as pointers to C strings, which can't be directly used by Fortran code. This subroutine copies such a C string into a Fortran character\*N string variable, from where it can be used by Fortran code.

**Examples**

```
Fortran    # *** this is param.ccl for some thorn ***

          # This example shows how we can use a Cactus string parameter to
          # specify the contents of a Cactus key/value table, or the name of
          # a Fortran output file

          string our_parameters "parameter string"
          {
            ".*" :: "any string acceptable to Util_TableCreateFromString()"
          } "order=3"
```

```
string output_file_name "name of our output file"
{
  "." :: "any valid file name"
} "foo.dat"

c *** this is sample Fortran code in this same thorn ***
#include "util_Table.h"
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

      subroutine my_Fortran_subroutine(CCTK_ARGUMENTS)
      DECLARE_CCTK_ARGUMENTS
      DECLARE_CCTK_PARAMETERS

      CCTK_INT :: string_length
      integer :: status
      integer :: table_handle

      integer, parameter :: max_string_length = 500
      character*max_string_length :: our_parameters_fstring
      character*max_string_length :: output_file_name_fstring

c
c create Cactus key/value table from our_parameters parameter
c
      call CCTK_FortranString(string_length,
$                               our_parameters,
$                               our_parameters_fstring)
      if (string_length .gt. max_string_length) then
        call CCTK_WARN(CCTK_WARN_ALERT, "'our_parameters' string too long!")
      end if
      call Util_TableCreateFromString(table_handle, our_parameters_fstring)

c
c open a Fortran output file named via output_file_name parameter
c
      call CCTK_FortranString(string_length,
$                               output_file_name,
$                               output_file_name_fstring)
      if (string_length .gt. max_string_length) then
        call CCTK_WARN(CCTK_WARN_ALERT, "'output_file_name' string too long!")
      end if
      open (unit=9, iostat=status, status='replace',
$          file=output_file_name_fstring)
```



**CCTK\_FullName**

---

Given a variable index, returns the full name of the variable

**Synopsis**

```
C          char * fullname = CCTK_FullName( int index)
```

**Parameters**

**implementation**    The full variable name

**index**              The variable index

**Discussion**

The full variable name must be explicitly freed after it has been used.

No Fortran routine exists at the moment. The full variable name is in the form  
<implementation>::<variable>

**Examples**

```
C          index = CCTK_VarIndex("evolve::phi");  
          name = CCTK_FullName(index);  
          printf ("Variable name: %s", name);  
          free (name);
```

**CCTK\_GetClockName**

---

Given a pointer to the `cTimerVal` corresponding to a timer clock returns a pointer to a string that is the name of the clock

**Synopsis**

```
C          const char * CCTK_GetClockName(val)
```

**Parameters**

```
const cTimerVal * val  
                timer clock value pointer
```

**Discussion**

Do not attempt to free the returned pointer directly. You must use the string before calling `CCTK.TimerDestroyData` on the containing timer info.

**CCTK\_GetClockResolution**

---

Given a pointer to the `cTimerVal` corresponding to a timer clock returns the resolution of the clock in seconds.

**Synopsis**

```
C           double CCTK_GetClockResolution(val)
```

**Parameters**

```
const cTimerVal * val
                timer clock value pointer
```

**Discussion**

Ideally, the resolution should represent a good lower bound on the smallest non-zero difference between two consecutive calls of `CCTK_GetClockSeconds`. In practice, it is sometimes far smaller than it should be. Often it just represents the smallest value representable due to how the information is stored internally.

**CCTK\_GetClockSeconds**

---

Given a pointer to the `cTimerVal` corresponding to a timer clock returns a the elapsed time in seconds between the preceding `CCTK_TimerStart` and `CCTK_TimerStop` as recorded by the requested clock.

**Synopsis**

```
C          double CCTK_GetClockSeconds(val)
```

**Parameters**

```
const cTimerVal * val
                timer clock value pointer
```

**Discussion**

Be aware, different clocks measure different things (proper time, CPU time spent on this process, etc.), and have varying resolution and accuracy.

**CCTK\_GetClockValue**

---

Given a name of a clock that is in the given `cTimerData` structure, returns a pointer to the `cTimerVal` structure holding the clock's value.

**Synopsis**

```
C                const cTimerVal * CCTK_GetClockValue(name, info)
```

**Parameters**

```
const char * name           Name of clock
const cTimerData * info     Timer information structure containing clock.
```

**Discussion**

Do not attempt to free the returned pointer directly.

**Errors**

A                                null return value indicates an error.

**CCTK\_GetClockValueI**

---

Given a index of a clock that is in the given `cTimerData` structure, returns a pointer to the `cTimerVal` structure holding the clock's value.

**Synopsis**

```
C          const cTimerVal * CCTK_GetClockValue(index, info)
```

**Parameters**

```
int index      Index of clock
const cTimerData * info
                Timer information structure containing clock.
```

**Discussion**

Do not attempt to free the returned pointer directly.

**Errors**

A null return value indicates an error.

**CCTK\_GHExtension**

---

Get the pointer to a registered extension to the Cactus GH structure

**Synopsis**

```
C          void * extension = CCTK_GHExtension( const GH * cctkGH, const char * name)
```

**Parameters**

<code>extension</code>	The pointer to the GH extension
<code>cctkGH</code>	The pointer to the CCTK grid hierarchy
<code>name</code>	The name of the GH extension

**Discussion**

No Fortran routine exists at the moment.

**Examples**

```
C          void *extension = CCTK_GHExtension(GH, "myExtension");
```

**Errors**

NULL	A NULL pointer is returned if an invalid extension name was given.
------	--

**CCTK\_GHExtensionHandle**

---

Get the handle associated with a extension to the Cactus GH structure

**Synopsis**

<b>C</b>	<code>int handle = CCTK_GHExtensionHandle( const char * name)</code>
<b>Fortran</b>	<code>call CCTK_GHExtensionHandle(handle , name )</code>
	<code>integer handle</code>
	<code>character*(*) name</code>

**Parameters**

<code>handle</code>	The GH extension handle
<code>group</code>	The name of the GH extension

**Examples**

<b>C</b>	<code>handle = CCTK_GHExtension("myExtension") ;</code>
<b>Fortran</b>	<code>call CCTK_GHExtension(handle,"myExtension")</code>



## CCTK\_GridArrayReductionOperator

---

The name of the implementation of the registered grid array reduction operator, NULL if none is registered

### Synopsis

```
C          #include "cctk.h"

          const char *ga_reduc_imp = CCTK_GridArrayReductionOperator();
```

### Result

`ga_reduc_imp` Returns the name of the implementation of the registered grid array reduction operator or NULL if none is registered

### Discussion

We only allow one grid array reduction operator currently. This function can be used to check if any grid array reduction operator has been registered.

### See Also

<code>CCTK_ReduceGridArrays()</code>	Performs reduction on a list of distributed grid arrays
<code>CCTK_RegisterGridArrayReductionOperator()</code>	Registers a function as a grid array reduction operator of a certain name
<code>CCTK_NumGridArrayReductionOperators()</code>	The number of grid array reduction operators registered

CCTK\_GroupbboxGI, CCTK\_GroupbboxGN

---

Given a group index or name, return an array of the bounding box of the group for each face

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GroupbboxGI(const cGH *cctkGH,
                                         int dim,
                                         int *bbox,
                                         int groupindex);

          int status = CCTK_GroupbboxGN(const cGH *cctkGH,
                                         int dim,
                                         int *bbox,
                                         const char *groupname);

Fortran    call CCTK_GroupbboxGI(status, cctkGH, dim, bbox, groupindex)

          call CCTK_GroupbboxGN(status, cctkGH, dim, bbox, groupname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      bbox(dim)
          integer      groupindex
          character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid group index

### Parameters

status	Return value.
cctkGH ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim ( $\geq 1$ )	Number of dimensions of group.
bbox ( $\neq$ NULL)	Pointer to array which will hold the return values.
groupindex	Group index.
groupname	Group's full name.

### Discussion

The bounding box for a given group is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupbboxVI, CCTK\_GroupbboxVN

Returns the lower bounds for a given variable.

CCTK\_GroupbboxVI, CCTK\_GroupbboxVN

---

Given a variable index or name, return an array of the bounding box of the variable for each face

## Synopsis

```

C
#include "cctk.h"

int status = CCTK_GroupbboxVI(const cGH *cctkGH,
                             int dim,
                             int *bbox,
                             int varindex);

int status = CCTK_GroupbboxVN(const cGH *cctkGH,
                             int dim,
                             int *bbox,
                             const char *varname);

Fortran
call CCTK_GroupbboxVI(status, cctkGH, dim, bbox, varindex)

call CCTK_GroupbboxVN(status, cctkGH, dim, bbox, varname)

integer      status
CCTK_POINTER cctkGH
integer      dim
integer      bbox(dim)
integer      varindex
character(*) varname

```

## Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid variable index

## Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of variable.
<b>bbox</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Group index.
<b>varname</b>	Group's full name.

## Discussion

The bounding box for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupbboxGI, CCTK\_GroupbboxGN

Returns the upper bounds for a given group.

## CCTK\_GroupData

---

Given a group index, returns information about the group and its variables.

### Synopsis

```
C          #include "cctk.h"
          int status = CCTK_GroupData(int group_index, cGroup* group_data_buffer);
```

### Result

0                    success

### Parameters

`group_index`        The group index for which the information is desired.  
`group_data_buffer` ( $\neq$  NULL)    Pointer to a `cGroup` structure in which the information should be stored. See the "Discussion" section below for more information about this structure.

### Discussion

The `cGroup` structure<sup>1</sup> contains (at least) the following members:<sup>2</sup>

```
int grouptype;      /* group type, as returned by CCTK_GroupTypeNumber() */
int vartype;        /* variable type, as returned by CCTK_VarTypeNumber() */
int disttype;       /* distribution type, */
                   /* as returned by CCTK_GroupDistribNumber() */
int stagtype;       /* stagger type, as accepted by CCTK_StaggerDirName() */
int dim;            /* dimension (rank) of the group */
                   /* e.g. 3 for a group of 3-D variables */
int numvars;        /* number of variables in the group */
int numtimelevels;  /* maximum number of time levels for this group's variables */
int vectorgroup;    /* 1 if this is a vector group, 0 if it's not */
int vectorlength;   /* vector length of group */
                   /* (i.e. number of vector elements) */
                   /* (it is numvars = vectorlength * num_basevars, */
                   /* where num_basevars is the number of */
                   /* variables that have been given names in the */
                   /* interface.ccl) */
                   /* 1 if this isn't a vector group */
int tagstable;      /* handle to the group's tags table; */
                   /* this is a Cactus key-value table used to store */
                   /* metadata about the group and its variables, */
                   /* such as the variables' tensor types */
```

### See Also

"`interface.ccl`"                    Defines variables, groups, tags tables, and lots of other things.  
`CCTK_GroupDynamicData` [[A85](#)]       Gets grid-size information for a group's variables.  
`CCTK_GroupIndex` [[A91](#)]               Gets the group index for a given group name.

---

<sup>1</sup>`cGroup` is a `typedef` for a structure. It's defined in "`cctk.Group.h`", which is `#included` by "`cctk.h`".

<sup>2</sup>Note that the members are **not** guaranteed to be declared in the order listed here.

<code>CCTK_GroupIndexFromVar</code> [ <a href="#">A92</a> ]	Gets the group index for a given variable name.
<code>CCTK_GroupName</code> [ <a href="#">A102</a> ]	Gets the group name for a given group index.
<code>CCTK_GroupNameFromVarI</code> [ <a href="#">A103</a> ]	Gets the group name for a given variable name.
<code>CCTK_GroupTypeI</code> [ <a href="#">A114</a> ]	Gets a group type index for a given group index.
<code>CCTK_GroupTypeFromVarI</code> [ <a href="#">A113</a> ]	Gets a group type index for a given variable index.

### Errors

-1	<code>group_index</code> is invalid.
-2	<code>group_data.buffer</code> is NULL.

### Examples

```
C      #include <stdio.h>
      #include "cctk.h"

      cGroup group_info;
      int group_index, status;

      group_index = CCTK_GroupIndex("BSSN_MoL::ADM_BSSN_metric");
      if (group_index < 0)
          CCTK_VWarn(CCTK_WARN_ABORT,
"error return %d trying to get BSSN metric's group index!",
                    group_index);                                /*NOTREACHED*/

      status = CCTK_GroupData(group_index, &group_info);
      if (status < 0)
          CCTK_VWarn(CCTK_WARN_ABORT,
"error return %d trying to get BSSN metric's group information!",
                    status);                                    /*NOTREACHED*/

      printf("this group's arrays are %-dimensional and have %d time levels\n",
            group_info.dim, group_info.numtimelevels);
```

**CCTK\_GroupDimFromVarI**

---

Given a variable index, returns the dimension of all variables in the corresponding group.

**Synopsis**

**C**                    `#include "cctk.h"`

`int dim = CCTK_GroupDimFromVarI(int varindex);`

**Fortran**            `call CCTK_GroupDimFromVarI(dim, varindex)`

**Result**

positive            the dimension of the group

-1                    invalid variable index

**Parameters**

**varindex**           Variable index

**Discussion**

The dimension of all variables in a group associated with the given variable is returned.

**See Also**

**CCTK\_GroupDimI**                    Returns the dimension for a given group



**CCTK\_GroupDimI**

---

Given a group index, returns the dimension of that group.

**Synopsis**

**C**                    `#include "cctk.h"`  
  
                      `int dim = CCTK_GroupDimI(int groupindex);`  
**Fortran**            `call CCTK_GroupDimI(dim, groupindex)`

**Result**

positive            the dimension of the group  
-1                   invalid group index

**Parameters**

groupindex        Group index

**Discussion**

The dimension of variables in the given group is returned.

**See Also**

CCTK\_GroupDimFromVarI        Returns the dimension for a group given by a member variable index

**CCTK\_GroupDynamicData**

---

Returns the driver's internal data for a given group

**Synopsis**

```
C          #include "cctk.h"
          int retval = CCTK_GroupDynamicData (const cGH *GH, int group, cGroupDynamicData *data);
```

**Result**

0	Success
-1	the given pointer to the data structure data is null
-3	the givenGH pointer is invalid
-77	the requested group has zero variables

**Parameters**

GH	a valid initialized GH structure for your driver
group	the index of the group you're interested in
data	a pointer to a caller-supplied data structure to store the group data

**Discussion**

This function returns information about the given grid hierarchy. The data structure used to store the information in is of type `cGroupDynamicData`. The members of this structure that are set are :

- dim: The number of dimensions in this grid hierarchy.
- lsh: The local(on this processor) size.
- gsh: The global grid size.
- lbnd: An array of integers containing the lowest index of the local grid as seen on the global grid.(These use zero based indexing)
- ubnd: An array of integers containing the largest index of the local grid as seen on the global grid.(These use zero based indexing)
- nghostzones: An array of integers with the number of ghostzones for each dimension.
- bbox: An array of integers containing which indicate whether the boundaries are internal(e.g. artificial boundaries between processors) or actual physical boundaries. A value of 1 indicates an actual physical boundary while a 0 indicates an internal one.
- activetimelevels: An array of which time levels this grid hierarchy is active.

-

**CCTK\_GroupGhostsizesI**

---

Given a group index, return a pointer to an array containing the ghost sizes of the group in each dimension.

**Synopsis**

```
C          #include "cctk.h"

          CCTK_INT **ghostsizes = CCTK_GroupGhostsizesI(int groupindex);
```

**Result**

non-NULL	a pointer to the ghost size array
NULL	invalid group index

**Parameters**

groupindex	Group index
------------	-------------

**Discussion**

The ghost sizes in each dimension for a given group are returned as a pointer reference.

**See Also**

CCTK_GroupDimI	Returns the dimension for a group.
CCTK_GroupSizesI	Returns the size arrays for a group.

CCTK\_GroupgshGI, CCTK\_GroupgshGN

---

Given a group index or name, return an array of the global size of the group in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GroupgshGI(const cGH *cctkGH,
                                      int dim,
                                      int *gsh,
                                      int groupindex);

          int status = CCTK_GroupgshGN(const cGH *cctkGH,
                                      int dim,
                                      int *gsh,
                                      const char *groupname);

Fortran    call CCTK_GroupgshGI(status, cctkGH, dim, gsh, groupindex)

          call CCTK_GroupgshGN(status, cctkGH, dim, gsh, groupname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      gsh(dim)
          integer      groupindex
          character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid group name

### Parameters

cctkGH ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim ( $\geq 1$ )	Number of dimensions of group.
gsh ( $\neq$ NULL)	Pointer to array which will hold the return values.
groupindex	Index of the group.
groupname	Name of the group.

### Discussion

The global size in each dimension for a given group is returned in a user-supplied array buffer.

### See Also

CCTK\_GroupgshVI, CCTK\_GroupgshVN  
Returns the global size for a given variable.

CCTK\_GrouplshGI, CCTK\_GrouplshGN  
Returns the local size for a given group.

CCTK\_GrouplshVI, CCTK\_GrouplshVN  
Returns the local size for a given variable.

CCTK\_GroupgshVI, CCTK\_GroupgshVN

---

Given a variable index or its full name, return an array of the global size of the variable in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GroupgshVI(const cGH *cctkGH,
                                      int dim,
                                      int *gsh,
                                      int varindex);

          int status = CCTK_GroupgshVN(const cGH *cctkGH,
                                      int dim,
                                      int *gsh,
                                      const char *varname);

Fortran    call CCTK_GroupgshVI(status, cctkGH, dim, gsh, varindex)

          call CCTK_GroupgshVN(status, cctkGH, dim, gsh, varname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      gsh(dim)
          integer      varindex
          character*(*) varname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid variable index

### Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of variable.
<b>gsh</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Variable index.
<b>varname</b>	Variable's full name.

### Discussion

The global size in each dimension for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupgshGI, CCTK\_GroupgshGN  
Returns the global size for a given group.

CCTK\_GrouplshGI, CCTK\_GrouplshGN  
Returns the local size for a given group.

CCTK\_GrouplshVI, CCTK\_GrouplshVN  
Returns the local size for a given variable.

**CCTK\_GroupIndex**

---

Get the index number for a group name

**Synopsis**

**C**                    `int index = CCTK_GroupIndex( const char * groupname)`

**Fortran**            `call CCTK_GroupIndex(index , groupname )`

integer index  
character\*(\*) groupname

**Parameters**

**groupname**            The name of the group

**Discussion**

The group name should be the given in its fully qualified form, that is `<implementation>::<group>` for a public or protected group, and `<thornname>::<group>` for a private group.

**Examples**

**C**                    `index = CCTK_GroupIndex("evolve::scalars");`

**Fortran**            `call CCTK_GroupIndex(index,"evolve::scalars")`



---

**CCTK\_GroupIndexFromVar**

---

Given a variable name, returns the index of the associated group

**Synopsis**

**C**                    `int groupindex = CCTK_GroupIndexFromVar( const char * name)`

**Fortran**            `call CCTK_GroupIndexFromVar(groupindex , name )`

integer groupindex  
character\*(\*) name

**Parameters**

groupindex        The index of the group

name              The full name of the variable

**Discussion**

The variable name should be in the form <implementation>::<variable>

**Examples**

**C**                    `groupindex = CCTK_GroupIndexFromVar("evolve::phi") ;`

**Fortran**            `call CCTK_GROUPINDEXFROMVAR(groupindex,"evolve::phi")`

**CCTK\_GroupIndexFromVarI**

---

Given a variable index, returns the index of the associated group

**Synopsis**

**C**                    `int groupindex = CCTK_GroupIndexFromVarI( int varindex)`  
**Fortran**            `call CCTK_GroupIndexFromVarI(groupindex , varindex )`

`integer groupindex`  
                     `integer varindex`

**Parameters**

`groupindex`        The index of the group  
`varindex`          The index of the variable

**Examples**

**C**                    `index = CCTK_VarIndex("evolve::phi");`  
                     `groupindex = CCTK_GroupIndexFromVarI(index);`

**Fortran**            `call CCTK_VARINDEX("evolve::phi")`  
                     `CCTK_GROUPINDEXFROMVARI(groupindex,index)`

CCTK\_Group1bndGI, CCTK\_Group1bndGN

---

Given a group index or name, return an array of the lower bounds of the group in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_Group1bndGI(const cGH *cctkGH,
                                         int dim,
                                         int *lbnd,
                                         int groupindex);

          int status = CCTK_Group1bndGN(const cGH *cctkGH,
                                         int dim,
                                         int *lbnd,
                                         const char *groupname);

Fortran    call CCTK_Group1bndGI(status, cctkGH, dim, lbnd, groupindex)

          call CCTK_Group1bndGN(status, cctkGH, dim, lbnd, groupname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      lbnd(dim)
          integer      groupindex
          character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid group index

### Parameters

<b>status</b>	Return value.
<b>cctkGH (<math>\neq</math> NULL)</b>	Pointer to a valid Cactus grid hierarchy.
<b>dim (<math>\geq 1</math>)</b>	Number of dimensions of group.
<b>lbnd (<math>\neq</math> NULL)</b>	Pointer to array which will hold the return values.
<b>groupindex</b>	Group index.
<b>groupname</b>	Group's full name.

### Discussion

The lower bounds in each dimension for a given group is returned in a user-supplied array buffer.

**See Also**

CCTK\_Group1bndVI, CCTK\_Group1bndVN

Returns the lower bounds for a given variable.

CCTK\_GroupubndGI, CCTK\_GroupubndGN

Returns the upper bounds for a given group.

CCTK\_GroupubndVI, CCTK\_GroupubndVN

Returns the upper bounds for a given variable.

CCTK\_GrouplbndVI, CCTK\_GrouplbndVN

---

Given a variable index or name, return an array of the lower bounds of the variable in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GrouplbndVI(const cGH *cctkGH,
                                         int dim,
                                         int *lbnd,
                                         int varindex);

          int status = CCTK_GrouplbndVN(const cGH *cctkGH,
                                         int dim,
                                         int *lbnd,
                                         const char *varname);

Fortran    call CCTK_GrouplbndVI(status, cctkGH, dim, lbnd, varindex)

          call CCTK_GrouplbndVN(status, cctkGH, dim, lbnd, varname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      lbnd(dim)
          integer      varindex
          character(*) varname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid variable index

### Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of variable.
<b>lbnd</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Group index.
<b>varname</b>	Group's full name.

### Discussion

The lower bounds in each dimension for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_Group1bndGI, CCTK\_Group1bndGN

Returns the lower bounds for a given group.

CCTK\_GroupubndGI, CCTK\_GroupubndGN

Returns the upper bounds for a given group.

CCTK\_GroupubndVI, CCTK\_GroupubndVN

Returns the upper bounds for a given variable.

CCTK\_Group1shGI, CCTK\_Group1shGN

---

Given a group index or name, return an array of the local size of the group in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_Group1shGI(const cGH *cctkGH,
                                      int dim,
                                      int *lsh,
                                      int groupindex);

          int status = CCTK_Group1shGN(const cGH *cctkGH,
                                      int dim,
                                      int *lsh,
                                      const char *groupname);

Fortran    call CCTK_Group1shGI(status, cctkGH, dim, lsh, groupindex)

          call CCTK_Group1shGN(status, cctkGH, dim, lsh, groupname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      lsh(dim)
          integer      groupindex
          character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid group name

### Parameters

cctkGH ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim ( $\geq 1$ )	Number of dimensions of group.
lsh ( $\neq$ NULL)	Pointer to array which will hold the return values.
groupindex	Index of the group.
groupname	Name of the group.

### Discussion

The local size in each dimension for a given group is returned in a user-supplied array buffer.

### See Also

CCTK\_GroupgshVI, CCTK\_GroupgshVN  
Returns the global size for a given variable.

CCTK\_GroupplshGI, CCTK\_GroupplshGN  
Returns the local size for a given group.

CCTK\_GroupplshVI, CCTK\_GroupplshVN  
Returns the local size for a given variable.



CCTK\_Group1shVI, CCTK\_Group1shVN

---

Given a variable index or its full name, return an array of the local size of the variable in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_Group1shVI(const cGH *cctkGH,
                                      int dim,
                                      int *lsh,
                                      int varindex);

          int status = CCTK_Group1shVN(const cGH *cctkGH,
                                      int dim,
                                      int *lsh,
                                      const char *varname);

Fortran    call CCTK_Group1shVI(status, cctkGH, dim, lsh, varindex)

          call CCTK_Group1shVN(status, cctkGH, dim, lsh, varname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      lsh(dim)
          integer      varindex
          character(*) varname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid variable index

### Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of variable.
<b>lsh</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Variable index.
<b>varname</b>	Variable's full name.

### Discussion

The local size in each dimension for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupgshGI, CCTK\_GroupgshGN

Returns the global size for a given group.

CCTK\_GroupgshVI, CCTK\_GroupgshVN

Returns the global size for a given variable.

CCTK\_GrouplshGI, CCTK\_GrouplshGN

Returns the local size for a given group.

**CCTK\_GroupName**

---

Given a group index, returns the group name

**Synopsis**

```
C          char * name = CCTK_GroupName( int index)
```

**Parameters**

<code>name</code>	The group name
<code>index</code>	The group index

**Discussion**

The group name must be explicitly freed after it has been used.  
No Fortran routine exists at the moment.

**Examples**

```
C          index = CCTK_GroupIndex("evolve::scalars");  
          name = CCTK_GroupName(index);  
          printf ("Group name: %s", name);  
          free (name);
```

**CCTK\_GroupNameFromVarI**

---

Given a variable index, return the name of the associated group

**Synopsis**

```
C          char * group = CCTK_GroupNameFromVarI( int varindex)
```

**Parameters**

<code>group</code>	The name of the group
<code>varindex</code>	The index of the variable

**Examples**

```
C          index = CCTK_VarIndex("evolve::phi");  
          group = CCTK_GroupNameFromVarI(index) ;
```

CCTK\_GroupnghostzonesGI, CCTK\_GroupnghostzonesGN

---

Given a group index or name, return an array with the number of ghostzones in each dimension of the group

### Synopsis

**C**

```
#include "cctk.h"

int status = CCTK_GroupnghostzonesGI(const cGH *cctkGH,
                                     int dim,
                                     int *nghostzones,
                                     int groupindex)

int status = CCTK_GroupnghostzonesGN(const cGH *cctkGH,
                                     int dim,
                                     int *nghostzones,
                                     const char *groupname)
```

**Fortran**

```
call CCTK_GroupnghostzonesGI(status, cctkGH, dim, nghostzones, groupindex)

call CCTK_GroupnghostzonesGN(status, cctkGH, dim, nghostzones, groupname)

integer      status
CCTK_POINTER cctkGH
integer      dim
integer      nghostzones(dim)
integer      groupindex
character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group

### Parameters

<b>status</b>	Return value.
<b>cctkGH (<math>\neq</math> NULL)</b>	Pointer to a valid Cactus grid hierarchy.
<b>dim (<math>\geq 1</math>)</b>	Number of dimensions of group.
<b>nghostzones (<math>\neq</math> NULL)</b>	Pointer to array which will hold the return values.
<b>groupindex</b>	Group index.
<b>groupname</b>	Group name.

### Discussion

The number of ghostzones in each dimension for a given group is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupnghostzonesVI, CCTK\_GroupnghostzonesVN

Returns the number of ghostzones for a given variable.

CCTK\_GroupnghostzonesVI, CCTK\_GroupnghostzonesVN

---

Given a variable index or its full name, return an array with the number of ghostzones in each dimension of the variable

### Synopsis

```

C                #include "cctk.h"

                    int status = CCTK_GroupnghostzonesVI(const cGH *cctkGH,
                                                            int dim,
                                                            int *nghostzones,
                                                            int varindex)

                    int status = CCTK_GroupnghostzonesVN(const cGH *cctkGH,
                                                            int dim,
                                                            int *nghostzones,
                                                            const char *varname)

Fortran          call CCTK_GroupnghostzonesVI(status, cctkGH, dim, nghostzones, varindex)

                    call CCTK_GroupnghostzonesVN(status, cctkGH, dim, nghostzones, varname)

                    integer          status
                    CCTK_POINTER    cctkGH
                    integer          dim
                    integer          nghostzones(dim)
                    integer          varindex
                    character*(*)    varname

```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group

### Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of group.
<b>nghostzones</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Variable index.
<b>varname</b>	Variable's full name.

### Discussion

The number of ghostzones in each dimension for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_GroupnghostzonesGI, CCTK\_GroupnghostzonesGN

Returns the number of ghostzones for a given group.



**CCTK\_GroupSizesI**

---

Given a group index, return a pointer to an array containing the sizes of the group in each dimension.

**Synopsis**

```
C          #include "cctk.h"

          CCTK_INT **ghostsizes = CCTK_GroupSizesI(int groupindex);
```

**Result**

non-NULL	a pointer to the size array
NULL	invalid group index

**Parameters**

groupindex	Group index
------------	-------------

**Discussion**

The sizes in each dimension for a given group are returned as a pointer reference.

**See Also**

CCTK_GroupDimI	Returns the dimension for a group.
CCTK_GroupGhostsizesI	Returns the size arrays for a group.

**CCTK\_GroupStorageDecrease**

---

Decrease the number of timelevels allocated for the given variable groups.

**Synopsis**

**C** `int numTL = CactusDefaultGroupStorageDecrease (const cGH *GH, int n_groups, const int *`

**Result**

The new total number of timelevels with storage enabled for all groups queried or modified.

**Parameters**

<b>GH</b>	pointer to grid hierarchy
<b>n_groups</b>	Number of groups
<b>groups</b>	list of group indices to reduce storage for
<b>timelevels</b>	number of time levels to reduce storage for for each group
<b>groups</b>	list of group indices to allocate storage for
<b>status</b>	optional return array which, if not NULL, will, on return, contain the number of timelevels which were previously allocated storage for each group

**Discussion**

The decrease group storage routine decreases the memory allocated to the specified number of timelevels for each listed group, returning the previous number of timelevels enabled for that group in the status array, if that is not NULL. It never increases the number of timelevels enabled, i.e., if it is asked to reduce to more timelevels than are enabled, it does not change the storage for that group.

There is a default implementation which checks for the presence of the older Disable-GroupStorage function, and if that is not available it flags an error. If it is available it makes a call to it, and puts its return value in the status flag for the group. Usually, a driver has overloaded the default implementation.

A driver should replace the appropriate GV pointers on the cGH structure when it changes the storage state of a GV.

**CCTK\_GroupStorageIncrease**

---

Increases the number of timelevels allocated for the given variable groups.

**Synopsis**

**C** `int numTL = CactusDefaultGroupStorageIncrease (const cGH *GH, int n_groups, const int *`

**Result**

The new total number of timelevels with storage enabled for all groups queried or modified.

**Parameters**

<b>GH</b>	pointer to grid hierarchy
<b>n_groups</b>	Number of groups
<b>groups</b>	list of group indices to allocate storage for
<b>timelevels</b>	number of time levels to allocate storage for for each group
<b>groups</b>	list of group indices to allocate storage for
<b>status</b>	optional return array which, if not NULL, will, on return, contain the number of timelevels which were previously allocated storage for each group

**Discussion**

The increase group storage routine increases the allocated memory to the specified number of timelevels of each listed group, returning the previous number of timelevels enabled for that group in the status array, if that is not NULL. It never decreases the number of timelevels enabled, i.e., if it is asked to enable less timelevels than are already enabled it does not change the storage for that group.

There is a default implementation which checks for the presence of the older EnableGroupStorage function, and if that is not available it flags an error. If it is available it makes a call to it, and puts its return value in the status flag for the group. Usually, a driver has overloaded the default implementation.

A driver should replace the appropriate GV pointers on the cGH structure when it changes the storage state of a GV.

**CCTK\_GroupTagsTable**

---

Given a group name, return the table handle of the group's tags table.

**Synopsis**

```
C          #include "cctk.h"
            int table_handle = CCTK_GroupTagsTable(const char* group_name);

Fortran    #include "cctk.h"
            integer table_handle
            character*(*) group_name
            call CCTK_VarIndex(table_handle, group_name)
```

**Result**

**table\_handle**      The table handle of the group's tags table.

**Parameters**

**group\_name**      The character-string name of group. This should be given in its fully qualified form, that is `implementation::group_name` or `thorn_name::group_name`.

**See Also**

**CCTK\_GroupData** [\[A81\]](#)      This function returns a variety of “static” information about a group (“static” in the sense that it doesn’t change during a Cactus run).

**CCTK\_GroupDynamicData** [\[A85\]](#)      This function returns a variety of “dynamic” information about a group (“dynamic” in the sense that a driver can (and often does) change this information during a Cactus run).

**Errors**

-1      no group exists with the specified name

**CCTK\_GroupTagsTableI**

---

Given a group name, return the table handle of the group's tags table.

**Synopsis**

```
C          #include "cctk.h"
            int table_handle = CCTK_GroupTagsTableI(int group_index);

Fortran    #include "cctk.h"
            integer table_handle
            integer group_index
            call CCTK_VarIndex(table_handle, group_index)
```

**Result**

**table\_handle**      The table handle of the group's tags table.

**Parameters**

**group\_index**      The group index of the group.

**See Also**

<b>CCTK_GroupData</b> <a href="#">[A81]</a>	This function returns a variety of “static” information about a group (“static” in the sense that it doesn’t change during a Cactus run).
<b>CCTK_GroupDynamicData</b> <a href="#">[A85]</a>	This function returns a variety of “dynamic” information about a group (“dynamic” in the sense that a driver can (and often does) change this information during a Cactus run).
<b>CCTK_GroupIndex</b> <a href="#">[A91]</a>	Get the group index for a specified group name.
<b>CCTK_GroupIndexFromVar</b> <a href="#">[A92]</a>	Get the group index for the group containing the variable with a specified name.
<b>CCTK_GroupIndexFromVarI</b> <a href="#">[A93]</a>	Get the group index for the group containing the variable with a specified variable index.

**Errors**

-1                      no group exists with the specified name

**CCTK\_GroupTypeFromVarI**

---

Provides a group's group type index given a variable index

**Synopsis**

**C**                    `int type = CCTK_GroupTypeFromVarI( int index)`

**Fortran**           `call CCTK_GroupTypeFromVarI(type , index )`

`integer type`

`integer index`

**Parameters**

`type`                The group's group type index

`group`              The variable index

**Discussion**

The group's group type index indicates the type of variables in the group. Either scalars, grid functions or arrays. The group type can be checked with the Cactus provided macros for `CCTK_SCALAR`, `CCTK_GF`, `CCTK_ARRAY`.

**Examples**

**C**                    `index = CCTK_GroupIndex("evolve::scalars")`  
                      `array = (CCTK_ARRAY == CCTK_GroupTypeFromVarI(index));`

**Fortran**           `call CCTK_GROUPTYPEFROMVARI(type,3)`

**CCTK\_GroupTypeI**

---

Provides a group's group type index given a group index

**Synopsis**

```
C          #include "cctk.h"
          int group_type = CCTK_GroupTypeI(int group);
```

**Result**

-1                    -1 is returned if the given group index is invalid.

**Parameters**

group                Group index.

**Discussion**

A group's group type index indicates the type of variables in the group. The three group types are scalars, grid functions, and grid arrays. The group type can be checked with the Cactus provided macros for `CCTK_SCALAR`, `CCTK_GF`, `CCTK_ARRAY`.

**See Also**

`CCTK_GroupTypeFromVarI` [[A113](#)]    This function takes a variable index rather than a group index as its argument.

CCTK\_GroupubndGI, CCTK\_GroupubndGN

---

Given a group index or name, return an array of the upper bounds of the group in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GroupubndGI(const cGH *cctkGH,
                                         int dim,
                                         int *ubnd,
                                         int groupindex);

          int status = CCTK_GroupubndGN(const cGH *cctkGH,
                                         int dim,
                                         int *ubnd,
                                         const char *groupname);

Fortran    call CCTK_GroupubndGI(status, cctkGH, dim, ubnd, groupindex)

          call CCTK_GroupubndGN(status, cctkGH, dim, ubnd, groupname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      ubnd(dim)
          integer      groupindex
          character(*) groupname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid group index

### Parameters

<b>status</b>	Return value.
<b>cctkGH (<math>\neq</math> NULL)</b>	Pointer to a valid Cactus grid hierarchy.
<b>dim (<math>\geq 1</math>)</b>	Number of dimensions of group.
<b>ubnd (<math>\neq</math> NULL)</b>	Pointer to array which will hold the return values.
<b>groupindex</b>	Group index.
<b>groupname</b>	Group's full name.

### Discussion

The upper bounds in each dimension for a given group is returned in a user-supplied array buffer.



**See Also**

CCTK\_Group1bndGI, CCTK\_Group1bndGN

Returns the lower bounds for a given group.

CCTK\_Group1bndVI, CCTK\_Group1bndVN

Returns the lower bounds for a given variable.

CCTK\_GroupubndVI, CCTK\_GroupubndVN

Returns the upper bounds for a given variable.

CCTK\_GroupubndVI, CCTK\_GroupubndVN

---

Given a variable index or name, return an array of the upper bounds of the variable in each dimension

### Synopsis

```
C          #include "cctk.h"

          int status = CCTK_GroupubndVI(const cGH *cctkGH,
                                         int dim,
                                         int *ubnd,
                                         int varindex);

          int status = CCTK_GroupubndVN(const cGH *cctkGH,
                                         int dim,
                                         int *ubnd,
                                         const char *varname);

Fortran    call CCTK_GroupubndVI(status, cctkGH, dim, ubnd, varindex)

          call CCTK_GroupubndVN(status, cctkGH, dim, ubnd, varname)

          integer      status
          CCTK_POINTER cctkGH
          integer      dim
          integer      ubnd(dim)
          integer      varindex
          character(*) varname
```

### Result

0	success
-1	incorrect dimension supplied
-2	data not available from driver
-3	called on a scalar group
-4	invalid variable index

### Parameters

<b>status</b>	Return value.
<b>cctkGH</b> ( $\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
<b>dim</b> ( $\geq 1$ )	Number of dimensions of variable.
<b>ubnd</b> ( $\neq$ NULL)	Pointer to array which will hold the return values.
<b>varindex</b>	Group index.
<b>varname</b>	Group's full name.

### Discussion

The upper bounds in each dimension for a given variable is returned in a user-supplied array buffer.

**See Also**

CCTK\_Group1bndGI, CCTK\_Group1bndGN

Returns the lower bounds for a given group.

CCTK\_Group1bndVI, CCTK\_Group1bndVN

Returns the lower bounds for a given variable.

CCTK\_GroupubndGI, CCTK\_GroupubndGN

Returns the upper bounds for a given group.

**CCTK\_ImpFromVarI**

---

Given a variable index, returns the implementation name

**Synopsis**

```
C          char * implementation = CCTK_ImpFromVarI( int index)
```

**Parameters**

<code>implementation</code>	The implementation name
<code>index</code>	The variable index

**Discussion**

No Fortran routine exists at the moment

**Examples**

```
C          index = CCTK_VarIndex("evolve::phi");  
          implementation = CCTK_ImpFromVarI(index);
```

**CCTK\_ImplementationRequires**

---

Return the ancestors for an implementation.

**Synopsis**

```
C          #include "cctk.h"

          uStringList *imps = CCTK_ImplementationRequires(const char *imp);
```

**Result**

imps (not documented)

**Parameters**

imp (not documented)

**See Also**

CCTK_ActivatingThorn <a href="#">[A16]</a>	Finds the thorn which activated a particular implementation
CCTK_CompiledImplementation <a href="#">[A40]</a>	Return the name of the compiled implementation with given index
CCTK_CompiledThorn <a href="#">[A41]</a>	Return the name of the compiled thorn with given index
CCTK_ImplementationThorn <a href="#">[A121]</a>	Returns the name of one thorn providing an implementation.
CCTK_ImpThornList <a href="#">[A122]</a>	Return the thorns for an implementation
CCTK_IsImplementationActive <a href="#">[A142]</a>	Reports whether an implementation was activated in a parameter file
CCTK_IsImplementationCompiled <a href="#">[A143]</a>	Reports whether an implementation was compiled into a configuration
CCTK_IsThornActive <a href="#">[A144]</a>	Reports whether a thorn was activated in a parameter file
CCTK_IsThornCompiled <a href="#">[A145]</a>	Reports whether a thorn was compiled into a configuration
CCTK_NumCompiledImplementations <a href="#">[A159]</a>	Return the number of implementations compiled in
CCTK_NumCompiledThorns <a href="#">[A160]</a>	Return the number of thorns compiled in
CCTK_ThornImplementation <a href="#">[A236]</a>	Returns the implementation provided by the thorn

**Errors**

(not documented)

**CCTK\_ImplementationThorn**

---

Returns the name of one thorn providing an implementation.

**Synopsis**

```
C          #include "cctk.h"

          const char *thorn = CCTK_ImplementationThorn(const char *name);
```

**Result**

thorn            Name of the thorn or NULL

**Parameters**

name            Name of the implementation

**See Also**

CCTK_ActivatingThorn	<a href="#">[A16]</a>	Finds the thorn which activated a particular implementation
CCTK_CompiledImplementation	<a href="#">[A40]</a>	Return the name of the compiled implementation with given index
CCTK_CompiledThorn	<a href="#">[A41]</a>	Return the name of the compiled thorn with given index
CCTK_ImplementationRequires	<a href="#">[A120]</a>	Return the ancestors for an implementation
CCTK_ImpThornList	<a href="#">[A122]</a>	Return the thorns for an implementation
CCTK_IsImplementationActive	<a href="#">[A142]</a>	Reports whether an implementation was activated in a parameter file
CCTK_IsImplementationCompiled	<a href="#">[A143]</a>	Reports whether an implementation was compiled into a configuration
CCTK_IsThornActive	<a href="#">[A144]</a>	Reports whether a thorn was activated in a parameter file
CCTK_IsThornCompiled	<a href="#">[A145]</a>	Reports whether a thorn was compiled into a configuration
CCTK_NumCompiledImplementations	<a href="#">[A159]</a>	Return the number of implementations compiled in
CCTK_NumCompiledThorns	<a href="#">[A160]</a>	Return the number of thorns compiled in
CCTK_ThornImplementation	<a href="#">[A236]</a>	Returns the implementation provided by the thorn

**Errors**

NULL            Error.

**CCTK\_ImpThornList**

---

Return the thorns for an implementation.

**Synopsis**

```
C          #include "cctk.h"

          t_sktree *thorns = CCTK_ImpThornList(const char *name);
```

**Result**

thorns (not documented)

**Parameters**

name Name of implementation

**Discussion**

(not documented)

**See Also**

CCTK\_ActivatingThorn [\[A16\]](#) Finds the thorn which activated a particular implementation

CCTK\_CompiledImplementation [\[A40\]](#) Return the name of the compiled implementation with given index

CCTK\_CompiledThorn [\[A41\]](#) Return the name of the compiled thorn with given index

CCTK\_ImplementationRequires [\[A120\]](#) Return the ancestors for an implementation

CCTK\_ImplementationThorn [\[A121\]](#) Returns the name of one thorn providing an implementation.

CCTK\_IsImplementationActive [\[A142\]](#) Reports whether an implementation was activated in a parameter file

CCTK\_IsImplementationCompiled [\[A143\]](#) Reports whether an implementation was compiled into a configuration

CCTK\_IsThornActive [\[A144\]](#) Reports whether a thorn was activated in a parameter file

CCTK\_IsThornCompiled [\[A145\]](#) Reports whether a thorn was compiled into a configuration

CCTK\_NumCompiledImplementations [\[A159\]](#) Return the number of implementations compiled in

CCTK\_NumCompiledThorns [\[A160\]](#) Return the number of thorns compiled in

CCTK\_ThornImplementation [\[A236\]](#) Returns the implementation provided by the thorn

**Errors**

(not documented)

**CCTK\_INFO**

---

Macro to print a single string as an information message to screen

**Synopsis**

**C**                    `#include "cctk.h"`  
                      `#include "cctk_WarnLevel.h"`  
  
                      `CCTK_INFO(const char *message);`  
  
**Fortran**            `#include "cctk.h"`  
  
                      `call CCTK_INFO(message)`  
                      `character*(*) message`

**Parameters**

**message**            The string to print as an info message

**Discussion**

This macro can be used by thorns to print a single string as an info message to screen. The macro `CCTK_INFO(message)` expands to a call to the underlying function `CCTK_Info`:  
`CCTK_Info(CCTK_THORNSTRING, message)`  
So the macro automatically includes the name of the originating thorn in the info message. It is recommended that the macro `CCTK_INFO` is used to print a message rather than calling `CCTK_Info` directly.  
To include variables in an info message from C, you can use the routine `CCTK_VInfo` which accepts a variable argument list. To include variables from Fortran, a string must be constructed and passed in a `CCTK_INFO` macro.

**See Also**

`CCTK_VInfo()`                    prints a formatted string with a variable argument list as an info message to screen

**Examples**

**C**                    `#include "cctk.h"`  
                      `#include "cctk_WarningLevel.h"`  
  
                      `CCTK_INFO("Output is disabled");`  
  
**Fortran**            `#include "cctk.h"`  
  
                      `integer            myint`  
                      `real               myreal`  
                      `character*200 message`  
  
                      `write(message, '(A32, G12.7, A5, I8)')`  
                      `&        'Your info message, including ', myreal, ' and ', myint`  
                      `call CCTK_INFO(message)`



## CCTK\_InfoCallbackRegister

---

Register one or more routines for dealing with information messages in addition to printing them to screen

### Synopsis

```
C      #include "cctk.h"
      #include "cctk_WarnLevel.h"

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

### Parameters

**data**           The void pointer holding extra information about the registered call back routine  
**callback**       The function pointer pointing to the call back function dealing with information messages. The definition of the function pointer is:

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

The argument list is the same as those in `CCTK_Info()` (see the discussion of `CCTK_Info()` page [A123](#)) except an extra void pointer to hold the information about the call back routine.

### Discussion

This function can be used by thorns to register their own routines to deal with information messages. The registered function pointers will be stored in a pointer chain. When `CCTK_VInfo()` is called, the registered routines will be called in the same order as they get registered in addition to dumping warning messages to `stderr`.

The function can only be called in C.

### See Also

<code>CCTK_VInfo()</code>	prints a formatted string with a variable argument list as an info message to screen
<code>CCTK_WarnCallbackRegister</code>	Register one or more routines for dealing with warning messages in addition to printing them to standard error

### Examples

```
C      /*DumpInfo will dump information messages to a file*/

      void DumpInfo(const char *thorn,
                    const char *message,
                    void *data)
      {
        DECLARE_CCTK_PARAMETERS
        FILE *fp;
```

```
char *str = (char *)malloc((strlen(thorn)
                           +strlen(message)
                           +100)*sizeof(char));

/*info_dump_file is a string set in the parameter file*/

if((fp = fopen (info_dump_file, "a"))==0)
{
    fprintf(stderr, "fatal error: can not open the file %s\n",info_dump_file);
    return;
}

sprintf(str, "\n[INFO]\nThorn->%s\nMsg->%s\n",thorn,message);

fprintf(fp, "%s", str);
free(str);
fclose(fp);
}

...

/*data = NULL; callback = DumpInfo*/

CCTK_InfoCallbackRegister(NULL,DumpInfo);
```

The computation is optimized for the case of interpolating a number of grid variables at a time; in this case all the interprocessor communication can be done together, and the same interpolation coefficients can be used for all the variables. A grid variable can be either a grid function or a grid array.

[illegible]

```

Fortran      call CCTK_InterpGridArrays(status,
.              cctkGH,
.              N_dims,
.              local_interp_handle, param_table_handle,
.              coord_system_handle,
.              N_interp_points,
.              interp_coords_type_code, interp_coords,
.              N_input_arrays, input_array_variable_indices,
.              N_output_arrays, output_array_type_codes,
.              output_arrays)
integer      status
CCTK_POINTER cctkGH
integer      local_interp_handle, param_table_handle, coord_system_handle
integer      N_dims, N_interp_points, N_input_arrays, N_output_arrays
CCTK_POINTER interp_coords(N_dims)
integer      interp_coords_type_code
CCTK_INT     input_array_variable_indices(N_input_arrays)
CCTK_INT     output_array_type_codes(N_output_arrays)
CCTK_POINTER output_arrays(N_output_arrays)

```

0	success
< 0	indicates an error condition (see <b>Errors</b> )

RCS file Revision : 4641 A126/A264

- cctkGH** ( $\neq \text{NULL}$ )  
Pointer to a valid Cactus grid hierarchy.
- N.dims** ( $\geq 1$ )  
Number of dimensions in which to interpolate. This must be  $\leq$  the dimensionality of the coordinate system defined by **coord\_system\_handle**. The default case is that it's  $=$ ; see the discussion of the **interpolation\_hyperslab\_handle** parameter-table entry for the  $<$  case.
- local\_interp\_handle** ( $\geq 0$ )  
Handle to the local interpolation operator as returned by **CCTK\_InterpHandle**.
- param\_table\_handle** ( $\geq 0$ )  
Handle to a key-value table containing zero or more additional parameters for the interpolation operation. The table is allowed to be modified by the local and/or global interpolation routine(s).
- coord\_system\_handle** ( $\geq 0$ )  
Cactus coordinate system handle defining the mapping between (usually floating-point) coordinates and integer grid subscripts, as returned by **CCTK\_CoordSystemHandle**.
- N.interp\_points** ( $\geq 0$ )  
The number of interpolation points requested by this processor.
- interp\_coords\_type\_code**  
One of the **CCTK\_VARIABLE\_\*** type codes, giving the data type of the interpolation-point coordinate arrays pointed to by **interp\_coords[]**. All interpolation-point coordinate arrays must be of the same data type. (In practice, this data type will almost always be **CCTK\_REAL** or one of the **CCTK\_REAL\*** types.)
- interp\_coords** ( $\neq \text{NULL}$ )  
(Pointer to) an array of **N.dims** pointers to 1-D arrays giving the coordinates of the interpolation points requested by this processor. These coordinates are with respect to the coordinate system defined by **coord\_system\_handle**.
- N.input\_arrays** ( $\geq 0$ )  
The number of input variables to be interpolated. If **N.input\_arrays** is zero then no interpolation is done; such a call may be useful for setup, interpolator querying, etc. Note that if the parameter table entry **operand\_indices** is used to specify a nontrivial (e.g. one-to-many) mapping of input variables to output arrays, only the unique set of input variables should be given here.
- input\_array\_variable\_indices** ( $\neq \text{NULL}$ )  
(Pointer to) an array of **N.input\_arrays** CCTK grid variable indices (as returned by **CCTK\_VarIndex**) specifying the input grid variables for the interpolation. For any element with an index value of -1 in the grid variable indices array, that interpolation is skipped. This may be useful if the main purpose of the call is e.g. to do some query or setup computation.
- N.output\_arrays** ( $\geq 0$ )  
The number of output arrays to be returned from the interpolation. If **N.output\_arrays** is zero then no interpolation is done; such a call may be useful for setup, interpolator querying, etc. Note that **N.output\_arrays** may differ from **N.input\_arrays**, e.g. if the **operand\_indices** parameter-table entry is used to specify a nontrivial (e.g. many-to-one) mapping of input variables to output arrays. If such a mapping is specified, only the unique set of output arrays should be given in the **output\_arrays** argument.
- output\_array\_type\_codes** ( $\neq \text{NULL}$ )  
(Pointer to) an array of **N.output\_arrays** **CCTK\_VARIABLE\_\*** type codes giving the data types of the 1-D output arrays pointed to by **output\_arrays[]**.
- output\_arrays** ( $\neq \text{NULL}$ )  
(Pointer to) an array of **N.output\_arrays** pointers to the (user-supplied) 1-D output arrays for the interpolation. If any of the pointers in the **output\_arrays** array is

NULL, then that interpolation is skipped. This may be useful if the main purpose of the call is e.g. to do some query or setup computation.

## Discussion

This function interpolates a list of CCTK grid variables (in a multiprocessor run these are generally distributed over processors) on a list of interpolation points. The grid topology and coordinates are implicitly specified via a Cactus coordinate system. The interpolation points may be anywhere in the global Cactus grid. In a multiprocessor run they may vary from processor to processor; each processor will get whatever interpolated data it asks for. The routine `CCTK_InterpGridArrays` does not do the actual interpolation itself but rather takes care of whatever interprocessor communication may be necessary, and – for each processor’s local patch of the domain-decomposed grid variables – calls `CCTK_InterpLocalUniform` to invoke an external local interpolation operator (as identified by an interpolation handle).

Additional parameters for the interpolation operation of both `CCTK_InterpGridArrays` and `CCTK_InterpLocalUniform` can be passed in via a handle to a key/value options table. All interpolation operators should check for a parameter table entry with the key `suppress_warnings` which – if present – indicates that the caller wants the interpolator to be silent in case of an error condition and only return an appropriate error code. One common parameter-table option, which a number of interpolation operators are likely to support, is `order`, a `CCTK_INT` specifying the order of the (presumably polynomial) interpolation (1=linear, 2=quadratic, 3=cubic, etc). As another example, a table might be used to specify that the local interpolator should take derivatives, by specifying

```
const CCTK_INT operand_indices[N_output_arrays];  
const CCTK_INT operation_codes[N_output_arrays];
```

Also, the global interpolator will typically need to specify some options of its own for the local interpolator.<sup>3</sup> These will overwrite any entries with the same keys in the `param_table_handle` table. Finally, the parameter table can be used to pass back arbitrary information by the local and/or global interpolation routine(s) by adding/modifying appropriate key/value pairs.

Note that `CCTK_InterpGridArrays` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing identical arguments except for the number of interpolation points, the interpolation coordinates, and the output array pointers. You may (and typically will) specify a different set of interpolation points on each processor’s call – you may even specify an empty set on some processors. The interpolation points may be “owned” by any processors (this function takes care of all interprocessor-communication issues), though it may be more efficient to have most or all of the interpolation points “owned” by the current processor.

In the multiprocessor case, the result returned by `CCTK_InterpGridArrays` is guaranteed to be the same on all processors. (All current implementations simply take the minimum of the per-processor results over all processors; this gives a result which is 0 if all processors succeeded, or which is the most negative error code encountered by any processor otherwise.)

The semantics of `CCTK_InterpGridArrays` are mostly independent of which Cactus driver is being used, but an implementation will most likely depend on, and make

---

<sup>3</sup> It is the caller’s responsibility to ensure that the specified local interpolator supports any optional parameter-table entries that `CCTK_InterpGridArrays` passes to it. Each thorn providing a `CCTK_InterpLocalUniform` interpolator should document what options it requires from the global interpolator.

use of, driver-specific internals. For that reason, `CCTK_InterpGridArrays` is made an overloadable function. The Cactus flesh will supply only a dummy routine for it which – if called – does nothing but print a warning message saying that it wasn't overloaded by another thorn, and stop the code. So one will always need to compile in and activate a driver-specific thorn which provides an interpolation routine for CCTK grid variables and properly overloads `CCTK_InterpGridArrays` with it at startup.

Details of the operation performed, and what (if any) inputs and/or outputs are specified in the parameter table, depend on which driver-specific interpolation thorn and interpolation operator (provided by a local interpolation thorn) you use. See the documentation on individual interpolator thorns (e.g. `PUGHInterp` in the `CactusPUGH` arrangement, `CarpetInterp` in the `Carpet` arrangement, `LocalInterp` in the `CactusBase` arrangement, and/or `AEILocalInterp` in the `AEIThorns` arrangement) for details.

Note that in a multiprocessor Cactus run, it's the user's responsibility to choose the interprocessor ghost-zone size (`driver::ghost_size`) large enough so that the local interpolator never has to off-center its molecules near interprocessor boundaries. (This ensures that the interpolation results are independent of the interprocessor decomposition, at least up to floating-point roundoff errors.) If the ghost-zone size is too small, the interpolator should return the `CCTK_ERROR_INTERP_GHOST_SIZE_TOO_SMALL` error code.

### See Also

<code>CCTK_InterpHandle()</code>	Get the interpolator handle for a given character-string name.
<code>CCTK_InterpLocalUniform()</code>	Interpolate a list of processor-local arrays which define a uniformly-spaced data grid

### Errors

The following list of error codes indicates specific error conditions. For the complete list of possible error return codes you should refer to the `ThornGuide`'s chapter of the corresponding interpolation thorn(s) you are using. To find the numerical values of the error codes (or more commonly, to find which error code corresponds to a given numerical value), look in the files `cctk.Interp.h`, `util.ErrorCodes.h`, and/or `util.Table.h` in the `src/include/` directory in the Cactus flesh.

<code>CCTK_ERROR_INTERP_POINT_OUTSIDE</code>	one or more of the interpolation points is out of range (in this case additional information about the out-of-range point may be reported through the parameter table; see the <code>Thorn Guide</code> for whatever thorn provides the local interpolation operator for further details)
<code>CCTK_ERROR_INTERP_GRID_TOO_SMALL</code>	one or more of the dimensions of the input arrays is/are smaller than the molecule size chosen by the interpolator (based on the parameter-table options, e.g. the interpolation order)
<code>CCTK_ERROR_INTERP_GHOST_SIZE_TOO_SMALL</code>	for a multi-processor run, the size of the interprocessor boundaries (the <i>ghostzone</i> size) is smaller than the molecule size chosen by the interpolator (based on the parameter-table options, e.g. the interpolation order). This error code is also returned if a processor's chunk of the global

	grid is smaller than the actual molecule size.
UTIL_ERROR_BAD_INPUT	one or more of the input arguments is invalid (e.g. NULL pointer)
UTIL_ERROR_NO_MEMORY	unable to allocate memory
UTIL_ERROR_BAD_HANDLE	parameter table handle is invalid
other error codes	this function may also return any error codes returned by the <code>Util.Table*</code> routines used to get parameters from (and/or set results in) the parameter table

## Examples

Here's a simple example to do quartic 3-D interpolation of a real and a complex grid array, at 1000 interpolation points:

C

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

#define N_DIMS          3
#define N_INTERP_POINTS 1000
#define N_INPUT_ARRAYS  2
#define N_OUTPUT_ARRAYS 2

const cGH *GH;
int operator_handle, coord_system_handle;

/* interpolation points */
CCTK_REAL interp_x[N_INTERP_POINTS],
            interp_y[N_INTERP_POINTS],
            interp_z[N_INTERP_POINTS];
const void *interp_coords[N_DIMS];

/* input and output arrays */
CCTK_INT input_array_variable_indices[N_INPUT_ARRAYS];
static const CCTK_INT output_array_type_codes[N_OUTPUT_ARRAYS]
    = { CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX };
void *output_arrays[N_OUTPUT_ARRAYS];
CCTK_REAL  output_for_real_array  [N_INTERP_POINTS];
CCTK_COMPLEX output_for_complex_array[N_INTERP_POINTS];

operator_handle = CCTK_InterpHandle("generalized polynomial interpolation");
if (operator_handle < 0)
{
    CCTK_WARN(CCTK_WARN_ABORT, "can't get operator handle!");
}

coord_system_handle = CCTK_CoordSystemHandle("cart3d");
if (coord_system_handle < 0)
{
    CCTK_WARN(CCTK_WARN_ABORT, "can't get coordinate-system handle!");
}

interp_coords[0] = (const void *) interp_x;
interp_coords[1] = (const void *) interp_y;
```

```
interp_coords[2] = (const void *) interp_z;
input_array_variable_indices[0] = CCTK_VarIndex("my_thorn::real_array");
input_array_variable_indices[1] = CCTK_VarIndex("my_thorn::complex_array");
output_arrays[0] = (void *) output_for_real_array;
output_arrays[1] = (void *) output_for_complex_array;

if (CCTK_InterpGridArrays(GH, N_DIMS,
                        operator_handle,
                        Util_TableCreateFromString("order=4"),
                        coord_system_handle,
                        N_INTERP_POINTS, CCTK_VARIABLE_REAL,
                        interp_coords,
                        N_INPUT_ARRAYS, input_array_variable_indices,
                        N_OUTPUT_ARRAYS, output_array_type_codes,
                        output_arrays) < 0)
{
    CCTK_WARN(CCTK_WARN_ABORT, "error return from interpolator!");
}
```



**CCTK\_InterpHandle**

---

Return the handle for a given interpolation operator

**Synopsis**

**C**                    `int handle = CCTK_InterpHandle( const char * operator)`

**Fortran**            `call CCTK_InterpHandle(handle , operator )`

integer handle  
character\*(\*) operator

**Parameters**

handle              Handle for the interpolation operator

operator            Name of interpolation operator

**Examples**

**C**                    `handle = CCTK_InterpHandle("my interpolation operator");`

**Fortran**            `call CCTK_InterpHandle(handle,"my interpolation operator")`

**Errors**

**negative**                      A negative value is returned for invalid/unregistered interpolation operator names.

**CCTK\_InterpLocalUniform**

---

Interpolate a list of processor-local arrays which define a uniformly-spaced data grid

The computation is optimized for the case of interpolating a number of arrays at a time; in this case the same interpolation coefficients can be used for all the arrays.

**Synopsis**

```
C      #include "util_ErrorCodes.h"
      #include "cctk.h"
      int status
          = CCTK_InterpLocalUniform(int N_dims,
                                     int operator_handle,
                                     int param_table_handle,
                                     const CCTK_REAL coord_origin[],
                                     const CCTK_REAL coord_delta[],
                                     int N_interp_points,
                                     int interp_coords_type_code,
                                     const void *const interp_coords[],
                                     int N_input_arrays,
                                     const CCTK_INT input_array_dims[],
                                     const CCTK_INT input_array_type_codes[],
                                     const void *const input_arrays[],
                                     int N_output_arrays,
                                     const CCTK_INT output_array_type_codes[],
                                     void *const output_arrays[]);

Fortran  call CCTK_InterpLocalUniform(status,
      .      N_dims,
      .      operator_handle,
      .      param_table_handle,
      .      coord_origin,
      .      coord_delta,
      .      N_interp_points,
      .      interp_coords_type_code,
      .      interp_coords,
      .      N_input_arrays,
      .      input_array_dims,
      .      input_array_type_codes,
      .      input_arrays,
      .      N_output_arrays,
      .      output_array_type_codes,
      .      output_arrays)
integer      status
integer      operator_handle, param_table_handle
integer      N_dims, N_interp_points, N_input_arrays, N_output_arrays
CCTK_REAL    coord_origin(N_dims), coord_delta(N_dims)
integer      interp_coords_type_code
CCTK_POINTER interp_coords(N_dims)
CCTK_INT     input_array_dims(N_dims), input_array_type_codes(N_input_arrays)
CCTK_POINTER input_arrays(N_input_arrays)
CCTK_INT     output_array_type_codes(N_output_arrays)
```

CCTK\_POINTER output\_arrays(N\_output\_arrays)

## Result

0                    success

## Parameters

**N.dims** ( $\geq 1$ )        Number of dimensions in which to interpolate. Note that this may be less than the number of dimensions of the input arrays if the storage is set up appropriately. For example, we might want to interpolate along 1-D lines or in 2-D planes of a 3-D input array; here **N.dims** would be 1 or 2 respectively. For details, see the section on “Non-Contiguous Input Arrays” in the Thorn Guide for thorn AEILocalInterp.

**operator\_handle** ( $\geq 0$ )  
Handle to the interpolation operator as returned by **CCTK\_InterpHandle**.

**param\_table\_handle** ( $\geq 0$ )  
Handle to a key-value table containing additional parameters for the interpolator.  
One common parameter-table option, which a number of interpolation operators are likely to support, is **order**, a **CCTK\_INT** specifying the order of the (presumably polynomial) interpolation (1=linear, 2=quadratic, 3=cubic, etc).  
See the Thorn Guide for the AEILocalInterp thorn for other parameters.

**coord\_origin** ( $\neq \text{NULL}$ )  
(Pointer to) an array giving the coordinates of the data point with integer array subscripts 0, 0, ..., 0, or more generally (if the actual array bounds don't include the all-zeros-subscript point) the coordinates which this data point would have if it existed. See the “Discussion” section below for more on how **coord\_origin[]** is actually used.

**coord\_delta** ( $\neq \text{NULL}$ )  
(Pointer to) an array giving the coordinate spacing of the data arrays. See the “Discussion” section below for more on how **coord\_delta[]** is actually used.

**N\_interp\_points** ( $\geq 0$ )  
The number of points at which interpolation is to be done.

**interp\_coords\_type\_code**  
One of the **CCTK\_VARIABLE\_\*** type codes, giving the data type of the 1-D interpolation-point-coordinate arrays pointed to by **interp\_coords[]**. (In practice, this data type will almost always be **CCTK\_REAL** or one of the **CCTK\_REAL\*** types.)

**interp\_coords** ( $\neq \text{NULL}$ )  
(Pointer to) an array of **N.dims** pointers to 1-D arrays giving the coordinates of the interpolation points. These coordinates are with respect to the coordinate system defined by **coord\_origin[]** and **coord\_delta[]**.

**N\_input\_arrays** ( $\geq 0$ )  
The number of input arrays to be interpolated. Note that if the parameter table entry **operand\_indices** is used to specify a 1-to-many mapping of input arrays to output arrays, only the unique set of input arrays should be given here.

- input\_array\_dims** ( $\neq$  NULL)  
(Pointer to) an array of `N_dims` integers giving the dimensions of the `N_dims`-D input arrays. By default all the input arrays are taken to have these dimensions, with `[0]` the most contiguous axis and `[N_dims-1]` the least contiguous axis, and array subscripts in the range  $0 \leq \text{subscript} < \text{input\_array\_dims}[\text{axis}]$ . See the discussion of the `input_array_strides` optional parameter (passed in the parameter table) for details of how this can be overridden.
- input\_array\_type\_codes** ( $\neq$  NULL)  
(Pointer to) an array of `N_input_arrays` `CCTK_VARIABLE_*` type codes giving the data types of the `N_dims`-D input arrays pointed to by `input_arrays[]`.
- input\_arrays** ( $\neq$  NULL)  
(Pointer to) an array of `N_input_arrays` pointers to the `N_dims`-D input arrays for the interpolation. If any `input_arrays[in]` pointer is NULL, that interpolation is skipped.
- N\_output\_arrays** ( $\geq 0$ )  
The number of output arrays to be returned from the interpolation.
- output\_array\_type\_codes** ( $\neq$  NULL)  
(Pointer to) an array of `N_output_arrays` `CCTK_VARIABLE_*` type codes giving the data types of the 1-D output arrays pointed to by `output_arrays[]`.
- output\_arrays** ( $\neq$  NULL)  
(Pointer to) an array of `N_output_arrays` pointers to the (user-supplied) 1-D output arrays for the interpolation. If any `output_arrays[out]` pointer is NULL, that interpolation is skipped.

## Discussion

`CCTK_InterpLocalUniform` is a generic API for interpolating processor-local arrays when the data points' *xyz* coordinates are *linear* functions of the integer array subscripts *ijk* (we're describing this for 3-D, but the generalization to other numbers of dimensions should be obvious). The `coord_origin[]` and `coord.delta[]` arguments specify these linear functions:

```
x = coord_origin[0] + i*coord.delta[0]
y = coord_origin[1] + j*coord.delta[1]
z = coord_origin[2] + k*coord.delta[2]
```

The  $(x, y, z)$  coordinates are used for the interpolation (i.e. the interpolator may internally use polynomials in these coordinates); `interp_coords[]` specifies coordinates in this same coordinate system.

Details of the operation performed, and what (if any) inputs and/or outputs are specified in the parameter table, depend on which interpolation operator you use. See the Thorn Guide for the `AEILocalInterp` thorn for further discussion.

## See Also

- |                                      |  |
|--------------------------------------|--|
| <code>CCTK_InterpHandle()</code>     | Get the interpolator handle for a given character-string name. |
| <code>CCTK_InterpGridArrays()</code> | Interpolate a list of Cactus grid arrays                       |

<code>CCTK_InterpRegisterOpLocalUniform()</code>	Register a <code>CCTK_InterpLocalUniform</code> interpolation operator
<code>CCTK_InterpLocalNonUniform()</code>	Interpolate a list of processor-local arrays, with non-uniformly spaced data points.

## Errors

To find the numerical values of the error codes (or more commonly, to find which error code corresponds to a given numerical value), look in the files `cctk_Interp.h`, `util_ErrorCodes.h`, and/or `util_Table.h` in the `src/include/` directory in the Cactus flesh.

<code>CCTK_ERROR_INTERP_POINT_OUTSIDE</code>	one or more of the interpolation points is out of range (in this case additional information about the out-of-range point may be reported through the parameter table; see the Thorn Guide for the <code>AEIInterp</code> thorn for further details)
<code>CCTK_ERROR_INTERP_GRID_TOO_SMALL</code>	one or more of the dimensions of the input arrays is/are smaller than the molecule size chosen by the interpolator (based on the parameter-table options, e.g. the interpolation order)
<code>UTIL_ERROR_BAD_INPUT</code>	one or more of the inputs is invalid (e.g. <code>NULL</code> pointer)
<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory
<code>UTIL_ERROR_BAD_HANDLE</code>	parameter table handle is invalid
other error codes	this function may also return any error codes returned by the <code>Util_Table*</code> routines used to get parameters from (and/or set results in) the parameter table

## Examples

Here's a simple example of interpolating a `CCTK_REAL` and a `CCTK_COMPLEX`  $10 \times 20$  2-D array, at 5 interpolation points, using cubic interpolation.

Note that since C allows arrays to be initialized only if the initializer values are compile-time constants, we have to declare the `interp_coords[]`, `input_arrays[]`, and `output_arrays[]` arrays as non-`const`, and set their values with ordinary (runtime) assignment statements. In C++, there's no restriction on initializer values, so we could declare the arrays `const` and initialize them as part of their declarations.

```
C
#define N_DIMS    2
#define N_INTERP_POINTS  5
#define N_INPUT_ARRAYS   2
#define N_OUTPUT_ARRAYS  2

/* (x,y) coordinates of data grid points */
#define X_ORIGIN    ...
#define X_DELTA     ...
#define Y_ORIGIN    ...
#define Y_DELTA     ...
const CCTK_REAL origin[N_DIMS] = { X_ORIGIN, Y_ORIGIN };
const CCTK_REAL delta [N_DIMS] = { X_DELTA,  Y_DELTA  };
```

---

```

/* (x,y) coordinates of interpolation points */
const CCTK_REAL interp_x[N_INTERP_POINTS];
const CCTK_REAL interp_y[N_INTERP_POINTS];
const void *interp_coords[N_DIMS];          /* see note above */

/* input arrays */
/* ... note Cactus uses Fortran storage ordering, i.e.\ X is contiguous */
#define NX    10
#define NY    20
const CCTK_REAL    input_real    [NY][NX];
const CCTK_COMPLEX input_complex[NY][NX];
const CCTK_INT input_array_dims[N_DIMS] = { NX, NY };
const CCTK_INT input_array_type_codes[N_INPUT_ARRAYS]
    = { CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX };
const void *input_arrays[N_INPUT_ARRAYS];    /* see note above */

/* output arrays */
CCTK_REAL    output_real    [N_INTERP_POINTS];
CCTK_COMPLEX output_complex[N_INTERP_POINTS];
const CCTK_INT output_array_type_codes[N_OUTPUT_ARRAYS]
    = { CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX };
void *const output_arrays[N_OUTPUT_ARRAYS];  /* see note above */

int operator_handle, param_table_handle;
operator_handle = CCTK_InterpHandle("my interpolation operator");
if (operator_handle < 0)
    CCTK_WARN(CCTK_WARN_ABORT, "can't get interpolation handle!");
param_table_handle = Util_TableCreateFromString("order=3");
if (param_table_handle < 0)
    CCTK_WARN(CCTK_WARN_ABORT, "can't create parameter table!");

/* initialize the rest of the parameter arrays */
interp_coords[0] = (const void *) interp_x;
interp_coords[1] = (const void *) interp_y;
input_arrays[0] = (const void *) input_real;
input_arrays[1] = (const void *) input_complex;
output_arrays[0] = (void *) output_real;
output_arrays[1] = (void *) output_complex;

/* do the actual interpolation, and check for error returns */
if (CCTK_InterpLocalUniform(N_DIMS,
                            operator_handle, param_table_handle,
                            origin, delta,
                            N_INTERP_POINTS,
                            CCTK_VARIABLE_REAL,
                            interp_coords,
                            N_INPUT_ARRAYS,
                            input_array_dims,
                            input_array_type_codes,
                            input_arrays,
                            N_OUTPUT_ARRAYS,
                            output_array_type_codes,

```

```
        output_arrays) < 0)  
CCTK_WARN(CCTK_WARN_ABORT, "error return from interpolator!");
```

**CCTK\_InterpRegisterOpLocalUniform**

---

Register a CCTK\_InterpLocalUniform interpolation operator.

**Synopsis**

```
C          #include "cctk.h"
          int CCTK_InterpRegisterOpLocalUniform(cInterpOpLocalUniform operator_ptr,
                                                  const char *operator_name,
                                                  const char *thorn_name);
```

**Result**

**handle** ( $\geq 0$ )     A cactus handle to refer to all interpolation operators registered under this operator name.

**Parameters**

**operator\_ptr** ( $\neq$  NULL)  
Pointer to the CCTK\_InterpLocalUniform interpolation operator. This argument must be a C function pointer of the appropriate type; the typedef can be found in `src/include/cctk_Interp.h` in the Cactus source code.

**operator\_name** ( $\neq$  NULL)  
(Pointer to) a (C-style null-terminated) character string giving the name under which to register the operator.

**thorn\_name** ( $\neq$  NULL)  
(Pointer to) a (C-style null-terminated) character string giving the name of the thorn which provides the interpolation operator.

**Discussion**

Only C functions (or other routines with C-compatible calling sequences) can be registered as interpolation operators.

**See Also**

<b>CCTK_InterpHandle()</b>	Get the interpolator handle for a given character-string name.
<b>CCTK_InterpLocalUniform()</b>	Interpolate a list of processor-local arrays, with uniformly spaced data points.

**Errors**

-1	NULL pointer was passed as interpolation operator routine
-2	interpolation handle could not be allocated
-3	Interpolation operator with this name already exists

**Examples**

```
C          /* prototype for function we want to register */
          int AEILocalInterp_InterpLocalUniform(int N_dims,
```



```
        int param_table_handle,
        /**** coordinate system ****/
        const CCTK_REAL coord_origin[],
        const CCTK_REAL coord_delta[],
        /**** interpolation points ****/
        int N_interp_points,
        int interp_coords_type_code,
        const void *const interp_coords[],
        /**** input arrays ****/
        int N_input_arrays,
        const CCTK_INT input_array_dims[],
        const CCTK_INT input_array_type_codes[],
        const void *const input_arrays[],
        /**** output arrays ****/
        int N_output_arrays,
        const CCTK_INT output_array_type_codes[],
        void *const output_arrays[]);

/* register it! */
CCTK_InterpRegisterOpLocalUniform(AEILocalInterp_InterpLocalUniform,
                                   "generalized polynomial interpolation",
                                   CCTK_THORNSTRING);
```

**CCTK\_IsFunctionAliased**

---

Reports whether an aliased function has been provided

**Synopsis**

<b>C</b>	<code>int istat = CCTK_IsFunctionAliased( const char * functionname)</code>
<b>Fortran</b>	<code>call CCTK_IsFunctionAliased(istat , functionname )</code>
	<code>integer istat</code>
	<code>character*(*) functionname</code>

**Parameters**

<code>istat</code>	the return status
<code>functionname</code>	the name of the function to check

**Discussion**

This function returns a non-zero value if the function given by `functionname` is provided by any active thorn, and zero otherwise.

**CCTK\_IsImplementationActive**

---

Reports whether an implementation was activated in a parameter file

**Synopsis**

**C**                    `int istat = CCTK_IsImplementationActive( const char * implementationname)`

**Fortran**           `CCTK_IsImplementationActive( istat, implementationname )`

integer istat  
character\*(\*) implementationname

**Parameters**

istat                the return status  
implementationname   the name of the implementation to check

**Discussion**

This function returns a non-zero value if the implementation given by `implementationname` was activated in a parameter file, and zero otherwise. See also `CCTK_ActivatingThorn` [\[A16\]](#), `CCTK_CompiledImplementation` [\[A40\]](#), `CCTK_CompiledThorn` [\[A41\]](#), `CCTK_ImplementationRequire` [\[A120\]](#), `CCTK_ImplementationThorn` [\[A121\]](#), `CCTK_ImpThornList` [\[A122\]](#), `CCTK_IsImplementationCompil` [\[A143\]](#), `CCTK_IsThornActive` [\[A144\]](#), `CCTK_NumCompiledImplementations` [\[A159\]](#), `CCTK_NumCompiledTho` [\[A160\]](#), `CCTK_ThornImplementation` [\[A236\]](#).

**CCTK\_IsImplementationCompiled**

---

Reports whether an implementation was compiled into the configuration

**Synopsis**

**C**                    `int istat = CCTK_IsImplementationCompiled( const char * implementationname)`

**Fortran**            `istat = CCTK_IsImplementationCompiled( implementationname )`

integer istat  
character\*(\*) implementationname

**Parameters**

`istat`                    the return status  
`implementationname`    the name of the implementation to check

**Discussion**

This function returns a non-zero value if the implementation given by `implementationname` was compiled into the configuration, and zero otherwise. See also `CCTK_ActivatingThorn` [\[A16\]](#), `CCTK_CompiledImplementation` [\[A40\]](#), `CCTK_CompiledThorn` [\[A41\]](#), `CCTK_ImplementationRequire` [\[A120\]](#), `CCTK_ImplementationThorn` [\[A121\]](#), `CCTK_ImpThornList` [\[A122\]](#), `CCTK_IsImplementationActive` [\[A142\]](#), `CCTK_IsThornActive` [\[A144\]](#), `CCTK_IsThornCompiled` [\[A145\]](#), `CCTK_NumCompiledImplementation` [\[A159\]](#), `CCTK_NumCompiledThorns` [\[A160\]](#), `CCTK_ThornImplementation` [\[A236\]](#).

**CCTK\_IsThornActive**

---

Reports whether a thorn was activated in a parameter file

**Synopsis**

<b>C</b>	<pre>#include "cctk.h"  int status = CCTK_IsThornActive(const char* thorn_name);</pre>
<b>Fortran</b>	<pre>#include "cctk.h"  integer status character *(*) thorn_name  status = CCTK_IsThornActive(thorn_name)</pre>

**Result**

<b>status</b>	This function returns a non-zero value if thorn <b>thorn_name</b> was activated in a parameter file, and zero otherwise.
---------------	--

**Parameters**

<b>thorn_name</b>	The character-string name of the thorn, for example "SymBase".
-------------------	--

**Discussion**

This function lets you find out at run-time whether or not a given thorn is active in the current Cactus run.

**CCTK\_IsThornCompiled**

---

Reports whether a thorn was activated in a parameter file

**Synopsis**

**C**                    `int istat = CCTK_IsThornCompiled( const char * thornname)`

**Fortran**            `istat = CCTK_IsThornCompiled( thornname )`

`integer istat`  
                  `character*(*) thornname`

**Parameters**

`istat`                the return status

`thornname`           the name of the thorn to check

**Discussion**

This function returns a non-zero value if the implementation given by `thornname` was compiled into the configuration, and zero otherwise.

**CCTK\_LocalArrayReduceOperator**

---

Returns the name of a registered reduction operator

**Synopsis**

```
C          #include "cctk.h"

          const char *name = CCTK_LocalArrayReduceOperator(int handle);
```

**Result**

**name** Returns the name of a registered local reduction operator of **handle** or NULL if the handle is invalid

**Parameters**

**handle** The handle of a registered local reduction operator

**Discussion**

This function returns the name of a registered reduction operator given its handle. NULL is returned if the handle is invalid

**See Also**

<b>CCTK_ReduceLocalArrays()</b>	Reduces a list of local arrays (new local array reduction API)
<b>CCTK_LocalArrayReductionHandle()</b>	Returns the handle of a given local array reduction operator
<b>CCTK_RegisterLocalArrayReductionOperator()</b>	Registers a function as a reduction operator of a certain name
<b>CCTK_LocalArrayReduceOperatorImplementation()</b>	Provide the implementation which provides an local array reduction operator
<b>CCTK_NumLocalArrayReduceOperators()</b>	The number of local reduction operators registered

**CCTK\_LocalArrayReduceOperatorImplementation**

---

Provide the implementation which provides an local array reduction operator

**Synopsis**

```
C          #include "cctk.h"

          const char *implementation = CCTK_LocalArrayReduceOperatorImplementation(
                                int handle);
```

**Result**

**implementation**    The name of the implementation implementing the local reduction operator of handle  
                      **handle**

**Parameters**

**handle**             The handle of a registered local reduction operator

**Discussion**

                      This function returns the implementation name of a registered reduction operator  
                      given its handle or NULL if the handle is invalid

**See Also**

<b>CCTK_ReduceLocalArrays()</b>	Reduces a list of local arrays (new local array reduction API)
<b>CCTK_LocalArrayReductionHandle()</b>	Returns the handle of a given local array reduction operator
<b>CCTK_RegisterLocalArrayReductionOperator()</b>	Registers a function as a reduction operator of a certain name
<b>CCTK_LocalArrayReduceOperator()</b>	Returns the name of a registered reduction operator
<b>CCTK_NumLocalArrayReduceOperators()</b>	The number of local reduction operators registered



**CCTK\_LocalArrayReductionHandle**

---

Returns the handle of a given local array reduction operator

**Synopsis**

```
C          #include "cctk.h"

          int handle = CCTK_ReduceLocalArrays(const char *operator);
```

**Result**

**handle**            The handle corresponding to the local reduction operator

**Parameters**

**operator**        The reduction operation to be performed. If no matching registered operator is found, a warning is issued and an error returned.

**Discussion**

This function returns the handle of the local array reduction operator. The local reduction handle is also used in the grid array reduction.

**See Also**

<code>CCTK_ReduceLocalArrays()</code>	Reduces a list of local arrays (new local array reduction API)
<code>CCTK_RegisterLocalArrayReductionOperator()</code>	Registers a function as a reduction operator of a certain name
<code>CCTK_LocalArrayReduceOperatorImplementation()</code>	Provide the implementation which provides an local array reduction operator
<code>CCTK_LocalArrayReduceOperator()</code>	Returns the name of a registered reduction operator
<code>CCTK_NumLocalArrayReduceOperators()</code>	The number of local reduction operators registered

**CCTK\_MaxDim**

---

Get the maximum dimension of any grid variable

**Synopsis**

**C**                    `int dim = CCTK_MaxDim()`

**Fortran**            `call CCTK_MaxDim(dim )`

`integer dim`

**Parameters**

`dim`                The maximum dimension

**Discussion**

Note that the maximum dimension will depend only on the active thorn list, and not the compiled thorn list.

**Examples**

**C**                    `dim = CCTK_MaxDim()`

**Fortran**            `call CCTK_MaxDim(dim)`

**CCTK\_MaxGFDim**

---

Get the maximum dimension of all grid functions

**Synopsis**

**C**                    `int dim = CCTK_MaxGFDim()`  
**Fortran**           `call CCTK_MaxGFDim(dim )`  
  
                     `integer dim`

**Parameters**

`dim`                    The maximum dimension of all grid functions

**Discussion**

Note that the maximum dimension will depend only on the active thorn list, and not the compiled thorn list.

**Examples**

**C**                    `dim = CCTK_MaxGFDim();`  
**Fortran**           `call CCTK_MaxGFDim(dim)}`

**CCTK\_MaxTimeLevels**

---

Gives the number of timelevels for a group

**Synopsis**

<b>C</b>	<code>int numlevels = CCTK_MaxTimeLevels( const char * name)</code>
<b>Fortran</b>	<code>call CCTK_MaxTimeLevels(numlevels , name )</code>
	<code>integer numlevels</code>
	<code>character*(*) name</code>

**Parameters**

<b>name</b>	The full group name
<b>numlevels</b>	The number of timelevels

**Discussion**

The group name should be in the form <implementation>::**group**>

**Examples**

<b>C</b>	<code>numlevels = CCTK_MaxTimeLevels("evolve::phivars");</code>
<b>Fortran</b>	<code>call CCTK_MAXTIMELEVELS(numlevels,"evolve::phivars")</code>

**CCTK\_MaxTimeLevelsGI**

---

Gives the number of timelevels for a group

**Synopsis**

**C**                    `int numlevels = CCTK_MaxTimeLevelsGI( int index)`

**Fortran**            `call CCTK_MaxTimeLevelsGI(numlevels , index )`

`integer numlevels`  
                  `integer index`

**Parameters**

`numlevels`            The number of timelevels

`index`                The group index

**Examples**

**C**                    `index = CCTK_GroupIndex("evolve::phivars")`  
                      `numlevels = CCTK_MaxTimeLevelsGI(index);`

**Fortran**            `call CCTK_MAXTIMELEVELSGI(numlevels,3)}`

**CCTK\_MaxTimeLevelsGN**

---

Gives the number of timelevels for a group

**Synopsis**

**C**                    `int retval = CCTK_MaxTimeLevelsGN(const char *group);`

**Result**

The maximum number of timelevels this group has, or -1 if the group name is incorrect.

**Parameters**

**group**              The variable group's name

**Discussion**

This function and its relatives return the maximum number of timelevels that the given variable group can have active. This function does not tell you anything about how many time levels are active at the time.

**CCTK\_MaxTimeLevelsVI**

---

Gives the number of timelevels for a variable

**Synopsis**

**C**                    `int numlevels = CCTK_MaxTimeLevelsVI( int index)`

**Fortran**            `call CCTK_MaxTimeLevelsVI(numlevels , index )`

`integer numlevels`  
                  `integer index`

**Parameters**

`numlevels`            The number of timelevels

`index`                The variable index

**Examples**

**C**                    `index = CCTK_VarIndex("evolve::phi")`  
                      `numlevels = CCTK_MaxTimeLevelsVI(index);`

**Fortran**            `call CCTK_MAXTIMELEVELSVI(numlevels,3)`

**CCTK\_MaxTimeLevelsVN**

---

Gives the number of timelevels for a variable

**Synopsis**

<b>C</b>	<code>int numlevels = CCTK_MaxTimeLevelsVN( const char * name)</code>
<b>Fortran</b>	<code>call CCTK_MaxTimeLevelsVN(numlevels , name )</code>
	<code>integer numlevels</code>
	<code>character*(*) name</code>

**Parameters**

<code>name</code>	The full variable name
<code>numlevels</code>	The number of timelevels

**Discussion**

The variable name should be in the form `<implementation>::<variable>`

**Examples**

<b>C</b>	<code>numlevels = CCTK_MaxTimeLevelsVN("evolve::phi")</code>
<b>Fortran</b>	<code>call CCTK_MAXTIMELEVELSVN(numlevels,"evolve::phi")</code>



**CCTK\_MyProc**

---

Returns the number of the local processor for a parallel run

**Synopsis**

**C**                    `int myproc = CCTK_MyProc( const cGH * cctkGH)`

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy

**Discussion**

For a single processor run this call will return zero. For multiprocessor runs, this call will return  $0 \leq \text{myproc} < \text{CCTK\_nProcs}(\text{cctkGH})$ .

Calling `CCTK_MyProc(NULL)` is safe (it will not crash). Current drivers (PUGH, Carpet) handle this case correctly (i.e. `CCTK_MyProc(NULL)` returns a correct result), but only a “best effort” is guaranteed for future drivers (or future revisions of current drivers).

**CCTK\_nProcs**

---

Returns the number of processors being used for a parallel run

**Synopsis**

<b>C</b>	<code>int nprocs = CCTK_nProcs( const cGH * cctkGH )</code>
<b>Fortran</b>	<code>nprocs = CCTK_nProcs( cctkGH )</code>
	<code>integer nprocs</code>
	<code>CCTK_POINTER cctkGH</code>

**Parameters**

<code>cctkGH</code>	pointer to CCTK grid hierarchy
---------------------	--------------------------------

**Discussion**

For a single processor run this call will return one.

Calling `CCTK_nProcs(NULL)` is safe (it will not crash). Current drivers (PUGH, Carpet) handle this case correctly (i.e. `CCTK_nProcs(NULL)` returns a correct result), but only a “best effort” is guaranteed for future drivers (or future revisions of current drivers).

**CCTK\_NullPointer**

---

Returns a C-style NULL pointer value.

**Synopsis**

```
Fortran      #include "cctk.h"

               CCTK_POINTER pointer_var

               pointer_var = CCTK_NullPointer()
```

**Result**

**pointer\_var**      a CCTK\_POINTER type variable which is initialized with a C-style NULL pointer

**Discussion**

Fortran doesn't know the concept of pointers so problems arise when a C function is to be called which expects a pointer as one (or more) of its argument(s).

In order to pass a NULL pointer from Fortran to C, a local CCTK\_POINTER variable should be used which has been initialized before with `CCTK_NullPointer`.

Note that there is only a Fortran wrapper available for `CCTK_NullPointer`.

**See Also**

`CCTK_PointerTo()`      Returns the address of a variable passed in by reference from a Fortran routine.

**Examples**

```
Fortran      #include "cctk.h"

               integer      ierror, table_handle
               CCTK_POINTER pointer_var

               pointer_var = CCTK_NullPointer()

               call Util_TableCreate(table_handle, 0)
               call Util_TableSetPointer(ierror, table_handle, pointer_var, "NULL pointer")
```

**CCTK\_NumCompiledImplementations**

---

Return the number of implementations compiled in.

**Synopsis**

```
C          #include "cctk.h"

          int numimpls = CCTK_NumCompiledImplementations();
```

**Result**

**numimpls**            Number of implementations compiled in.

**See Also**

CCTK_ActivatingThorn [A16]	Finds the thorn which activated a particular implementation
CCTK_CompiledImplementation [A40]	Return the name of the compiled implementation with given index
CCTK_CompiledThorn [A41]	Return the name of the compiled thorn with given index
CCTK_ImplementationRequires [A120]	Return the ancestors for an implementation
CCTK_ImplementationThorn [A121]	Returns the name of one thorn providing an implementation.
CCTK_ImpThornList [A122]	Return the thorns for an implementation
CCTK_IsImplementationActive [A142]	Reports whether an implementation was activated in a parameter file
CCTK_IsImplementationCompiled [A143]	Reports whether an implementation was compiled into a configuration
CCTK_IsThornActive [A144]	Reports whether a thorn was activated in a parameter file
CCTK_IsThornCompiled [A145]	Reports whether a thorn was compiled into a configuration
CCTK_NumCompiledThorns [A160]	Return the number of thorns compiled in
CCTK_ThornImplementation [A236]	Returns the implementation provided by the thorn

**CCTK\_NumCompiledThorns**

---

Return the number of thorns compiled in.

**Synopsis**

```
C          #include "cctk.h"

          int numthorns = CCTK_NumCompiledThorns();
```

**Result**

**numthorns**            Number of thorns compiled in.

**See Also**

CCTK_ActivatingThorn [A16]	Finds the thorn which activated a particular implementation
CCTK_CompiledImplementation [A40]	Return the name of the compiled implementation with given index
CCTK_CompiledThorn [A41]	Return the name of the compiled thorn with given index
CCTK_ImplementationRequires [A120]	Return the ancestors for an implementation
CCTK_ImplementationThorn [A121]	Returns the name of one thorn providing an implementation.
CCTK_ImpThornList [A122]	Return the thorns for an implementation
CCTK_IsImplementationActive [A142]	Reports whether an implementation was activated in a parameter file
CCTK_IsImplementationCompiled [A143]	Reports whether an implementation was compiled into a configuration
CCTK_IsThornActive [A144]	Reports whether a thorn was activated in a parameter file
CCTK_IsThornCompiled [A145]	Reports whether a thorn was compiled into a configuration
CCTK_NumCompiledImplementations [A159]	Return the number of implementations compiled in
CCTK_ThornImplementation [A236]	Returns the implementation provided by the thorn

**CCTK\_NumGridArrayReductionOperators**

---

The number of grid array reduction operators registered

**Synopsis**

```
C          #include "cctk.h"

          int num_ga_reduc = CCTK_NumGridArrayReductionOperators();
```

**Result**

`num_ga_reduc`      The number of registered grid array reduction operators (currently either 1 or 0)

**Discussion**

This function returns the number of grid array reduction operators. Since we only allow one grid array reduction operator currently, this function can be used to check if a grid array reduction operator has been registered or not.

**See Also**

<code>CCTK_ReduceGridArrays()</code>	Performs reduction on a list of distributed grid arrays
<code>CCTK_RegisterGridArrayReductionOperator()</code>	Registers a function as a grid array reduction operator of a certain name
<code>CCTK_GridArrayReductionOperator()</code>	The name of the grid reduction operator, or NULL if none is registered

**CCTK\_NumGroups**

---

Get the number of groups of variables compiled in the code

**Synopsis**

**C**                    `int number = CCTK_NumGroups()`

**Fortran**            `call CCTK_NumGroups(number )`

`integer number`

**Parameters**

**number**            The number of groups compiled from the thorns `interface.ccl` files

**Examples**

**C**                    `number = CCTK_NumGroups();`

**Fortran**            `call CCTK_NumGroups(number);`

**CCTK\_NumIOMethods**

---

Find the total number of I/O methods registered with the flesh

**Synopsis**

<b>C</b>	<code>int num_methods = CCTK_NumIOMethods (void);</code>
<b>Fortran</b>	<code>call CCTK_NumIOMethods (num_methods)</code> <code>integer num_methods</code>

**Parameters**

<code>num_methods</code>	number of registered IO methods
--------------------------	---------------------------------

**Discussion**

Returns the total number of IO methods registered with the flesh.



**CCTK\_NumLocalArrayReduceOperators**

---

The number of local reduction operators registered

**Synopsis**

```
C          #include "cctk.h"

          int num_ga_reduc = CCTK_NumLocalArrayReduceOperators();
```

**Result**

`num_ga_reduc`      The number of registered local array operators

**Discussion**

This function returns the total number of registered local array reduction operators

**See Also**

<code>CCTK_ReduceLocalArrays()</code>	Reduces a list of local arrays (new local array reduction API)
<code>CCTK_LocalArrayReductionHandle()</code>	Returns the handle of a given local array reduction operator
<code>CCTK_RegisterLocalArrayReductionOperator()</code>	Registers a function as a reduction operator of a certain name
<code>CCTK_LocalArrayReduceOperatorImplementation()</code>	Provide the implementation which provides an local array reduction operator
<code>CCTK_LocalArrayReduceOperator()</code>	Returns the name of a registered reduction operator

**CCTK\_NumReductionArraysGloballyOperators**

---

The number of global array reduction operators registered, either 1 or 0.

**Synopsis**

```
C          #include "cctk.h"

          int num_reduc = CCTK_NumReductionArraysGloballyOperators();
```

**Result**

`num_reduc`      The number of registered global array operators

**Discussion**

This function returns the total number of registered global array reduction operators, it is either 1 or 0 as we do not allow multiple array reductions.

**See Also**

<code>CCTK_ReduceArraysGlobally()</code>	Reduces a list of arrays globally
<code>CCTK_LocalArrayReductionHandle()</code>	Returns the handle of a given local array reduction operator
<code>CCTK_RegisterReduceArraysGloballyOperator()</code>	Registers a function as a reduction operator of a certain name

**CCTK\_NumTimeLevels**

---

Returns the number of active time levels for a group (deprecated).

**Synopsis**

```
C          #include "cctk.h"

            int timelevels = CCTK_NumTimeLevels(const cGH *cctkGH,
                                                const char *groupname);

            int timelevels = CCTK_NumTimeLevelsGI(const cGH *cctkGH,
                                                  int groupindex);

            int timelevels = CCTK_NumTimeLevelsGN(const cGH *cctkGH,
                                                  const char *groupname);

            int timelevels = CCTK_NumTimeLevelsVI(const cGH *cctkGH,
                                                  int varindex);

            int timelevels = CCTK_NumTimeLevelsVN(const cGH *cctkGH,
                                                  const char *varname);

Fortran   #include "cctk.h"

            subroutine CCTK_NumTimeLevels(timelevels, cctkGH, groupname)
               integer      timelevels
               CCTK_POINTER cctkGH
               character*(*) groupname
            end subroutine CCTK_NumTimeLevels

            subroutine CCTK_NumTimeLevelsGI(timelevels, cctkGH, groupindex)
               integer      timelevels
               CCTK_POINTER cctkGH
               integer      groupindex
            end subroutine CCTK_NumTimeLevelsGI

            subroutine CCTK_NumTimeLevelsGN(timelevels, cctkGH, groupname)
               integer      timelevels
               CCTK_POINTER cctkGH
               character*(*) groupname
            end subroutine CCTK_NumTimeLevelsGN

            subroutine CCTK_NumTimeLevelsVI(timelevels, cctkGH, varindex)
               integer      timelevels
               CCTK_POINTER cctkGH
               integer      varindex
            end subroutine CCTK_NumTimeLevelsVI

            subroutine CCTK_NumTimeLevelsVN(timelevels, cctkGH, varname)
               integer      timelevels
               CCTK_POINTER cctkGH
```

```
character*(*) varname  
end subroutine CCTK_NumTimeLevelsVN
```

**Result**

`timelevels`      The currently active number of timelevels for the group.

**Parameters**

`GH` ( $\neq$  NULL)      Pointer to a valid Cactus grid hierarchy.  
`groupname`      Name of the group.  
`groupindex`      Index of the group.  
`varname`      Name of a variable in the group.  
`varindex`      Index of a variable in the group.

**Discussion**

This function returns the number of timelevels for which storage has been activated, which is always equal to or less than the maximum number of timelevels which may have storage provided by `CCTK_MaxTimeLevels`.

This function has been superceded by `CCTK_ActiveTimeLevels` and should not be used any more.

**See Also**

`CCTK_ActiveTimeLevels` [\[A17\]](#)      Returns the number of active time levels for a group.  
`CCTK_MaxTimeLevels` [\[A151\]](#)      Return the maximum number of active timelevels.  
`CCTK_GroupStorageDecrease` [\[A109\]](#)      Base function, overloaded by the driver, which decreases the number of active timelevels, and also returns the number of active timelevels.  
`CCTK_GroupStorageIncrease` [\[A110\]](#)      Base function, overloaded by the driver, which increases the number of active timelevels, and also returns the number of active timelevels.

**Errors**

`timelevels < 0`      Illegal arguments given.

### CCTK\_NumTimerClocks

---

Given a `cTimerData` structure, returns its number of clocks.

#### Synopsis

```
C          int err = CCTK_NumTimerClocks(info)
```

#### Parameters

```
const cTimerData * info
```

The timer information structure whose clocks are to be counted.

**CCTK\_NumVars**

---

Get the number of grid variables compiled in the code

**Synopsis**

**C**                    `int number = CCTK_NumVars()`  
**Fortran**            `call CCTK_NumVars(number )`  
  
                     `integer number`

**Parameters**

**number**            The number of grid variables compiled from the thorn's `interface.ccl` files

**Examples**

**C**                    `number = CCTK_NumVars();`  
**Fortran**            `call CCTK_NumVars(number)`

**CCTK\_NumVarsInGroup**

---

Provides the number of variables in a group from the group name

**Synopsis**

**C**                    `int num = CCTK_NumVarsInGroup( const char * name)`

**Fortran**            `call CCTK_NumVarsInGroup(num , name )`

integer num  
character\*(\*) name

**Parameters**

num                    The number of variables in the group

group                  The full group name

**Discussion**

The group name should be given in the form <implementation>::<group>

**Examples**

**C**                    `numvars = CCTK_NumVarsInGroup("evolve::scalars")`

**Fortran**            `call CCTK_NUMVARSINGROUP(numvars,"evolve::scalars")`

**CCTK\_NumVarsInGroupI**

---

Provides the number of variables in a group from the group index

**Synopsis**

**C**                    `int num = CCTK_NumVarsInGroupI( int index)`

**Fortran**            `call CCTK_NumVarsInGroupI(num , index )`

integer num  
integer index

**Parameters**

num                    The number of variables in the group

group                  The group index

**Discussion****Examples**

**C**                    `index = CCTK_GroupIndex("evolve::scalars")}`

`firstvar = CCTK_NumVarsInGroupI(index)`

**Fortran**            `call CCTK_NUMVARSINGROUPI(firstvar,3)`



**CCTK\_OutputGH**

---

Output all variables living on the GH looping over all registered IO methods.

**Synopsis**

**C**                    `int istat = CCTK_OutputGH (const cGH *cctkGH);`

**Fortran**            `call CCTK_OutputGH (istat, cctkGH)`  
                     `integer istat`  
                     `CCTK_POINTER cctkGH`

**Parameters**

`istat`                total number of variables for which output was done by all IO methods  
`cctkGH`              pointer to CCTK grid hierarchy

**Discussion**

The IO methods decide themselves whether it is time to do output now or not.

**Errors**

0                     it wasn't time to output anything yet by any IO method  
-1                    if no IO methods were registered

## Synopsis

```

Fortran      call CCTK_OutputVar (istat, cctkGH, variable)
               integer istat
               CCTK_POINTER cctkGH
               character(*) variable

```

<b>istat</b>	return status
<b>cctkGH</b>	pointer to CCTK grid hierarchy
<b>variable</b>	full name of variable to output, with an optional options string in curly braces

The output should take place if at all possible. If the appropriate file exists the data is appended, otherwise a new file is created.

0	for success
negative	for some error condition (e.g. IO method is not registered)

## Synopsis

<b>C</b>	<pre> int istat = CCTK_OutputVarAs (const cGH *cctkGH,                                const char *variable,                                const char *alias); </pre>
<b>Fortran</b>	<pre> call CCTK_OutputVarAsByMethod (istat, cctkGH, variable, alias) integer istat CCTK_POINTER cctkGH character(*) variable character(*) alias </pre>

<b>istat</b>	return status
<b>cctkGH</b>	pointer to CCTK grid hierarchy
<b>variable</b>	full name of variable to output, with an optional options string in curly braces
<b>alias</b>	alias name to base the output filename on

The output should take place if at all possible. If 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.

positive	the number of IO methods which did output of variable
0	for success
negative	if no IO methods were registered





**CCTK\_ParallelInit**

---

Initialize the parallel subsystem

**Synopsis**

```
C          int istat = CCTK_ParallelInit( cGH * cctkGH)
```

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy

**Discussion**

Initializes the parallel subsystem.

**CCTK\_ParameterData**

---

Get parameter properties for given parameter/thorn pair.

**Synopsis**

```
C          #include "cctk.h"

          const cParamData *paramdata = CCTK_ParameterData (const char *name,
                                                             const char *thorn);
```

**Result**

**paramdata**      Pointer to parameter data structure

**Parameters**

**name**            Parameter name

**thorn**           Thorn name (for private parameters) or implementation name (for restricted parameters)

**Discussion**

The thorn or implementation name must be the name of the place where the parameter is originally defined. It is not possible to pass the thorn or implementation name of a thorn that merely declares the parameter as used.

**See Also**

**CCTK\_ParameterGet** [[A179](#)]      Get the data pointer to and type of a parameter's value

**CCTK\_ParameterLevel** [[A180](#)]      Return the parameter checking level

**CCTK\_ParameterQueryTimesSet** [[A181](#)]      Return number of times a parameter has been set

**CCTK\_ParameterSet** [[A182](#)]      Sets the value of a parameter

**CCTK\_ParameterValString** [[A187](#)]      Get the string representation of a parameter's value

**CCTK\_ParameterWalk** [[A189](#)]      Walk through list of parameters

**Errors**

**NULL**            No parameter with that name was found.

**CCTK\_ParameterGet**

---

Get the data pointer to and type of a parameter's value.

**Synopsis**

```
C          #include "cctk.h"

          const void *paramval = CCTK_ParameterGet (const char *name,
                                                    const char *thorn,
                                                    int *type);
```

**Result**

**paramval**            Pointer to the parameter value

**Parameters**

**name**                Parameter name

**thorn**              Thorn name (for private parameters) or implementation name (for restricted parameters)

**type**                If not NULL, a pointer to an integer which will hold the type of the parameter

**Discussion**

The thorn or implementation name must be the name of the place where the parameter is originally defined. It is not possible to pass the thorn or implementation name of a thorn that merely declares the parameter as used.

**See Also**

**CCTK\_ParameterData** [[A178](#)]            Get parameter properties for given parameter/thorn pair

**CCTK\_ParameterLevel** [[A180](#)]           Return the parameter checking level

**CCTK\_ParameterQueryTimesSet** [[A181](#)]        Return number of times a parameter has been set

**CCTK\_ParameterSet** [[A182](#)]               Sets the value of a parameter

**CCTK\_ParameterValString** [[A187](#)]        Get the string representation of a parameter's value

**CCTK\_ParameterWalk** [[A189](#)]            Walk through list of parameters

**Errors**

**NULL**                No parameter with that name was found.



## CCTK\_ParameterLevel

---

Return the parameter checking level.

### Synopsis

```
C          #include "cctk.h"

          int level = CCTK_ParameterLevel (void);
```

### Result

level            Parameter checking level now being used.

### See Also

CCTK_ParameterData	<a href="#">[A178]</a>	Get parameter properties for given parameter/thorn pair
CCTK_ParameterGet	<a href="#">[A179]</a>	Get the data pointer to and type of a parameter's value
CCTK_ParameterQueryTimesSet	<a href="#">[A181]</a>	Return number of times a parameter has been set
CCTK_ParameterSet	<a href="#">[A182]</a>	Sets the value of a parameter
CCTK_ParameterValString	<a href="#">[A187]</a>	Get the string representation of a parameter's value
CCTK_ParameterWalk	<a href="#">[A189]</a>	Walk through list of parameters

**CCTK\_ParameterQueryTimesSet**

---

Return number of times a parameter has been set.

**Synopsis**

```
C          #include "cctk.h"

          int nset = CCTK_ParameterQueryTimesSet (const char *name,
                                                  const char *thorn);
```

**Result**

**nset**                Number of times the parameter has been set.

**Parameters**

**name**                Parameter name

**thorn**              Thorn name (for private parameters) or implementation name (for restricted parameters)

**Discussion**

The number of times that a parameter has been set is 0 if the parameter was not set in a parameter file. The number increases when `CCTK_ParameterSet` is called.

The thorn or implementation name must be the name of the place where the parameter is originally defined. It is not possible to pass the thorn or implementation name of a thorn that merely declares the parameter as used.

**See Also**

<code>CCTK_ParameterData</code> [ <a href="#">A178</a> ]	Get parameter properties for given parameter/thorn pair
<code>CCTK_ParameterGet</code> [ <a href="#">A179</a> ]	Get the data pointer to and type of a parameter's value
<code>CCTK_ParameterLevel</code> [ <a href="#">A180</a> ]	Return the parameter checking level
<code>CCTK_ParameterSet</code> [ <a href="#">A182</a> ]	Sets the value of a parameter
<code>CCTK_ParameterValString</code> [ <a href="#">A187</a> ]	Get the string representation of a parameter's value
<code>CCTK_ParameterWalk</code> [ <a href="#">A189</a> ]	Walk through list of parameters

**Errors**

-1                    No parameter with that name exists.

**CCTK\_ParameterSet**

---

Sets the value of a parameter.

**Synopsis**

```
C          #include "cctk.h"

          int ierr = CCTK_ParameterSet (const char *name,
                                       const char *thorn,
                                       const char *value);

Fortran    call CCTK_ParameterSet (ierr, name, thorn, value)
          CCTK_INT ierr
          character*(*) name
          character*(*) thorn
          character*(*) value
```

**Result**

ierr            Error code

**Parameters**

name            Parameter name

thorn           Thorn name (for private parameters) or implementation name (for restricted parameters)

value           The new (stringified) value for the parameter parameter

**Discussion**

The thorn or implementation name must be the name of the place where the parameter is originally defined. It is not possible to pass the thorn or implementation name of a thorn that merely declares the parameter as used.

While setting a new parameter value is immediately reflected in Cactus' database, the value of the parameter is not changed immediately in the routine that sets the new value: It is updated only the next time a routine is entered (or rather, when the `DECLARE_CCTK_PARAMETERS` is encountered the next time). It is therefore advisable to set the new parameter value in a routine scheduled at a time earlier to when the new value is required.

**See Also**

CCTK\_ParameterData [\[A178\]](#)            Get parameter properties for given parameter/thorn pair

CCTK\_ParameterLevel [\[A180\]](#)           Return the parameter checking level

CCTK\_ParameterQueryTimesSet [\[A181\]](#)           Return number of times a parameter has been set

CCTK\_ParameterSetNotifyRegister [\[A184\]](#)           Registers a parameter set operation notify callback

CCTK\_ParameterSetNotifyUnregister [\[A186\]](#)           Unregisters a parameter set operation notify callback

CCTK\_ParameterValString [\[A187\]](#)           Get the string representation of a parameter's value

CCTK\_ParameterWalk [\[A189\]](#)

Walk through list of parameters

## Errors

ierr

0 success

−1 parameter is out of range

−2 parameter was not found

−3 trying to steer a non-steerable parameter

−6 not a valid integer or float

−7 tried to set an accumulator parameter directly

−8 tried to set an accumulator parameter directly

−9 final value of accumulator out of range

**CCTK\_ParameterSetNotifyRegister**

---

Registers a parameter set operation notify callback

**Synopsis**

```
C          #include "cctk.h"

          int handle =
            CCTK_ParameterSetNotifyRegister (cParameterSetNotifyCallbackFn callback,
                                              void *data,
                                              const char *name,
                                              const char *thorn_regex,
                                              const char *param_regex)

Fortran    call CCTK_ParameterSetNotifyRegister (handle, callback, data,
          .
          integer      handle
          external     callback
          integer      callback
          CCTK_POINTER data
          character(*) name
          character(*) thorn_regex
          character(*) param_regex
```

**Result**

0	success
-1	another callback has already been registered under the given name
-2	memory allocation error
-3	invalid regular expression given for thorn_regex / param_regex

**Parameters**

<b>callback</b>	Function pointer of the notify callback to be registered
<b>data</b>	optional user-defined data pointer to associate with the notify callback
<b>name</b>	Unique name under which the notify callback is to be registered
<b>thorn_regex</b>	Optional regular expression string to match a thorn name in a full parameter name
<b>param_regex</b>	Optional regular expression string to match a parameter name in a full parameter name

**Discussion**

Declaring a parameter steerable at runtime in its `param.ccl` definition requires a thorn writer to add extra logic to the code which checks if a parameter value has changed, either periodically in a scheduled function, or by direct notification from the flesh's parameter set routine `CCTK_ParameterSet()`.

With `CCTK_ParameterSetNotifyRegister()` thorns can register a callback function which in turn is automatically invoked by `CCTK_ParameterSet()` whenever a parameter is being steered. Each callback function gets passed the triple of thorn name, parameter name, and (stringified) new parameter value (as passed to `CCTK_ParameterSet()`),



**CCTK\_ParameterSetNotifyUnregister**

---

Unregisters a parameter set operation notify callback

**Synopsis**

```
C          #include "cctk.h"

          int ierr = CCTK_ParameterSetNotifyUnregister (const char *name);
Fortran    call CCTK_ParameterSetNotifyUnregister (ierr, name)
          integer      ierr
          character*(*) name
```

**Result**

0	success
-1	no callback was registered under the given name

**Parameters**

name	Unique name under which the notify callback was registered
------	--

**Discussion**

Notify callbacks should be unregistered when not needed anymore.

**See Also**

CCTK_ParameterSet	<a href="#">[A182]</a>	Sets the value of a parameter
CCTK_ParameterSetNotifyRegister	<a href="#">[A184]</a>	Registers a parameter set operation notify callback

**Examples**

```
Fortran    #include "cctk.h"

          call CCTK_ParameterSetNotifyUnregister (CCTK_THORNSTRING)
```

**CCTK\_ParameterValString**

---

Get the string representation of a parameter's value.

**Synopsis**

```
C          #include "cctk.h"

          char *valstring = CCTK_ParameterValString (const char *name,
                                                    const char *thorn);

Fortran    subroutine CCTK_ParameterValString (nchars, name, thorn, value)
           integer      nchars
           character*(*) name
           character*(*) thorn
           character*(*) value
           end subroutine
```

**Result**

**valstring**      Pointer to parameter value as string. *The memory for this string must be released with a call to `free()` after it has been used.*

**Parameters**

**name**            Parameter name

**thorn**           Thorn name (for private parameters) or implementation name (for restricted parameters)

**nchars**          On exit, the number of characters in the stringified parameter value, or `-1` if the parameter doesn't exist

**value**           On exit, contains as many characters of the stringified parameter value as fit into the Fortran string provided. You should check for truncation by comparing **nchars** against the length of your Fortran string.

**Discussion**

In C, the string **valstring** must be freed afterwards.

The thorn or implementation name must be the name of the place where the parameter is originally defined. It is not possible to pass the thorn or implementation name of a thorn that merely declares the parameter as used.

Real variables are formatted according to the C `"%.20g"` format.

**See Also**

<code>CCTK_ParameterData</code> <a href="#">[A178]</a>	Get parameter properties for given parameter/thorn pair
<code>CCTK_ParameterGet</code> <a href="#">[A179]</a>	Get the data pointer to and type of a parameter's value
<code>CCTK_ParameterLevel</code> <a href="#">[A180]</a>	Return the parameter checking level
<code>CCTK_ParameterQueryTimesSet</code> <a href="#">[A181]</a>	Return number of times a parameter has been set
<code>CCTK_ParameterSet</code> <a href="#">[A182]</a>	Sets the value of a parameter
<code>CCTK_ParameterWalk</code> <a href="#">[A189]</a>	Walk through list of parameters



## Errors

NULL                      No parameter with that name was found.

## CCTK\_ParameterWalk

---

Walk through the list of parameters.

### Synopsis

```
C      #include "cctk.h"
      %
      int istat = CCTK_ParameterWalk (int first,
                                     const char *origin,
                                     char **fullname,
                                     const cParamData **paramdata);
```

### Result

**istat**                Zero for success, positive if parameter was not found, negative if initial startpoint was not set.

### Parameters

**origin**             Thorn name, or NULL for all thorns.  
**fullname**          Address of a pointer that will point to the full parameter name. This name must be freed after use.  
**paramdata**        Address of a pointer that will point to the parameter data structure.

### Discussion

Gets parameters in order, restricted to ones from **origin**, or all if **origin** is NULL. Starts with the first parameter if **first** is true, otherwise gets the next one. Can be used for generating full help file, or for walking the list and checkpointing.

### See Also

CCTK_ParameterData <a href="#">[A178]</a>	Get parameter properties for given parameter/thorn pair
CCTK_ParameterGet <a href="#">[A179]</a>	Get the data pointer to and type of a parameter's value
CCTK_ParameterLevel <a href="#">[A180]</a>	Return the parameter checking level
CCTK_ParameterQueryTimesSet <a href="#">[A181]</a>	Return number of times a parameter has been set
CCTK_ParameterSet <a href="#">[A182]</a>	Sets the value of a parameter
CCTK_ParameterValString <a href="#">[A187]</a>	Get the string representation of a parameter's value

### Errors

**negative**            The initial startpoint was not set.

**CCTK\_PARAMWARN**

---

Prints a warning from parameter checking, and possibly stops the code

**Synopsis**

**C**                   = CCTK\_PARAMWARN( const char \* message)

**Fortran**           call CCTK\_PARAMWARN( , message )

                    character\*(\*) message

**Parameters**

message           The warning message

**Discussion**

The call should be used in routines registered at the schedule point CCTK\_PARAMCHECK to indicate that there is parameter error or conflict and the code should terminate. The code will terminate only after all the parameters have been checked.

**Examples**

**C**                   CCTK\_PARAMWARN("Mass cannot be negative");

**Fortran**           call CCTK\_PARAMWARN("Inside interpolator")

## CCTK\_PointerTo

---

Returns a pointer to a Fortran variable.

### Synopsis

```
Fortran      #include "cctk.h"

              CCTK_POINTER addr, var

              addr = CCTK_PointerTo(var)
```

### Result

addr            the address of variable *var*

### Parameters

var            variable in the Fortran context from which to take the address

### Discussion

Fortran doesn't know the concept of pointers so problems arise when a C function is to be called which expects a pointer as one (or more) of its argument(s).

To obtain the pointer to a variable in Fortran, one can use `CCTK_PointerTo()` which takes the variable itself as a single argument and returns the pointer to it.

Note that there is only a Fortran wrapper available for `CCTK_PointerTo`.

### See Also

`CCTK_NullPointer()`            Returns a C-style NULL pointer value.

### Examples

```
Fortran      #include "cctk.h"

              integer      ierror, table_handle
              CCTK_POINTER addr, var

              addr = CCTK_PointerTo(var)

              call Util_TableCreate(table_handle, 0)
              call Util_TableSetPointer(ierror, table_handle, addr, "variable")
```

**CCTK\_PrintGroup**

---

Prints a group name from its index

**Synopsis**

**C**                    = CCTK\_PrintGroup( int index)  
**Fortran**            call CCTK\_PrintGroup( , index )  
  
                      integer index

**Parameters**

index                The group index

**Discussion**

This routine is for debugging purposes for Fortran programmers.

**Examples**

**C**                    CCTK\_PrintGroup(1)  
**Fortran**            call CCTK\_PRINTGROUP(1)

**CCTK\_PrintString**

---

Prints a Cactus string

**Synopsis**

**C**                   = CCTK\_PrintString( char \* string)  
**Fortran**           call CCTK\_PrintString( , string )  
  
                    CCTK\_STRING string

**Parameters**

**string**           The string to print

**Discussion**

This routine can be used to print Cactus string variables and parameters from Fortran.

**Examples**

**C**                   CCTK\_PrintString(string\_param)  
**Fortran**           call CCTK\_PRINTSTRING(string\_param)

**CCTK\_PrintVar**

---

Prints a variable name from its index

**Synopsis**

**C**                    = CCTK\_PrintVar( int index)  
**Fortran**            call CCTK\_PrintVar( , index )  
  
                      integer index

**Parameters**

index                The variable index

**Discussion**

This routine is for debugging purposes for Fortran programmers.

**Examples**

**C**                    CCTK\_PrintVar(1)  
**Fortran**            call CCTK\_PRINTVAR(1)

**CCTK\_QueryGroupStorage**

---

Query storage for a group given by its group name

**Synopsis**

**C**                    `int istat = CCTK_QueryGroupStorage( const cGH * cctkGH, const char * groupname)`  
**Fortran**           `call CCTK_QueryGroupStorage(istat , cctkGH, groupname )`

`integer istat`  
                  `CCTK_POINTER cctkGH`  
                  `character*(*) groupname`

**Parameters**

`cctkGH`             pointer to CCTK grid hierarchy  
`groupname`        the group to query, given by its full name  
`istat`             the return code

**Discussion**

This routine queries whether the variables in a group have storage assigned. If so it returns true (a positive value), otherwise false (zero).

**Errors**

**negative**                                    A negative error code is returned for an invalid group name.



**CCTK\_QueryGroupStorageB**

---

**Synopsis**

**C**                    `int storage = CCTK_QueryGroupStorageB( const cGH * cctkGH, int groupindex, const char *`

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy  
`groupindex`       the group to query, given by its index  
`groupname`       the group to query, given by its full name  
`istat`            the return code

**Discussion**

This routine queries whether the variables in a group have storage assigned. If so it returns true (a positive value), otherwise false (zero).

The group can be specified either through the group index `groupindex`, or through the group name `groupname`. The groupname takes precedence; only if it is passed as NULL, the group index is used.

**Errors**

`negative`                    A negative error code is returned for an invalid group name.

**CCTK\_QueryGroupStorageI**

---

Query storage for a group given by its group index

**Synopsis**

**C**                    `int istat = CCTK_QueryGroupStorageI( const cGH * cctkGH, int groupindex)`  
**Fortran**            `call CCTK_QueryGroupStorageI(istat , cctkGH, groupindex )`

`integer istat`  
                  `cctkGH`  
                  `integer groupindex`

**Parameters**

`cctkGH`            pointer to CCTK grid hierarchy  
`groupindex`       the group to query, given by its index  
`istat`            the return code

**Discussion**

This routine queries whether the variables in a group have storage assigned. If so it returns true (a positive value), otherwise false (zero).

**Errors**

**negative**                                A negative error code is returned for an invalid group name.

**CCTK\_ReduceArraysGlobally**

---

Performs global reduction on a list of arrays

The computation is optimized for the case of reducing a number of grid arrays at a time; in this case all the interprocessor communication can be done together.

**Synopsis**

```
C          #include "cctk.h"

            int CCTK_ReduceArraysGlobally(const cGH *GH,
                                           int dest_proc,
                                           int local_reduce_handle,
                                           int param_table_handle,
                                           int N_input_arrays,
                                           const void * const input_arrays[],
                                           int input_dims,
                                           const CCTK_INT input_array_dims[],
                                           const CCTK_INT input_array_type_codes[],
                                           int M_output_values,
                                           const CCTK_INT output_value_type_codes[],
                                           void* const output_values[]);

Fortran    call CCTK_ReduceGridArrays(status,
            .      GH,
            .      dest_processor,
            .      local_reduce_handle,
            .      param_table_handle,
            .      N_input_arrays,
            .      input_arrays,
            .      input_dims,
            .      input_array_dims,
            .      input_array_type_codes,
            .      M_output_values,
            .      output_value_type_codes,
            .      output_values)
            integer      status
            CCTK_POINTER_TO_CONST GH
            integer      dest_processor
            integer      local_reduce_handle
            integer      param_table_handle
            integer      N_input_arrays
            CCTK_INT     input_arrays(N_input_arrays)
            integer      input_dims
            CCTK_INT     input_array_dims(input_dims)
            CCTK_INT     input_array_type_codes(N_input_arrays)
            integer      M_output_values
            CCTK_INT     output_value_type_codes(M_output_values)
            CCTK_POINTER output_values(M_output_values)
```

**Result**

0                    success  
< 0                indicates an error condition

### Parameters

**cctkGH** ( $\neq$  NULL)  
Pointer to a valid Cactus grid hierarchy.

**dest\_processor**    The destination processor. -1 will distribute the result to all processors.

**local\_reduce\_handle** ( $\geq 0$ )  
Handle to the local reduction operator as returned by `CCTK_LocalArrayReductionHandle()`. It is the caller's responsibility to ensure that the specified reducer supports any optional parameter-table entries that `CCTK_ReduceGridArrays()` passes to it. Each thorn providing a `CCTK_ReduceGridArrays()` reducer should document what options it requires from the local reducer.

**param\_table\_handle** ( $\geq 0$ )  
Handle to a key-value table containing zero or more additional parameters for the reduction operation. The table can be modified by the local and/or global reduction routine(s).  
  
Also, the global reducer will typically need to specify some options of its own for the local reducer. These will override any entries with the same keys in the `param_table_handle` table. The discussion of individual table entries below says if these are modified in this manner.  
  
Finally, the `param_table_handle` table can be used to pass back arbitrary information by the local and/or global reduction routine(s) by adding/modifying appropriate key/value pairs.

**N\_input\_arrays** ( $\geq 0$ )  
The number of input arrays to be reduced. If `N_input_arrays` is zero, then no reduction is done; such a call may be useful for setup, reducer querying, etc. If the `operand_indices` parameter table entry is used to specify a nontrivial (eg 1-to-many) mapping of input arrays to output values, only the unique set of input arrays should be given here.

**input\_arrays**        (Pointer to) an array of `N_input_arrays` local arrays specifying the input arrays for the reduction.

**input\_dims** ( $\geq 0$ )  
The number of dimensions of the input arrays

**input\_array\_dims** ( $\geq 0$ )  
(Pointer to) an array of size `input_dims` containing the dimensions of the arrays to be reduced.

**input\_array\_type\_codes** ( $\geq 0$ )  
(Pointer to) an array of `input_dims` `CCTK_VARIABLE_*` type codes giving the data types of the arrays to be reduced.

**M\_output\_values** ( $\geq 0$ )  
The number of output values to be returned from the reduction. If `N_input_arrays`  $\neq 0$  then no reduction is done; such a call may be useful for setup, reducer querying, etc. Note that `M_output_values` may differ from `N_input_arrays`, eg if the `operand_indices` parameter table entry is used to specify a nontrivial (eg many-to-1) mapping of input arrays to output values, If such a mapping is specified, only the unique set of output values should be given here.

`output_value_type_codes`

(Pointer to) an array of `M.output_values` `CCTK_VARIABLE_*` type codes giving the data types of the output values pointed to by `output_values[]`.

`output_values`

(Pointer to) an array of `M.output_values` pointers to the (caller-supplied) output values for the reduction. If `output_values[out]` is NULL for some index or indices `out`, then that reduction is skipped. (This may be useful if the main purpose of the call is (eg) to do some query or setup computation.) These pointers may (and typically will) vary from processor to processor in a multiprocessor Cactus run. However, any given pointer must be either NULL on all processors, or non-NULL on all processors.

4

## Discussion

This function reduces a list of CCTK local arrays globally. This function does not perform the actual reduction, it only handles interprocessor communication. The actual reduction is performed by the local reduction implementation, that is passed arguments and parameters from the grid array reduction implementation.

Note that `CCTK_ReduceArraysGlobally` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing identical arguments.

## See Also

`CCTK_LocalArrayReductionHandle()`

Returns the handle of a given local array reduction operator

`CCTK_RegisterGridArrayReductionOperator()`

Registers a function as a grid array reduction operator of a certain name

`CCTK_GridArrayReductionOperator()`

The name of the grid reduction operator, or NULL if the handle is invalid

`CCTK_GridArrayReductionOperator()`

The number of grid array reduction operators registered

## Examples

Here's a simple example to perform grid array reduction of two grids arrays of different types.

C

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

#define N_INPUT_ARRAYS 2
#define M_OUTPUT_VALUES 2
const cGH *GH;                                /* input */

/* create empty parameter table */
const int param_table_handle = Util_CreateTable(UTIL_TABLE_FLAGS_CASE_INSENSITIVE);
/* input arrays and output values */
const CCTK_INT input_array_variable_indices[N_INPUT_ARRAYS]
    = { CCTK_VarIndex("my_thorn::real_array"), /* no error checking */
        CCTK_VarIndex("my_thorn::complex_array") }; /* here */
```

```
const CCTK_INT output_value_type_codes[M_OUTPUT_VALUES]
    = { CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX };
void *const output_numbers[M_OUTPUT_VALUES]
    = { (void *) output_for_real_values,
        (void *) output_for_complex_values };

const int status
    = CCTK_ReduceGridArrays(GH,
                           0,
                           param_table_handle,
                           N_INPUT_ARRAYS, input_array_variable_indices,
                           M_OUTPUT_VALUES, output_value_type_codes,
                           output_values);

Util_TableDestroy(param_table_handle);
```

**CCTK\_ReduceGridArrays**

---

Performs reduction on a list of distributed grid arrays

The computation is optimized for the case of reducing a number of grid arrays at a time; in this case all the interprocessor communication can be done together.

**Synopsis**

```
C          #include "cctk.h"

          int status = CCTK_ReduceGridArrays(const cGH *GH,
                                             int dest_processor,
                                             int local_reduce_handle,
                                             int param_table_handle,
                                             int N_input_arrays,
                                             const CCTK_INT input_array_variable_indices[],
                                             int M_output_values,
                                             const CCTK_INT output_value_type_codes[],
                                             void* const output_values());

Fortran    call CCTK_ReduceGridArrays(status,
          .      GH,
          .      dest_processor,
          .      local_reduce_handle,
          .      param_table_handle,
          .      N_input_arrays,
          .      input_array_variable_indices,
          .      M_output_values,
          .      output_value_type_codes,
          .      output_values)
          integer      status
          CCTK_POINTER_TO_CONST GH
          integer      dest_processor
          integer      local_reduce_handle
          integer      param_table_handle
          integer      N_input_arrays
          CCTK_INT      input_array_variable_indices(N_input_arrays)
          integer      M_output_values
          CCTK_INT      output_value_type_codes(M_output_values)
          CCTK_POINTER  output_values(M_output_values)
```

**Result**

0                    success  
< 0                  indicates an error condition

**Parameters**

**cctkGH** ( $\neq$  NULL)      Pointer to a valid Cactus grid hierarchy.  
**dest\_processor**      The destination processor. -1 will distribute the result to all processors.

**local\_reduce\_handle** ( $\geq 0$ )

Handle to the local reduction operator as returned by `CCTK_LocalArrayReductionHandle()`. It is the caller's responsibility to ensure that the specified reducer supports any optional parameter-table entries that `CCTK_ReduceGridArrays()` passes to it. Each thorn providing a `CCTK_ReduceGridArrays()` reducer should document what options it requires from the local reducer.

**param\_table\_handle** ( $\geq 0$ )

Handle to a key-value table containing zero or more additional parameters for the reduction operation. The table can be modified by the local and/or global reduction routine(s).

Also, the global reducer will typically need to specify some options of its own for the local reducer. These will override any entries with the same keys in the `param_table_handle` table. The discussion of individual table entries below says if these are modified in this manner.

Finally, the `param_table_handle` table can be used to pass back arbitrary information by the local and/or global reduction routine(s) by adding/modifying appropriate key/value pairs.

**N\_input\_arrays** ( $\geq 0$ )

The number of input arrays to be reduced. If `N_input_arrays` is zero, then no reduction is done; such a call may be useful for setup, reducer querying, etc. If the `operand_indices` parameter table entry is used to specify a nontrivial (eg 1-to-many) mapping of input arrays to output values, only the unique set of input arrays should be given here.

**input\_array\_variable\_indices**

(Pointer to) an array of `N_input_arrays` Cactus variable indices (as returned by `CCTK_VarIndex()`) specifying the input grid arrays for the reduction. If `input_array_variable_indices[in] == -1` for some index or indices `in`, then that reduction is skipped. (This may be useful if the main purpose of the call is (eg) to do some query or setup computation.)

**M\_output\_values** ( $\geq 0$ )

The number of output values to be returned from the reduction. If `N_input_arrays == 0` then no reduction is done; such a call may be useful for setup, reducer querying, etc. Note that `M_output_values` may differ from `N_input_arrays`, eg if the `operand_indices` parameter table entry is used to specify a nontrivial (eg many-to-1) mapping of input arrays to output values. If such a mapping is specified, only the unique set of output values should be given here.

**output\_value\_type\_codes**

(Pointer to) an array of `M_output_values` `CCTK_VARIABLE_*` type codes giving the data types of the output values pointed to by `output_values[]`.

**output\_values**

(Pointer to) an array of `M_output_values` pointers to the (caller-supplied) output values for the reduction. If `output_values[out]` is `NULL` for some index or indices `out`, then that reduction is skipped. (This may be useful if the main purpose of the call is (eg) to do some query or setup computation.) These pointers may (and typically will) vary from processor to processor in a multiprocessor Cactus run. However, any given pointer must be either `NULL` on all processors, or non-`NULL` on all processors.

## Discussion

This function reduces a list of CCTK grid arrays (in a multiprocessor run these are generally distributed over processors). This function does not perform the actual



reduction, it only handles interprocessor communication. The actual reduction is performed by the local reduction implementation, that is passed arguments and parameters from the grid array reduction implementation.

Note that `CCTK_ReduceGridArrays` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing identical arguments.

### See Also

<code>CCTK_LocalArrayReductionHandle()</code>	Returns the handle of a given local array reduction operator
<code>CCTK_RegisterGridArrayReductionOperator()</code>	Registers a function as a grid array reduction operator of a certain name
<code>CCTK_GridArrayReductionOperator()</code>	The name of the grid reduction operator, or NULL if the handle is invalid
<code>CCTK_GridArrayReductionOperator()</code>	The number of grid array reduction operators registered

### Examples

Here's a simple example to perform grid array reduction of two grids arrays of different types.

C

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

#define N_INPUT_ARRAYS 2
#define M_OUTPUT_VALUES 2
const cGH *GH;                                /* input */

/* create empty parameter table */
const int param_table_handle = Util_CreateTable(UTIL_TABLE_FLAGS_CASE_INSENSITIVE);
/* input arrays and output values */
const CCTK_INT input_array_variable_indices[N_INPUT_ARRAYS]
    = { CCTK_VarIndex("my_thorn::real_array"), /* no error checking */
        CCTK_VarIndex("my_thorn::complex_array") }; /* here */
const CCTK_INT output_value_type_codes[M_OUTPUT_VALUES]
    = { CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX };
void *const output_numbers[M_OUTPUT_VALUES]
    = { (void *) output_for_real_values,
        (void *) output_for_complex_values };

const int status
    = CCTK_ReduceGridArrays(GH,
        0,
        param_table_handle,
        N_INPUT_ARRAYS, input_array_variable_indices,
        M_OUTPUT_VALUES, output_value_type_codes,
        output_values);
```

```
Util_TableDestroy(param_table_handle);
```

**CCTK\_ReduceLocalArrays**

---

Performs reduction on a list of local grid arrays

**Synopsis**

```
C          #include "cctk.h"

          int status = CCTK_ReduceLocalArrays(int N_dims, int operator_handle,
                                              int param_table_handle, int N_input_arrays,
                                              const CCTK_INT input_array_dims[],
                                              const CCTK_INT input_array_type_codes[],
                                              const void *const input_arrays[],
                                              int M_output_numbers,
                                              const CCTK_INT output_number_type_codes[],
                                              void *const output_values[]);

Fortran    call CCTK_ReduceLocalArrays(status,
.          N_dims, operator_handle,
.          param_table_handle, N_input_arrays,
.          input_array_dims,
.          input_array_type_codes,
.          input_arrays,
.          M_output_numbers,
.          output_number_type_codes,
.          output_values)
integer    status
integer    N_dims
integer    operator_handle
integer    param_table_handle
integer    N_input_arrays
CCTK_INT   input_array_dims(N_dims)
CCTK_INT   input_array_type_codes(N_input_arrays)
CCTK_POINTER input_arrays(N_input_arrays)
integer    M_output_values
CCTK_INT   output_value_type_codes(M_output_values)
CCTK_POINTER output_values(M_output_values)
```

**Result**

0                    success  
< 0                  indicates an error condition

**Parameters**

**N\_dims**            Number of dimensions of input arrays. This is required to find proper indices for arrays in memory

**operator\_handle**   Handle to the local reduction operator as returned by  
                     CCTK\_LocalArrayReductionHandle().

**param\_table\_handle**   Handle to a key-value table containing zero or more additional parameters for the  
                     reduction operation. The table can be modified by the local and/or global reduction

routine(s).

The parameter table may be used to specify non-default storage indexing for input or output arrays, and/or various options for the reduction itself. Some reducers may not implement all of these options.

**N\_input\_arrays** ( $\geq 0$ )

The number of input arrays to be reduced. If **N\_input\_arrays** is zero, then no reduction is done; such a call may be useful for setup, reducer querying, etc. If the **operand\_indices** parameter table entry is used to specify a nontrivial (eg 1-to-many) mapping of input arrays to output values, only the unique set of input arrays should be given here.

**input\_array\_dims**

array of input array dimensions (common to all input arrays) and of size **N\_dims**

**input\_array\_type\_codes**

array of input array dimensions (common to all input arrays) and of size **N\_input\_arrays**

**M\_output\_values** ( $\geq 0$ )

The number of output values to be returned from the reduction. If **N\_input\_arrays** == 0 then no reduction is done; such a call may be useful for setup, reducer querying, etc. Note that **M\_output\_values** may differ from **N\_input\_arrays**, eg if the **operand\_indices** parameter table entry is used to specify a nontrivial (eg many-to-1) mapping of input arrays to output values. If such a mapping is specified, only the unique set of output values should be given here.

**output\_value\_type\_codes**

(Pointer to) an array of **M\_output\_values** **CCTK\_VARIABLE\_\*** type codes giving the data types of the output values pointed to by **output\_values[]**.

**output\_values**

(Pointer to) an array of **M\_output\_values** pointers to the (caller-supplied) output values for the reduction. If **output\_values[out]** is NULL for some index or indices **out**, then that reduction is skipped. (This may be useful if the main purpose of the call is (eg) to do some query or setup computation.)

## Discussion

Sometimes one of the arrays used by the reduction isn't contiguous in memory. So, we use several optional table entries (these should be supported by all reduction operators):

For the input arrays, we use

```
const CCTK_INT input_array_offsets[N_input_arrays];
/* next 3 table entries are shared by all input arrays */
const CCTK_INT input_array_strides      [N_dims];
const CCTK_INT input_array_min_subscripts[N_dims];
const CCTK_INT input_array_max_subscripts[N_dims];
```

Then for input array number *a*, the generic subscripting expression for the 3-D case is

```
data_pointer[offset + i*istride + j*jstride + k*kstride]
```

where

```
data_pointer = input_arrays[a]
offset = input_array_offsets[a]
(istride,jstride,kstride) = input_array_stride[]
```

and where (*i*, *j*, *k*) run from **input\_array\_min\_subscripts[]** to **input\_array\_max\_subscripts[]** inclusive.

The defaults are `offset=0`, `stride=determined from input_array_dims[]` in the usual Fortran manner, `input_array_min_subscripts[] = 0`, `input_array_max_subscripts[] = input_array_dims[]-1`. If the stride and max subscript are both specified explicitly, then the `input_array_dims[]` function argument is ignored.

### See Also

<code>CCTK_LocalArrayReductionHandle()</code>	Returns the handle of a given local array reduction operator
<code>CCTK_RegisterLocalArrayReductionOperator()</code>	Registers a function as a reduction operator of a certain name
<code>CCTK_LocalArrayReduceOperatorImplementation()</code>	Provide the implementation which provides an local array reduction operator
<code>CCTK_LocalArrayReduceOperator()</code>	Returns the name of a registered reduction operator
<code>CCTK_NumLocalArrayReduceOperators()</code>	The number of local reduction operators registered

### Examples

Here's a simple example, written in Fortran 77, to do reduction of a real and a complex local array in 3-D:

#### Fortran 77

```
c input arrays:
    integer ni, nj, nk
    parameter (ni=..., nj=..., nk=...)
    CCTK_REAL    real_array    (ni,nj,nk)
    CCTK_COMPLEX complex_array(ni,nj,nk)

c output numbers:
    CCTK_REAL    My_real    (M_reduce)
    CCTK_COMPLEX My_complex(M_reduce)

    integer status, dummy
    CCTK_INT input_array_type_codes(2)
    data input_array_type_codes /CCTK_VARIABLE_REAL,
$                                CCTK_VARIABLE_COMPLEX/
    CCTK_INT input_array_dims(3)
    CCTK_POINTER input_arrays(2)
    CCTK_POINTER output_numbers(2)

    input_array_dims(1) = ni
    input_array_dims(2) = nj
    input_array_dims(3) = nk
    output_numbers(1) = Util_PointerTo(My_real)
    output_numbers(2) = Util_PointerTo(My_complex)

    call CCTK_ReduceLocalArrays
$      (status,                ! return code
        3,                    ! number of dimensions
        operator_handle,
```

```
      N_reduce,  
      2,                                ! number of input arrays  
      input_array_type_codes, input_array_dims, input_arrays,  
      2,                                ! number of output numbers  
      output_numbers_type_codes, output_numbers)  
  
if (status .lt. 0) then  
  call CCTK_WARN(CCTK_WARN_ABORT, "Error return from reducer!")  
end if
```

## CCTK\_ReductionHandle

---

Handle for given reduction method

### Synopsis

<b>C</b>	<code>int handle = CCTK_ReductionHandle( const char * reduction)</code>
<b>Fortran</b>	<code>handle = CCTK_ReductionHandle( reduction )</code>
	<code>integer handle</code>
	<code>character*(*) reduction</code>

### Parameters

<code>handle</code>	handle returned for this method
<code>name</code>	name of the reduction method required

### Discussion

Reduction methods should be registered at `CCTK_STARTUP`. Note that integer reduction handles are used to call `CCTK_Reduce` to avoid problems with passing Fortran strings. Note that the name of the reduction operator is case dependent.

### Examples

<b>C</b>	<code>handle = CCTK_ReductionHandle("maximum");</code>
<b>Fortran</b>	<code>call CCTK_ReductionHandle(handle,"maximum")</code>

**CCTK\_RegisterBanner**

---

Register a banner for a thorn

**Synopsis**

**C**                    `void = CCTK_RegisterBanner( const char * message)`  
**Fortran**            `call CCTK_RegisterBanner( , message )`  
  
                     `character*(*) message`

**Parameters**

**message**            String which will be displayed as a banner

**Discussion**

The banner must be registered during `CCTK_STARTUP`. The banners are displayed in the order in which they are registered.

**Examples**

**C**                    `CCTK_RegisterBanner("My Thorn: Does Something Useful");`  
**Fortran**            `call CCTK_REGISTERBANNER("*** MY THORN ***")`



## CCTK\_RegisterGHEExtension

---

Register an extension to the CactusGH

### Synopsis

```
C          int istat = CCTK_RegisterGHEExtension( const char * name)
```

## CCTK\_RegisterGHEExtensionInitGH

---

Register a function which will initialise a given extension to the Cactus GH

### Synopsis

**C**                    `int istat = CCTK_RegisterGHEExtensionInitGH( int handle, void * (*func)(cGH *))`

## CCTK\_RegisterGHExtensionScheduleTraverseGH

---

Register a GH extension schedule traversal routine

### Synopsis

**C**                    `int istat = CCTK_RegisterGHExtensionScheduleTraverseGH( int handle, int (*func)(cGH *,c`

## CCTK\_RegisterGHExtensionSetupGH

---

Register a function which will set up a given extension to the Cactus GH

### Synopsis

**C**                    `int istat = CCTK_RegisterGHExtensionSetupGH( int handle, void * (*func)(tFleshConfig *`

**CCTK\_RegisterGridArrayReductionOperator**

---

Registers a function as a grid array reduction operator of a certain name

**Synopsis**

```
C          #include "cctk.h"

          int status = CCTK_RegisterGridArrayReductionOperator(
                        cGridArrayReduceOperator operator)
```

**Result**

0                    success  
< 0                  indicates an error condition

**Parameters**

**operator**            The function to register as a global reduction function.

**Discussion**

This function simply registers a function as the grid array reduction. Currently we support a single function as a global reduction function (this can be modified to accomodate more functions if need be).

**See Also**

CCTK_ReduceGridArrays()	Performs reduction on a list of distributed grid arrays
CCTK_GridArrayReductionOperator()	The name of the grid reduction operator, or NULL if none is registered
CCTK_NumGridArrayReductionOperators()	The number of grid array reduction operators registered

**CCTK\_RegisterIOMethod**

---

Register a new I/O method

**Synopsis**

**C**                    `int handle = CCTK_RegisterIOMethod( const char * name)`

**Fortran**            `handle = CCTK_RegisterIOMethod( name )`

`integer handle`  
                  `name`

**Parameters**

`handle`            handle returned by registration

`name`             name of the I/O method

**Discussion**

IO methods should be registered at CCTK\_STARTUP.

## CCTK\_RegisterIOMethodOutputGH

---

Register a routine for an I/O method which will be called from CCTK\_OutputGH.

### Synopsis

**C**                    `int istat = CCTK_RegisterIOMethodOutputGH( int handle, int (* func)(const cGH *))`

## CCTK\_RegisterIOMethodOutputVarAs

---

Register a routine for an I/O method which will provide aliased variable output

### Synopsis

C                    `int istat = CCTK_RegisterIOMethodOutputVarAs( int handle, int (* func)(const cGH *,con`



### CCTK\_RegisterIOMethodTimeToOutput

---

Register a routine for an I/O method which will decide if it is time for the method to output.

#### Synopsis

**C**                    `int istat = CCTK_RegisterIOMethodTimeToOutput( int handle, int (* func)(const cGH *,in`

## CCTK\_RegisterIOMethodTriggerOutput

---

Register a routine for an I/O method which will handle trigger output

### Synopsis

**C**                    `int istat = CCTK_RegisterIOMethodTriggerOutput( int handle, int (* func)(const cGH *,i`

**CCTK\_RegisterLocalArrayReductionOperator**

---

Registers a function as a reduction operator of a certain name

**Synopsis**

```
C          #include "cctk.h"

          int handle = CCTK_RegisterLocalArrayReductionOperator(
                      cLocalArrayReduceOperator operator, const char *name);
```

**Result**

**handle**            The handle corresponding to the registered local reduction operator, -1 if an error occurred.

**Parameters**

**operator**        The function to be registered as a local reduction operator  
**name**            The name under which the operator is registered as a local reduction operator

**Discussion**

This function registers a local array reduction operator. It registers an **operator** under a **name** with the flesh and returns its assigned handle. If another reduction operator exists with the same **name**, an error is returned.

**See Also**

<b>CCTK_ReduceLocalArrays()</b>	Reduces a list of local arrays (new local array reduction API)
<b>CCTK_LocalArrayReductionHandle()</b>	Returns the handle of a given local array reduction operator
<b>CCTK_LocalArrayReduceOperatorImplementation()</b>	Provide the implementation which provides an local array reduction operator
<b>CCTK_LocalArrayReduceOperator()</b>	Returns the name of a registered reduction operator
<b>CCTK_NumLocalArrayReduceOperators()</b>	The number of local reduction operators registered

**CCTK\_RegisterReduceArraysGloballyOperator**

---

Registers a function as a reduction operator of a certain name

**Synopsis**

```
C          #include "cctk.h"

          int handle = CCTK_RegisterReduceArraysGloballyOperator(
                        cReduceArraysGloballyOperator operator, const char *name);
```

**Result**

**handle**            The handle corresponding to the registered global array reduction operator, -1 if an error occurred.

**Parameters**

**operator**        The function to be registered as a global array reduction operator  
**name**            The name under which the operator is registered as a global array reduction operator

**Discussion**

This function registers a global array reduction operator. It registers an **operator** under a **name** with the flesh and returns its assigned handle. If another reduction operator exists with the same **name**, an error is returned.

**See Also**

**CCTK\_ReduceArraysGlobally()**      Reduces a list of local arrays globally

CCTK\_RegisterReductionOperator

---

Synopsis

C                   CCTK\_RegisterReductionOperator()

**CCTK\_SchedulePrintTimes**

---

Output the timing results for a certain schedule item to stdout

**Synopsis**

```
C          #include "cctk.h"
          int status = CCTK_SchedulePrintTimes(const char *where)
```

**Result**

Return code of DoScheduleTraverse, or

0                Success.

**Parameters**

**where**                Name of schedule item, or NULL to print the whole schedule

**Discussion**

Output the timing results for a certain schedule item to stdout. The schedule item is traversed recursively if it is a schedule group or a schedule bin.

This routine is used to produce the timing output when the parameter `Cactus::cctk_timer_output` is set to **yes**.

**See Also**

`CCTK_SchedulePrintTimesToFile`    Output the timing results for a certain schedule item to a file

**Examples**

```
C          Output the timer results for the Analysis bin:
          #include "cctk.h"
          int status = CCTK_SchedulePrintTimes("CCTK_ANALYSIS")
```

**CCTK\_SchedulePrintTimesToFile**

---

Output the timing results for a certain schedule item to stdout

**Synopsis**

```
C          #include "cctk.h"
          int status = CCTK_SchedulePrintTimesToFile(const char *where)
```

**Result**

Return code of DoScheduleTraverse, or

0                Success.

**Parameters**

**where**                Name of schedule item, or NULL to print the whole schedule  
**file**                 File to which the results are output; the file must be open for writing

**Discussion**

Output the timing results for a certain schedule item to a file. The schedule item is traversed recursively if it is a schedule group or a schedule bin.

Note that each processor will output its results. You should either call this routine on only a single processor, or you should pass different files on different processors.

**See Also**

**CCTK\_SchedulePrintTimes**                Output the timing results for a certain schedule item to stdout

**Examples**

```
C          Output the timer results of processor 3 for the Analysis bin to a file:
          #include <stdio.h>
          #include "cctk.h"
          if (CCTK_MyProc(cctkGH)==3)
          {
              FILE *file = fopen("timing-results.txt", "a");
              int status = CCTK_SchedulePrintTimesToFile("CCTK_ANALYSIS", file)
              fclose(file);
          }
```

## CCTK\_SetupGH

---

Setup a new GH

### Synopsis

**C**                    `cGH * cctkGH = CCTK_SetupGH( tFleshConfig config, int convlevel)`



## CCTK\_SyncGroup

---

Synchronise the ghostzones for a group of grid variables (identified by the group name)

### Synopsis

```
C          #include "cctk.h"
          int status = CCTK_SyncGroup(const cGH* GH, const char* group_name)

Fortran    #include "cctk.h"
          integer status
          CCTK_POINTER GH
          character*(*) group_name
          call CCTK_SyncGroup(status, GH, group_name)
```

### Result

0                      Success.

### Parameters

GH                      A pointer to a Cactus grid hierarchy.

group\_name              The full name (Implementation::group or Thorn::group) of the group to be synchronized.

### Discussion

Only those grid variables which have communication enabled will be synchronised. This is usually equivalent to the variables which have storage assigned, unless communication has been explicitly turned off with a call to `CCTK_DisableGroupComm`.

Note that an alternative to calling `CCTK_SyncGroup` explicitly from within a thorn, is to use the `SYNC` keyword in a thorns `schedule.ccl` file to indicate which groups of variables need to be synchronised on exit from the routine. This latter method is the preferred method from synchronising variables.

Note that `CCTK_SyncGroup` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing the same `group_name` argument.

### See Also

CCTK\_SyncGroupI [\[A230\]](#)                      Synchronise the ghostzones for a group of grid variables (identified by the group index)

CCTK\_SyncGroupsI [\[A232\]](#)                      Synchronise the ghostzones for a list of groups of grid variables (identified by their group indices)

### Errors

-1                      `group_name` was invalid.

-2                      The driver returned an error on syncing the group.

### Examples

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

```
#include "cctk_Arguments.h"

/* this function synchronizes the ADM metric */
void synchronize_ADM_metric(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS      /* defines "magic variable" cctkGH */

    const int status = CCTK_SyncGroup(cctkGH, "ADMBase::metric");
    if (status < 0)
        CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric():\n"
"        failed to synchronize ADM metric!\n"
"        (CCTK_SyncGroup() returned error code %d)\n"
"        ,
        status);                      /*NOTREACHED*/
}
```

## CCTK\_SyncGroupI

---

Synchronise the ghostzones for a group of grid variables (identified by the group index)

### Synopsis

```
C          #include "cctk.h"
          int status = CCTK_SyncGroupI(const cGH* GH, int group_index)

Fortran    #include "cctk.h"
          integer status
          CCTK_POINTER GH
          integer group_index
          call CCTK_SyncGroupI(status, GH, group_index)
```

### Result

0                      Success.

### Parameters

GH                      A pointer to a Cactus grid hierarchy.  
group\_index            The group index of the group to be synchronized.

### Discussion

Only those grid variables which have communication enabled will be synchronised. This is usually equivalent to the variables which have storage assigned, unless communication has been explicitly turned off with a call to `CCTK_DisableGroupComm`.

Note that an alternative to calling `CCTK_SyncGroupI` explicitly from within a thorn, is to use the `SYNC` keyword in a thorns `schedule.ccl` file to indicate which groups of variables need to be synchronised on exit from the routine. This latter method is the preferred method from synchronising variables.

Note that `CCTK_SyncGroupI` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing the same `group_name` argument.

### See Also

CCTK_SyncGroup	[A228]	Synchronise the ghostzones for a group of grid variables (identified by the group name)
CCTK_SyncGroupsI	[A232]	Synchronise the ghostzones for a list of groups of grid variables (identified by their group indices)
CCTK_GroupIndex	[A91]	Gets the group index for a given group name.
CCTK_GroupIndexFromVar	[A92]	Gets the group index for a given variable name.

### Errors

-1	<code>group_name</code> was invalid.
-2	The driver returned an error on syncing the group.

### Examples

C

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

/* this function synchronizes the ADM metric */
void synchronize_ADM_metric(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS      /* defines "magic variable" cctkGH */

  int group_index, status;

  group_index = CCTK_GroupIndex("ADMBase::metric");
  if (group_index < 0)
    CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric():\n"
"      couldn't get group index for ADM metric!\n"
"      (CCTK_GroupIndex() returned error code %d)\n"
"      ,
              group_index);                      /*NOTREACHED*/

  status = CCTK_SyncGroupI(cctkGH, group_index);
  if (status < 0)
    CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric():\n"
"      failed to synchronize ADM metric!\n"
"      (CCTK_SyncGroupI() returned error code %d)\n"
"      ,
              status);                      /*NOTREACHED*/
}
```

## CCTK\_SyncGroupsI

---

Synchronise the ghostzones for a list of groups of grid variables (identified by their group indices)

### Synopsis

```
C          #include "cctk.h"
          int status = CCTK_SyncGroupsI(const cGH* GH, int num_groups, const int *groups)

Fortran    #include "cctk.h"
          integer status
          CCTK_POINTER GH
          integer num_groups
          integer groups(num_groups)
          call CCTK_SyncGroupsI(status, GH, num_groups, groups)
```

### Result

0 Returns the number of groups that have been synchronised.

### Parameters

GH A pointer to a Cactus grid hierarchy.  
num\_groups The number of groups to be synchronised.  
groups The group indices of the groups to be synchronized.

### Discussion

Only those grid variables which have communication enabled will be synchronised. This is usually equivalent to the variables which have storage assigned, unless communication has been explicitly turned off with a call to `CCTK_DisableGroupComm`.

Note that an alternative to calling `CCTK_SyncGroupsI` explicitly from within a thorn, is to use the `SYNC` keyword in a thorns `schedule.ccl` file to indicate which groups of variables need to be synchronised on exit from the routine. This latter method is the preferred method from synchronising variables.

Note that `CCTK_SyncGroupsI` is a collective operation, so in the multiprocessor case you *must* call this function in parallel on *each* processor, passing the same number of groups in the same order.

### See Also

<code>CCTK_SyncGroup</code> <a href="#">[A228]</a>	Synchronise the ghostzones for a single group of grid variables (identified by the group name)
<code>CCTK_SyncGroupI</code> <a href="#">[A230]</a>	Synchronise the ghostzones for a single group of grid variables (identified by the group index)
<code>CCTK_GroupIndex</code> <a href="#">[A91]</a>	Gets the group index for a given group name.
<code>CCTK_GroupIndexFromVar</code> <a href="#">[A92]</a>	Gets the group index for a given variable name.

### Examples

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

```
/* this function synchronizes the ADM metric and lapse */
void synchronize_ADM_metric_and_lapse(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS      /* defines "magic variable" cctkGH */

  int group_indices[2], status;

  group_indices[0] = CCTK_GroupIndex("ADMBase::metric");
  group_indices[1] = CCTK_GroupIndex("ADMBase::lapse");
  if (group_indices[0] < 0)
    CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric():\n"
"      couldn't get group index for ADM metric!\n"
"      (CCTK_GroupIndex() returned error code %d)\n"
"      ,
              group_indices[0]);                                /*NOTREACHED*/
  if (group_indices[1] < 0)
    CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric_and_lapse():\n"
"      couldn't get group index for ADM lapse!\n"
"      (CCTK_GroupIndex() returned error code %d)\n"
"      ,
              group_indices[1]);                                /*NOTREACHED*/

  status = CCTK_SyncGroupsI(cctkGH, 2, group_indices);
  if (status != 2)
    CCTK_VWarn(CCTK_WARN_ABORT, __LINE__, __FILE__, CCTK_THORNSTRING,
"***** synchronize_ADM_metric_and_lapse():\n"
"      failed to synchronize ADM metric and lapse!\n"
"      (CCTK_SyncGroupsI() returned error code %d)\n"
"      ,
              status);                                          /*NOTREACHED*/
}
```

**CCTK\_TerminateNext**

---

Causes a Cactus simulation to terminate after present iteration finishes

**Synopsis**

```
C          #include "cctk.h"

              void CCTK_TerminateNext (const cGH *cctkGH)

Fortran    #include "cctk.h"

              call CCTK_TerminateNext (cctkGH)
              CCTK_POINTER_TO_CONST    cctkGH
```

**Parameters**

cctkGH      Pointer to a Cactus grid hierarchy.

**Discussion**

This function triggers unconditional termination of Cactus after the present iteration. It bypasses all other termination conditions specified in the `Cactus::terminate` keyword parameter.

At this time, the `cctkGH` parameter does nothing.

**See Also**

CCTK\_TerminationReached [\[A235\]](#)    Returns true if CCTK\_TerminateNext has been called.

**CCTK\_TerminationReached**

---

Returns true if CCTK\_TerminateNext has been called.

**Synopsis**

```
C           #include "cctk.h"

              void CCTK_TerminationReached (const cGH *cctkGH)

Fortran     #include "cctk.h"

              call CCTK_TerminationReached (cctkGH)
              CCTK_POINTER_TO_CONST      cctkGH
```

**Parameters**

cctkGH            Pointer to a Cactus grid hierarchy.

**Discussion**

Returns true if Cactus has been requested to terminate after the present iteration by the CCTK\_TerminateNext function.

At this time, the cctkGH parameter does nothing.

**See Also**

CCTK\_TerminateNext [\[A234\]](#)            Causes a Cactus simulation to terminate after the present iteration.



**CCTK\_ThornImplementation**

---

Returns the implementation provided by the thorn.

**Synopsis**

```
C          #include "cctk.h"

          const char *imp = CCTK_ThornImplementationThorn(const char *name);
```

**Result**

imp                      Name of the implementation or NULL

**Parameters**

name                    Name of the thorn

**See Also**

CCTK_ActivatingThorn <a href="#">[A16]</a>	Finds the thorn which activated a particular implementation
CCTK_CompiledImplementation <a href="#">[A40]</a>	Return the name of the compiled implementation with given index
CCTK_CompiledThorn <a href="#">[A41]</a>	Return the name of the compiled thorn with given index
CCTK_ImplementationRequires <a href="#">[A120]</a>	Return the ancestors for an implementation
CCTK_ImplementationThorn <a href="#">[A121]</a>	Returns the name of one thorn providing an implementation.
CCTK_ImpThornList <a href="#">[A122]</a>	Return the thorns for an implementation
CCTK_IsImplementationActive <a href="#">[A142]</a>	Reports whether an implementation was activated in a parameter file
CCTK_IsImplementationCompiled <a href="#">[A143]</a>	Reports whether an implementation was compiled into a configuration
CCTK_IsThornActive <a href="#">[A144]</a>	Reports whether a thorn was activated in a parameter file
CCTK_IsThornCompiled <a href="#">[A145]</a>	Reports whether a thorn was compiled into a configuration
CCTK_NumCompiledImplementations <a href="#">[A159]</a>	Return the number of implementations compiled in
CCTK_NumCompiledThorns <a href="#">[A160]</a>	Return the number of thorns compiled in

**Errors**

NULL                    Error.

## CCTK\_Timer

---

Fills a `cTimerData` structure with timer clock info, for the timer specified by name.

### Synopsis

```
C                int err = CCTK_Timer(name,info)
```

### Parameters

```
const char * name      Timer name
cTimerData * info      Timer clock info pointer
```

### Errors

A negative return value indicates an error.

## CCTK\_TimerCreate

---

Creates a timer with a given name, returns an index to the timer.

### Synopsis

```
C                int index = CCTK_TimerCreate(name)
```

### Parameters

```
const char * name
           timer name
```

### Errors

```
< 0                A negative return value indicates an error.
```

### CCTK\_TimerCreateData

---

Allocates the `cTimerData` structure, which is used to store timer clock info.

#### Synopsis

```
C          cTimerData * info = CCTK_TimerCreateData()
```

#### Errors

NULL                                      A null return value indicates an error.

## CCTK\_TimerCreateI

---

Creates an unnamed timer, returns an index to the timer.

### Synopsis

```
C          int index = CCTK_TimerCreate()
```

### Errors

< 0                      A negative return value indicates an error.

## CCTK\_TimerDestroy

---

Reclaims resources used by the given timer, specified by name.

### Synopsis

```
C          int err = CCTK_TimerDestroy(name)
```

### Parameters

```
const char * name
           timer name
```

### Errors

< 0                      A negative return value indicates an error.

**CCTK\_TimerDestroyData**

---

Releases resources from the `cTimerData` structure, created by `CCTK_TimerCreateData`.

**Synopsis**

```
C          int err = CCTK_TimerDestroyData(info)
```

**Parameters**

```
cTimerData * info  
           Timer clock info pointer
```

**Errors**

```
< 0          A negative return value indicates an error.
```

**CCTK\_TimerDestroyI**

---

Reclaims resources used by the given timer, specified by index.

**Synopsis**

```
C          int err = CCTK_TimerDestroyI(index)
```

**Parameters**

int index          timer index

**Errors**

< 0                      A negative return value indicates an error.



**CCTK\_TimerI**

---

Fills a `cTimerData` structure with timer clock info, for the timer specified by index.

**Synopsis**

```
C                int err = CCTK_TimerI(index,info)
```

**Parameters**

```
int index        Timer index
cTimerData * info Timer clock info pointer
```

**Errors**

```
< 0                A negative return value indicates an error.
```

## CCTK\_TimerReset

---

Gets values from all the clocks in the given timer, specified by name.

### Synopsis

```
C                int err = CCTK_TimerReset(name)
```

### Parameters

```
const char * name
           timer name
```

### Errors

```
< 0                A negative return value indicates an error.
```

**CCTK\_TimerResetI**

---

Gets values from all the clocks in the given timer, specified by index.

**Synopsis**

```
C                int err = CCTK_TimerResetI(index)
```

**Parameters**

`int index`            timer index

**Errors**

`< 0`                            A negative return value indicates an error.

## CCTK\_TimerStart

---

Initialises all the clocks in the given timer, specified by name.

### Synopsis

```
C                int err = CCTK_TimerStart(name)
```

### Parameters

```
const char * name
           timer name
```

### Errors

< 0                      A negative return value indicates an error.

**CCTK\_TimerStartI**

---

Initialises all the clocks in the given timer, specified by index.

**Synopsis**

```
C                int err = CCTK_TimerStartI(index)
```

**Parameters**

`int index`            timer index

**Errors**

`< 0`                            A negative return value indicates an error.

**CCTK\_TimerStop**

---

Gets values from all the clocks in the given timer, specified by name.

**Synopsis**

```
C          int err = CCTK_TimerStop(name)
```

**Parameters**

`int name`            timer name

**Discussion**

Call this before getting the values from any of the timer's clocks.

**Errors**

`< 0`                            A negative return value indicates an error.

**CCTK\_TimerStopI**

---

Gets values from all the clocks in the given timer, specified by index.

**Synopsis**

```
C          int err = CCTK_TimerStopI(index)
```

**Parameters**

int index            timer index

**Discussion**

Call this before getting the values from any of the timer's clocks.

**Errors**

< 0                            A negative return value indicates an error.

**CCTK\_VarDataPtr**

---

Returns the data pointer for a grid variable

**Synopsis**

```
C          void * ptr = CCTK_VarDataPtr( const cGH * cctkGH, int timelevel, char * name)
```

**Parameters**

<code>cctkGH</code>	pointer to CCTK grid hierarchy
<code>timelevel</code>	The timelevel of the grid variable
<code>name</code>	The full name of the variable

**Discussion**

The variable name should be in the form `<implementation>::<variable>`.

**Examples**

```
C          myVar = (CCTK_REAL *) (CCTK_VarDataPtr(GH,0,"imp::realvar"))
```



**CCTK\_VarDataPtrB**

---

Returns the data pointer for a grid variable from the variable index or the variable name

**Synopsis**

```
C          void * ptr = CCTK_VarDataPtrB( const cGH * cctkGH, int timelevel, int index, char * name )
```

**Parameters**

**ptr**                    a void pointer to the grid variable data

**cctkGH**                The timelevel of the grid variable

**index**                The index of the variable

**name**                The full name of the variable

**Discussion**

If the name is `NULL` the index will be used, if the index is negative the name will be used.

**Examples**

```
C          myVar = (CCTK_REAL *) (CCTK_VarDataPtrB(GH,0,CCTK_VarIndex("imp::realvar"),NULL));
```

**CCTK\_VarDataPtrI**

---

Returns the data pointer for a grid variable from the variable index

**Synopsis**

```
C          void * ptr = CCTK_VarDataPtrI( const cGH * cctkGH, int timelevel, int index)
```

**Parameters**

<code>cctkGH</code>	pointer to CCTK grid hierarchy
<code>timelevel</code>	The timelevel of the grid variable
<code>index</code>	The index of the variable

**Examples**

```
C          myVar = (CCTK_REAL *) (CCTK_VarDataPtr(GH,0,CCTK_VarIndex("imp::realvar")));
```

## CCTK\_VarIndex

---

Get the index for a variable.

### Synopsis

<b>C</b>	<pre>#include "cctk.h" int index = CCTK_VarIndex(const char *varname);</pre>
<b>Fortran</b>	<pre>call CCTK_VarIndex(index, varname)       integer index       character*(*) varname</pre>

### Parameters

<b>varname</b>	The name of the variable.
----------------	---------------------------

### Discussion

The variable name should be the given in its fully qualified form, that is `<implementation>::<variable>` for a public or protected variable, and `<thornname>::<variable>` for a private variable.

### Errors

-1	no variable of this name exists
-2	failed to catch error code from Util_SplitString
-3	given full name is in wrong format
-4	memory allocation failed

### Examples

<b>C</b>	<pre>index = CCTK_VarIndex("evolve::phi");</pre>
<b>Fortran</b>	<pre>call CCTK_VarIndex(index,"evolve::phi")</pre>

**CCTK\_VarName**

---

Given a variable index, returns the variable name

**Synopsis**

```
C          const char * name = CCTK_VarName( int index)
```

**Parameters**

name	The variable name
index	The variable index

**Discussion**

The pointer returned is part of a structure managed by Cactus and so must *not* be freed after use.

No Fortran routine exists at the moment.

**Examples**

```
C          index = CCTK_VarIndex("evolve::phi");  
          name = CCTK_VarName(index);
```

**CCTK\_VarTypeI**

---

Provides variable type index from the variable index

**Synopsis**

**C**                    `int type = CCTK_VarTypeI( int index)`

**Fortran**            `call CCTK_VarTypeI(type , index )`

integer type  
integer index

**Parameters**

type                The variable type index

group               The variable index

**Discussion**

The variable type index indicates the type of the variable. Either character, int, complex or real. The group type can be checked with the Cactus provided macros for CCTK\_VARIABLE\_INT, CCTK\_VARIABLE\_REAL, CCTK\_VARIABLE\_COMPLEX or CCTK\_VARIABLE\_CHAR.

**Examples**

**C**                    `index = CCTK_VarIndex("evolve::phi")`  
                      `real = (CCTK_VARIABLE_REAL == CCTK_VarTypeI(index)) ;`

**Fortran**            `call CCTK_VARTYPEI(type,3)`

**CCTK\_VarTypeSize**

---

Provides variable type size in bytes from the variable type index

**Synopsis**

**C**                    `#include "cctk.h"`

`int CCTK_VarTypeSize(int vtype);`

**Fortran**            `#include "cctk.h"`

`CCTK_INT size, vtype`  
                      `call CCTK_VarTypeSize(size, vtype);`

**Parameters**

**vtype**              Variable type index.

**Discussion**

Given a CCTK\_VARIABLE\_\* type code (e.g. CCTK\_VARIABLE\_INT, CCTK\_VARIABLE\_REAL, CCTK\_VARIABLE\_COMPLEX, etc.), this function returns the size in bytes of the corresponding data type (CCTK\_INT, CCTK\_REAL, CCTK\_COMPLEX, etc.).

**Errors**

-1                    **vtype** is not one of the CCTK\_VARIABLE\_\* values.

**CCTK\_VInfo**

---

Prints a formatted string with a variable argument list as an info message to screen

**Synopsis**

```
C      #include "cctk.h"
      #include "cctk_WarnLevel.h"

      int status = CCTK_VInfo(const char *thorn,
                             const char *format,
                             ...);
```

**Result**

0                    ok

**Parameters**

<b>thorn</b>	The name of the thorn printing this info message. You can use the <code>CCTK_THORNSTRING</code> macro here (defined in <code>cctk.h</code> ).
<b>format</b>	The <code>printf</code> -like format string to use for printing the info message.
<b>...</b>	The variable argument list.

**Discussion**

This routine can be used by thorns to print a formatted string with a variable argument list as an info message to screen. The message will include the name of the originating thorn, otherwise its semantics is equivalent to `printf`.

**See Also**

`CCTK_INFO` [\[A123\]](#)                    macro to print an info message with a single string argument

**Examples**

```
C      #include "cctk.h"
      #include "cctk_WarningLevel.h"

      const char *outdir;

      CCTK_VInfo(CCTK_THORNSTRING, "Output files will go to '%s'", outdir);
```

## CCTK\_VWarn

---

Possibly prints a formatted string with a variable argument list as warning message and/or stops the code

### Synopsis

```
C      #include "cctk.h"
      #include "cctk_WarnLevel.h"

      int status = CCTK_VWarn(int level,
                             int line,
                             const char *file,
                             const char *thorn,
                             const char *format,
                             ...);
```

### Result

0                    ok<sup>4</sup>

### Parameters

level ( $\geq 0$ )	The warning level for the message to print, with level 0 being the severest level and greater levels being less severe.
line	The line number in the originating source file where the <code>CCTK_VWarn</code> call occurred. You can use the standardized <code>__LINE__</code> preprocessor macro here.
file	The file name of the originating source file where the <code>CCTK_VWarn</code> call occurred. You can use the standardized <code>__FILE__</code> preprocessor macro here.
thorn	The thorn name of the originating source file where the <code>CCTK_VWarn</code> call occurred. You can use the <code>CCTK_THORNSTRING</code> macro here (defined in <code>cctk.h</code> ).
format	The <code>printf</code> -like format string to use for printing the warning message.
...	The variable argument list.

### Discussion

This routine can be used by thorns to print a formatted string followed by a variable argument list as a warning message to `stderr`. If the message’s “warning level” is severe enough, then after printing the message Cactus aborts the run (and `CCTK_VWarn` does *not* return to the caller).

Cactus’s behavior when `CCTK_VWarn` is called depends on the `-W` and `-E` command-line options:

- Cactus prints any warning with a warning level  $\leq$  the `-W` level to standard error (any warnings with warning levels  $>$  the `-W` level are silently discarded). The default `-W` level is 1, i.e. only level 0 and level 1 warnings will be printed.
- Cactus stops (aborts) the current run for any warning with a warning level  $\leq$  the `-E` level. The default `-W` level is 0, i.e. only level 0 warnings will abort the run.

---

<sup>4</sup>When this function is called, the calling code almost always ignores the return result. However, it’s still useful for this function to be declared as returning a value, rather than having type `void`, since this allows it to be used in C conditional expressions.



Cactus guarantees that the `-W` level  $\geq$  the `-E` level  $\geq 0$ . This implies that a message will always be printed for any warning that's severe enough to halt the Cactus run. It also implies that a level 0 warning is guaranteed (to be printed and) to halt the Cactus run.

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.

In any case, the Boolean flesh parameter `cctk.full.warnings` determines whether all the details about the warning origin (processor ID, line number, source file, source thorn) are shown. The default is to print everything.

## See Also

`CCTK_WARN` [\[A261\]](#) macro to print a warning message with a single string argument

## Examples

```
C      #include "cctk.h"
      #include "cctk_WarningLevel.h"

      const char *outdir;

      CCTK_VWarn(CCTK_WARN_ALERT, __LINE__, __FILE__, CCTK_THORNSTRING,
                 "Output directory '%s' could not be created", outdir);
```

**CCTK\_WARN**

---

Macro to print a single string as a warning message and possibly stop the code

**Synopsis**

```
C          #include "cctk.h"
          #include "cctk_WarnLevel.h"

          CCTK_WARN(int level, const char *message);

Fortran    #include "cctk.h"

          call CCTK_WARN(level, message)
          integer      level
          character*(*) message
```

**Parameters**

<b>level</b>	The warning level to use; see the description of <code>CCTK_VWarn()</code> on page <a href="#">A258</a> for a detailed discussion of this parameter and the Cactus macros for standard warning levels
<b>message</b>	The warning message to print

**Discussion**

This macro can be used by thorns to print a single string as a warning message to `stderr`.

`CCTK_WARN(level, message)` expands to a call to an internal function which is equivalent to `CCTK_VWarn()`, but without the variable-number-of-arguments feature (so it can be used from Fortran).<sup>5</sup> The macro automatically includes details about the origin of the warning (the thorn name, the source code file name and the line number where the macro occurs).

To include variables in a warning message from C, you can use the routine `CCTK_VWarn` which accepts a variable argument list. To include variables from Fortran, a string must be constructed and passed in a `CCTK_WARN` macro.

**See Also**

<code>CCTK_VWarn()</code>	prints a warning message with a variable argument list
---------------------------	--

**Examples**

```
C          #include "cctk.h"

          CCTK_WARN(CCTK_WARN_ABORT, "Divide by 0");

Fortran    #include "cctk.h"

          integer      myint
```

---

<sup>5</sup>Some code calls this internal function directly. For reference, the function is:

```
int CCTK_Warn(int level,
              int line_number, const char* file_name, const char* thorn_name,
              const char* message)
```

```
real          myreal
character*200 message

write(message, '(A32, G12.7, A5, I8)')
&      'Your warning message, including ', myreal, ' and ', myint
call CCTK_WARN(CCTK_WARN_ALERT, message)
```

**CCTK\_WarnCallbackRegister**

---

Register one or more routines for dealing with warning messages in addition to printing them to standard error

**Synopsis**

```
C      #include "cctk.h"
      #include "cctk_WarnLevel.h"

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

**Parameters**

<code>minlevel</code>	The minimum warning level to use. You can find a detailed discussion of the Cactus macros for standard warning levels on page <a href="#">A258</a> . Both <code>minlevel</code> and <code>maxlevel</code> follow that definition.
<code>maxlevel</code>	The maximum warning level to use
<code>data</code>	The void pointer holding extra information about the registered call back routine
<code>callback</code>	The function pointer pointing to the call back function dealing with warning messages. The definition of the function pointer is:

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

The argument list is the same as those in `CCTK_Warn()` (see the footnote of `CCTK_WARN()` page [A261](#)) except an extra void pointer to hold the information about the call back routine.

**Discussion**

This function can be used by thorns to register their own routines to deal with warning messages. The registered function pointers will be stored in a pointer chain. When `CCTK_VWarn()` is called, the registered routines will be called in the same order as they get registered in addition to dumping warning messages to `stderr`.

The function can only be called in C.

**See Also**

<code>CCTK_InfoCallbackRegister()</code>	Register one or more routines for dealing with information messages in addition to printing them to screen
<code>CCTK_VWarn()</code>	Prints a formatted string with a variable argument list as a warning message to standard error and possibly stops the code

## Examples

```
C      /*DumpWarn will dump warning messages to a file*/

void DumpWarn(int level,
               int line,
               const char *file,
               const char *thorn,
               const char *message,
               void *data)
{
    DECLARE_CCTK_PARAMETERS
    FILE *fp;
    char *str = (char *)malloc((strlen(file)+strlen(thorn)+strlen(message)+100);

    /*warn_dump_file is a string set in the parameter file*/

    if((fp = fopen (warn_dump_file, "a"))==0)
    {
        fprintf(stderr, "fatal error: can not open the file %s\n",warn_dump_file);
        return;
    }
    sprintf(str, "\n[WARN]\nLevel->%d\nLine->%d\nFile->%s\nThorn->%s\nMsg->%s\n",
            level,line,file,thorn,message);
    fprintf(fp, "%s", str);
    free(str);
    fclose(fp);
}

...

/*minlevel = 0; maxlevel = 5; data = NULL; callback = DumpWarn*/

CCTK_WarnCallbackRegister(0,5,NULL,DumpWarn);
```

## Part B

# Util\_\* Functions Reference

---

In this chapter all `Util_*`() Cactus utility functions are described. These are low-level functions mainly for more complicated programming, which are used by the rest of Cactus, but don't depend heavily on it. Some of them are callable from Fortran or C, but many are C-only.

# Chapter B1

## Functions Alphabetically

Here the functions are listed alphabetically within each section.

### B1.1 Miscellaneous Functions

`Util_CurrentDate`    [\[B7\]](#) Fills string with current local date

`Util_CurrentDateTime`    [\[B8\]](#) Returns the current datetime in a machine-processable format as defined in ISO 8601 chapter 5.4.

`Util_CurrentTime`    [\[B9\]](#) Fills string with current local time

`Util_snprintf`    [\[B10\]](#) Safely format data into a caller-supplied buffer.

`Util_vsnprintf`    [\[B12\]](#) Safely format data into a caller-supplied buffer.

### B1.2 String Functions

`Util_StrCmpi`    [\[B14\]](#) Compare two strings, ignoring upper/lower case.

`Util_Strdup`    [\[B16\]](#) “Duplicate” a string, i.e. copy it to a newly-allocated buffer.

`Util_Strlcat`    [\[B18\]](#) Concatenate two strings safely.

`Util_Strlcpy`    [\[B20\]](#) Copy a string safely.

`Util_StrSep`    [\[B22\]](#) Separate first token from a string.

### B1.3 Table Functions

`Util_TableClone`    [\[B26\]](#) Create a new table which is a “clone” (exact copy) of an existing table

`Util_TableCreate`    [\[B28\]](#) Create a new (empty) table



**Util\_TableCreateFromString**

[B30] Create a new table (with the case-insensitive flag set) and sets values in it based on a string argument (interpreted with “parameter-file” semantics)

**Util\_TableDeleteKey**

[B32] Delete a specified key/value entry from a table

**Util\_TableDestroy** [B33] Destroy a table**Util\_TableGet\***

[B34] This is a family of functions, one for each Cactus data type, to get the single (1-element array) value, or more generally the first array element of the value, associated with a specified key in a key/value table.

**Util\_TableGet\*Array**

[B36] This is a family of functions, one for each Cactus data type, to get a copy of the value associated with a specified key, and store it (more accurately, as much of it as will fit) in a specified array

**Util\_TableGetGeneric**

[B38] Get the single (1-element array) value, or more generally the first array element of the value, associated with a specified key in a key/value table; the value’s data type is generic

**Util\_TableGetGenericArray**

[B40] Get a copy of the value associated with a specified key, and store it (more accurately, as much of it as will fit) in a specified array; the array’s data type is generic

**Util\_TableGetString**

[B43] Gets a copy of the character-string value associated with a specified key in a table, and stores it (more accurately, as much of it as will fit) in a specified character string

**Util\_TableItAdvance**

[B45] Advance a table iterator to the next entry in the table

**Util\_TableItClone**

[B46] Creates a new table iterator which is a “clone” (exact copy) of an existing table iterator

**Util\_TableItCreate**

[B48] Create a new table iterator

**Util\_TableItDestroy**

[B49] Destroy a table iterator

**Util\_TableItQueryIsNonNull**

[B50] Query whether a table iterator is *not* in the “null-pointer” state

**Util\_TableItQueryIsNull**

[B51] Query whether a table iterator is in the “null-pointer” state

**Util\_TableItQueryKeyValueInfo**

[B52] Query the key and the type and number of elements of the value corresponding to that key, of the table entry to which an iterator points

**Util\_TableItQueryTableHandle**

[B55] Query what table a table iterator iterates over

**Util\_TableItResetToStart**

[B56] Reset a table iterator to point to the starting table entry

- `Util_TableItSetToKey`  
[B57] Set a key/value iterator to point to a specified entry in the table.
- `Util_TableItSetToNull`  
[B58] Set a key/value iterator to the “null-pointer” state.
- `Util_TableQueryFlags`  
[B59] Query a table’s flags word
- `Util_TableQueryValueInfo`  
[B61] Query whether or not a specified key is in the table, and optionally the type and/or number of elements of the value corresponding to this key
- `Util_TableQueryMaxKeyLength`  
[B63] Query the maximum key length in a table
- `Util_TableQueryNKeys`  
[B64] Query the number of key/value entries in a table
- `Util_TableSet*` [B65] This is a family of functions, one for each Cactus data type, to set the value associated with a specified key to be a specified single (1-element array) value
- `Util_TableSet*Array`  
[B67] This is a family of functions, one for each Cactus data type, to set the value associated with a specified key to be a copy of a specified array
- `Util_TableSetFromString`  
[B69] Sets values in a table based on a string argument (interpreted with “parameter-file” semantics)
- `Util_TableSetGeneric`  
[B72] Set the value associated with a specified key to be a specified single (1-element array) value, whose data type is generic
- `Util_TableSetGenericArray`  
[B74] Set the value associated with a specified key to be a copy of a specified array, whose data type is generic
- `Util_TableSetString`  
[B77] Sets the value associated with a specified key in a table, to be a copy of a specified C-style null-terminated character string

## Chapter B2

# Full Descriptions of Miscellaneous Functions

**Util\_CurrentDate**

---

Fills string with current local date

**Synopsis**

```
C          #include "cctk.h"
          #include "cctk_Misc.h.h"

          int retval = Util_CurrentDate (int len, char *now);
```

**Parameters**

**len**                   length of the user-supplied string buffer  
**now**                   user-supplied string buffer to write the date stamp to

**Result**

**retval**               length of the string returned in now, or 0 if the string was truncated

**See Also**

**Util\_CurrentTime** [\[B9\]](#)               Fills string with current local time  
**Util\_CurrentDateTime** [\[B8\]](#)           Returns the current datetime in a machine-processable format as defined in ISO 8601 chapter 5.4.

**Util\_CurrentDateTime**

---

Returns the current datetime in a machine-processable format as defined in ISO 8601 chapter 5.4.

**Synopsis**

```
C          #include "cctk.h"
          #include "cctk_Misc.h.h"

          char *current_datetime = Util_CurrentDateTime ();
```

**Result**

**current\_datetime** Pointer to an allocated formatted string containing the current datetime stamp. The pointer should be freed by the caller.

**Discussion**

The formatted string returned contains the current datetime in a machine-processable format as defined in ISO 8601 chapter 5.4: "YYYY-MM-DDThh:mm:ss+hh:mm"

**See Also**

<b>Util_CurrentDate</b> <a href="#">[B7]</a>	Fills string with current local date
<b>Util_CurrentTime</b> <a href="#">[B9]</a>	Fills string with current local time

## Util\_CurrentTime

---

Fills string with current local time

### Synopsis

```
C          #include "cctk.h"
          #include "cctk_Misc.h.h"

          int retval = Util_CurrentTime (int len, char *now);
```

### Parameters

**len**                      length of the user-supplied string buffer  
**now**                      user-supplied string buffer to write the time stamp to

### Result

**retval**                      length of the string returned in now, or 0 if the string was truncated

### See Also

**Util\_CurrentDate** [\[B7\]](#)                      Fills string with current local date  
**Util\_CurrentDateTime** [\[B8\]](#)                      Returns the current datetime in a machine-processable format as defined in ISO 8601 chapter 5.4.

## Util\_snprintf

---

Safely format data into a caller-supplied buffer.

### Synopsis

```
C          #include "util_String.h"
          int count = Util_snprintf(char* buffer, size_t size, const char* format, ...)
```

### Result

**result\_len**      The number of characters (not including the trailing NUL) that would have been stored in the destination string if **size** had been infinite.

### Parameters

**buffer**            A non-NULL pointer to the (caller-provided) buffer.  
**size**                The size of the buffer pointed to by **buffer**.  
**format**            A (non-NULL pointer to a) C-style NUL-terminated string describing how to format any further arguments  
**...**                Zero or more further arguments, with types as specified by the **format** argument.

### Discussion

C99 defines, and many systems provide, a C library function **snprintf()**, which safely formats data into a caller-supplied buffer. However, a few systems don't provide this,<sup>1</sup> so Cactus provides its own version, **Util\_snprintf()**.<sup>2</sup>

The interpretation of **format** is the same as that of **printf()**. See the **printf()** documentation on your favorite computer system (notably, on any Unix system, type “**man printf**”) for lots and lots of details.

**Util\_snprintf()** stores at most **size** characters in the destination buffer; the last character it stores is always the terminating NUL character. If **result\_len** >= **size** then the destination string was truncated to fit into the destination buffer.

### See Also

<b>Util_vsnprintf</b> [B12]	Similar function which takes a <stdarg.h> variable argument list.
<b>snprintf()</b>	Standard C library function which this function tries to clone.
<b>sprintf()</b>	Unsafe and dangerous C library function similar to <b>snprintf()</b> , which doesn't check the buffer length.

### Errors

< 0	Some sort of error occurred. It's indeterminate what (if anything) has been stored in the destination buffer.
-----	---

### Examples

```
C          #include "util_String.h"
```

---

<sup>1</sup>There's also a related (older) API **sprintf()**. Don't use it – it almost guarantees buffer overflows.

<sup>2</sup>Contrary to the usual Cactus convention, the “s” in “**Util\_snprintf**” is in *lower* case, not upper case.

```
/* some values to be formatted */
char   c = '@';
int     i = 42;
double d = 3.14159265358979323846;
const char s[] = "this is a string to format";

int len;
#define N_BUFFER 100
char buffer[N_BUFFER];

/* safely format the values into the buffer */
Util_snprintf(buffer, N_BUFFER,
               "values are c='%c' i=%d d=%g s=\"%s\"",
               c, i, d, s);

/*
 * same as above example, but now explicitly check for the output
 * being truncated due to the buffer being too small
 */
const int len = Util_snprintf(buffer, N_BUFFER,
                              "values are c='%c' i=%d d=%g s=\"%s\"",
                              c, i, d, s);

if (len >= N_BUFFER)
{
    /*
     * output was truncated (i.e. buffer was too small)
     * ( buffer probably doesn't have all the information we wanted
     * but the code is still "safe", in the sense that buffer is
     * still NUL-terminated, and no buffer-overflow has occurred)
     */
}
```



**Util\_vsnprintf**

---

Safely format data into a caller-supplied buffer.

**Synopsis**

```
C          #include "util_String.h"
          int count = Util_vsnprintf(char* buffer, size_t size, const char* format,
                                   va_list arg)
```

**Discussion**

This function is identical to `Util_snprintf`, except that it takes its data arguments in the form of a `va_list` “cookie” (as defined by `<stdarg.h>`, which is already included by `"util_String.h"`), instead of in the form of a variable length argument list.

**See Also**

<code>Util_snprintf</code> <a href="#">[B10]</a>	Similar function which takes a variable length argument list.
<code>vsnprintf()</code>	Standard C library function which this function tries to clone.
<code>vsprintf()</code>	Unsafe and dangerous C library function similar to <code>vsnprintf()</code> , which doesn’t check the buffer length.
<code>&lt;stdarg.h&gt;</code>	System header file which defines the <code>va_list</code> “cookie” type and various macros to manipulate it. On most Unix systems the man page for this header file this also includes a mini-tutorial on how to use <code>va_list</code> objects.

## Chapter B3

# Full Descriptions of String Functions

**Util\_StrCmpi**

---

Compare two strings, ignoring upper/lower case.

**Synopsis**

```
C          #include "util_String.h"
          int cmp = Util_StrCmpi(const char *str1, const char *str2);
```

**Result**

**cmp**            An integer which is:

- < 0    if **str1** < **str2** in lexicographic order ignoring upper/lower case distinctions
- 0     if **str1** = **str2** ignoring upper/lower case distinctions
- > 0    if **str1** > **str2** in lexicographic order ignoring upper/lower case distinctions

**Parameters**

**str1**            A non-NULL pointer to a (C-style NUL-terminated) string to be compared.

**str2**            A non-NULL pointer to a (C-style NUL-terminated) string to be compared.

**Discussion**

The standard C library `strcmp()` function does a *case-sensitive* string comparison, i.e. `strcmp("cactus", "Cactus")` will find the two strings not equal. Sometimes it's useful to do *case-insensitive* string comparison, where upper/lower case distinctions are ignored. Many systems provide a `strcascmp()` or `strcmpi()` function to do this, but some systems don't, and even on those that do, the name isn't standardised. So, Cactus provides its own version, `Util_StrCmpi()`.

Notice that the return value of `Util_StrCmpi()`, like that of `strcmp()`, is zero (logical "false" in C) for equal strings, and nonzero (logical "true" in C) for non-equal strings. Code of the form

```
if (Util_StrCmpi(str1, str2))
    { /* strings differ */ }
```

or

```
if (!Util_StrCmpi(str1, str2))
    { /* strings are identical apart from case distinctions */ }
```

may be confusing to readers, because the sense of the comparison isn't immediately obvious. Writing an explicit comparison against zero make things clearer:

```
if (Util_StrCmpi(str1, str2) != 0)
    { /* strings differ */ }
```

or

```
if (Util_StrCmpi(str1, str2) == 0)
    { /* strings are identical apart from case distinctions */ }
```

Unfortunately, the basic concept of "case-insensitive" string operations doesn't generalize well to non-English character sets,<sup>1</sup> where lower-case ↔ upper-case mappings

---

<sup>1</sup>Hawaiian and Swahili are apparently the only other living languages that use solely the 26-letter "English" Latin alphabet.

may be context-dependent, many-to-one, and/or time-dependent.<sup>2</sup> At present Cactus basically ignores these issues. :(

### See Also

`strcmp()` Standard C library function (prototype in `<string.h>`) to compare two strings.

### Examples

```
C      #include "util_String.h"

      /* does the Cactus parameter driver specify the PUGH driver? */
      /* (Cactus parameters are supposed to be case-insensitive) */
      if (Util_StrCmpi(driver, "pugh") == 0)
          { /* PUGH code */ }
      else
          { /* non-PUGH code */ }
```

---

<sup>2</sup>For example, the (lower-case) German “ß” doesn’t have a unique upper-case equivalent: “ß” usually maps to “SS” (for example “groß” ↔ “GROSS”), *but* if that would conflict with another word, then “ß” maps to “SZ” (for example “maße” ↔ “MASZE” because there’s a different word “MASSE”). Or at least that’s the way it was prior to 1998. The 1998 revisions to German orthography removed the SZ rule, so now (post-1998) the two distinct German words “masse” (English “mass”) and “maße” (“measures”) have identical upper-case forms “MASSE”. To further complicate matters, (the German-speaking parts of) Switzerland have a slightly different orthography, which never had the SZ rule.

French provides another tricky example: In France “é” ↔ “É” and “è” ↔ “È”, whereas in (the French-speaking parts of) Canada there are no accents on upper-case letters, so “é” ↔ “E” and “è” ↔ “E”.

## Util\_Strdup

---

“Duplicate” a string, i.e. copy it to a newly-allocated buffer.

### Synopsis

```
C          #include "util_String.h"
          char* copy = Util_Strdup(const char *str);
```

### Result

**copy**            A pointer to a buffer obtained from `malloc()`, which this function sets to a copy of the (C-style NUL-terminated) string `str`. This buffer should be freed with `free()` when it's not needed any more.

### Parameters

**str**            A non-NULL pointer to a (C-style NUL-terminated) string.

### Discussion

Many systems have a C library function `strdup()`, which `mallocs` sufficient memory for a copy of its argument string, does the copy, and returns a pointer to the copy. However, some systems lack this function, so Cactus provides its own version, `Util_Strdup()`.

### See Also

<code>&lt;stdlib.h&gt;</code>	System header file containing prototypes for <code>malloc()</code> and <code>free</code> .
<code>strcpy()</code>	Standard C library function (prototype in <code>&lt;string.h&gt;</code> ) to copy a string to a buffer. <i>This does not check that the buffer is big enough to hold the string, and is thus very dangerous. Use <code>Util_Strncpy()</code> instead!</i>
<code>Util_Strncpy()</code> <a href="#">[B20]</a>	Safely copy a string.

### Errors

**NULL**            `malloc()` was unable to allocate memory for the buffer.

### Examples

```
C          #include "util_String.h"

          /*
           * return the (positive) answer to a question,
           * or negative if an error occurred
           */
          int answer_question(const char* question)
          {
              /*
               * we need to modify the question string in the process of parsing it
               * but we must not destroy the input ==> copy it and modify the copy
               */
          }
```

```
* ... note the const qualifier on question_copy says that
*     the pointer question_copy won't itself change, but
*     we can modify the string that it points to
*/
char* const question_copy = Util_Strdup(question);
if (question_copy == NULL)
    { return -1; }    /* couldn't get memory for copy buffer */

/* code that will modify question_copy */

free(question_copy);
return 42;
}
```

## Util\_Strlcat

---

Concatenate strings safely.

### Synopsis

```
C          #include "util_String.h"
          size_t result_len = Util_Strlcat(char *dst, const char *src, size_t size);
```

### Result

**result\_len**      The size of the string the function tried to create, i.e. the initial `strlen(dst)` plus `strlen(src)`.

### Parameters

**dst**            A non-NULL pointer to the (C-style NUL-terminated) destination string.  
**src**            A non-NULL pointer to the (C-style NUL-terminated) source string.  
**size**           The size of the destination buffer.

### Discussion

The standard `strcat()` and `strcpy()` functions provide no way to specify the size of the destination buffer, so code using these functions is often vulnerable to buffer overflows. The standard `strncat()` and `strncpy()` functions can be used to write safe code, but their API is cumbersome, error-prone, and sometimes surprisingly inefficient:

- Their **size** arguments are the number of characters *remaining* in the destination buffer, which must often be calculated at run-time, and is prone to off-by-one errors.
- `strncpy()` doesn't always NUL-terminate the destination string.
- `strncpy()` NUL-fills the remainder of the buffer not used for the source string; this NUL-filling can be *very* expensive.

To solve these problems, the OpenBSD project developed the `strlcat()` and `strlcpy()` functions. See <http://www.openbsd.org/papers/strlcpy-paper.ps> for a history and general discussion of these functions. Some other Unix systems (notably Solaris) now provide these, but many don't, so Cactus provides its own versions, `Util_Strlcat()` and `Util_Strlcpy()`.

`Util_Strlcat()` appends the NUL-terminated string **src** to the end of the NUL-terminated string **dst**. It will append at most `size - strlen(dst) - 1` characters (hence it never overflows the destination buffer), and it always leaves **dst** string NUL-terminated.

### See Also

`strcat()`            Standard C library function (prototype in `<string.h>`) to concatenate two strings. *This does not check that the buffer is big enough to hold the result, and is thus very dangerous. Use `Util_Strlcat()` instead!*

`Util_Strlcpy()` [B20]      Safely copy a string.

## Examples

```
C      #include "util_String.h"

      /*
      * safely concatenate strings s1,s2,s3 into buffer:
      * ... this code is safe (it will never overflow the buffer), but
      *    quick-n-dirty in that it doesn't give any error indication
      *    if the result is truncated to fit in the buffer
      */
      #define BUFFER_SIZE      1024
      char buffer[BUFFER_SIZE];

      Util_Strncpy(buffer, s1, sizeof(buffer));
      Util_Strlcat(buffer, s2, sizeof(buffer));
      Util_Strlcat(buffer, s3, sizeof(buffer));

C      #include "util_String.h"

      #define OK                0
      #define ERROR_TRUNC      1

      /*
      * safely concatenate strings s1,s2,s3 into buffer[N_buffer];
      * return OK if ok, ERROR_TRUNC if result was truncated to fit in buffer
      */
      int cat3(int N_buffer, char buffer[],
               const char s1[], const char s2[], const char s3[])
      {
      int length;

      length = Util_Strncpy(buffer, s1, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      length = Util_Strlcat(buffer, s2, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      length = Util_Strlcat(buffer, s3, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      return OK;                                   /*** NORMAL RETURN ***/
      }
```



## Util\_Strlcpy

---

Copies a string safely.

### Synopsis

```
C          #include "util_String.h"
          size_t result_len = Util_Strlcpy(char *dst, const char *src, size_t size);
```

### Result

**result\_len**      The size of the string the function tried to create, i.e. `strlen(src)`.

### Parameters

**dst**              A non-NULL pointer to the (C-style NUL-terminated) destination string.  
**src**              A non-NULL pointer to the (C-style NUL-terminated) source string.  
**size**             The size of the destination buffer.

### Discussion

The standard `strcat()` and `strcpy()` functions provide no way to specify the size of the destination buffer, so code using these functions is often vulnerable to buffer overflows. The standard `strncat()` and `strncpy()` functions can be used to write safe code, but their API is cumbersome, error-prone, and sometimes surprisingly inefficient:

- Their **size** arguments are the number of characters *remaining* in the destination buffer, which must often be calculated at run-time, and is prone to off-by-one errors.
- `strncpy()` doesn't always NUL-terminate the destination string.
- `strncpy()` NUL-fills the remainder of the buffer not used for the source string; this NUL-filling can be *very* expensive.

To solve these problems, the OpenBSD project developed the `strlcat()` and `strlcpy()` functions. See <http://www.openbsd.org/papers/strlcpy-paper.ps> for a history and general discussion of these functions. Some other Unix systems (notably Solaris) now provide these, but many don't, so Cactus provides its own versions, `Util_Strlcat()` and `Util_Strlcpy()`.

`Util_Strlcpy()` copies up to **size**-1 characters from the source string to the destination string, followed by a NUL character (so **dst** is always NUL-terminated). Unlike `strncpy()`, `Util_Strlcpy()` does *not* fill any left-over space at the end of the destination buffer with NUL characters.

### See Also

**strcpy()**              Standard C library function (prototype in `<string.h>`) to copy a string to a buffer. *This does not check that the buffer is big enough to hold the string, and is thus very dangerous. Use `Util_Strlcpy()` instead!*

**Util\_Strdup()** [B16]      "Duplicate" a string, i.e. copy it to a newly-allocated buffer.

**Util\_Strlcat()** [B18]      Safely concatenates two strings.

**Examples**

```
C      #include "util_String.h"

      /*
      * safely concatenate strings s1,s2,s3 into buffer:
      * ... this code is safe (it will never overflow the buffer), but
      *    quick-n-dirty in that it doesn't give any error indication
      *    if the result is truncated to fit in the buffer
      */
      #define BUFFER_SIZE      1024
      char buffer[BUFFER_SIZE];

      Util_Strncpy(buffer, s1, sizeof(buffer));
      Util_Strncat(buffer, s2, sizeof(buffer));
      Util_Strncat(buffer, s3, sizeof(buffer));

C      #include "util_String.h"

      #define OK                0
      #define ERROR_TRUNC      1

      /*
      * safely concatenate strings s1,s2,s3 into buffer[N_buffer];
      * return OK if ok, ERROR_TRUNC if result was truncated to fit in buffer
      */
      int cat3(int N_buffer, char buffer[],
               const char s1[], const char s2[], const char s3[])
      {
      int length;

      length = Util_Strncpy(buffer, s1, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      length = Util_Strncat(buffer, s2, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      length = Util_Strncat(buffer, s3, N_buffer);
      if (length >= N_buffer)
          return ERROR_TRUNC;                      /*** ERROR EXIT ***/

      return OK;                                    /*** NORMAL RETURN ***/
      }
```

## Util\_StrSep

---

Separate off the first token from a string.

### Synopsis

```
C          #include "util_String.h"
          char* token = Util_StrSep(const char** string_ptr, const char* delim_set);
```

### Result

**token**                This function returns the original value of **\*string\_ptr**, or NULL if the end of the string is reached.

### Parameters

**string\_ptr**           A non-NULL pointer to a (modifyable) non-NULL pointer to the (C-style NUL-terminated) string to operate on.

**delim\_set**            A non-NULL pointer to a (C-style NUL-terminated) string representing a set of delimiter characters (the order of these characters doesn't matter).

### Discussion

Many Unix systems define a function **strsep()** which provides a clean way of splitting a string into "words". However, some systems only provide the older (and inferior-in-several-ways) **strtok()** function, so Cactus implements its own **strsep()** function, **Util\_StrSep()**.

**Util\_StrSep()** finds the first occurrence in the string pointed to by **\*string\_ptr** of any character in the string pointed to by **delim\_set** (or the terminating NUL if there is no such character), and replaces this by NUL. The location of the next character after the NUL character just stored (or NULL, if the end of the string was reached) is stored in **\*string\_ptr**.

An "empty" field, i.e. one caused by two adjacent delimiter characters, can be detected (after **Util\_StrSep()** returns) by the test **\*\*string\_ptr == '\0'**, or equivalently **strlen(\*string\_ptr) == 0**.

See the example section below for the typical usage of **Util\_StrSep()**.

### See Also

**strsep()**                Some systems provide this in the standard C library (prototype in **<string.h>**); **Util\_StrSep()** is a clone of this.

**strtok()**                Inferior API for splitting a string into tokens (defined by the ANSI/ISO C standard).

### Examples

```
C          #include <stdio.h>
          #include <stdlib.h>
          #include "util_String.h"

          /* prototypes */
          int parse_string(char* string,
```

```
int N_argv, char* argv[]);

/*
 * Suppose we have a Cactus parameter gridfn_list containing a
 * whitespace-separated list of grid functions. This function
 * "processes" (here just prints the name of) each grid function.
 */
void process_gridfn_list(const char* gridfn_list)
{
#define MAX_N_GRIDFNS 100
int N_gridfns;
int i;
char* copy_of_gridfn_list;
char* gridfn[MAX_N_GRIDFNS];

copy_of_gridfn_list = Util_Strdup(gridfn_list);
N_gridfns = parse_string(copy_of_gridfn_list,
                        MAX_N_GRIDFNS, gridfn);

    for (i = 0 ; i < N_gridfns ; ++i)
    {
        /* "process" (here just print the name of) each gridfn */
        printf("grid function %d is \"%s\"\n", i, gridfn[i]);
    }

free(copy_of_gridfn_list);
}

/*
 * This function parses a string containing whitespace-separated
 * tokens into a main()-style argument vector (of size N_argv ).
 * This function returns the number of pointers stored into argv[] .
 *
 * Adjacent sequences of whitespace are treated the same as single
 * whitespace characters.
 *
 * Note that this function this modifies its input string; see
 * Util_Strdup() if this is a problem
 */
int parse_string(char* string,
                int N_argv, char* argv[])
{
int i;

    for (i = 0 ; i < N_argv ; )
    {
        argv[i] = Util_StrSep(&string, " \t\n\r\v");
        if (argv[i] == NULL)
            { break; }          /* reached end-of-string */

        if (*argv[i] == '\0')
            {

```

```
        /*
        * found a 0-length "token" (a sequence of
        * two or more adjacent whitespace characters)
        * ==> skip this "token" (don't store it)
        * ==> no-op here
        */
    }
else {
    /* token has length > 0 ==> store it */
    ++i;
}

return i;
}
```

## Chapter B4

# Full Descriptions of Table Functions

## Util\_TableClone

---

Creates a new table which is a “clone” (exact copy) of an existing table

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int clone_handle = Util_TableClone(int handle);

Fortran    call Util_TableClone(clone_handle, handle)
          integer clone_handle, handle
```

### Result

clone\_handle ( $\geq 0$ )  
A handle to the clone table

### Parameters

handle            Handle to the table to be cloned

### Discussion

Viewing a table as a set of key/value pairs, this function creates a new table (with the same flags word as the original) containing copies of all the original table’s key/value pairs. The two tables are completely independent, i.e. future changes to one won’t affect the other.

Note that if there are any `CCTK_POINTER` and/or `CCTK_FPOINTER` values in the table, they are “shallow copied”, i.e. the (pointer) values in the table are copied. This results in the clone table’s pointer values pointing to the same places as the original table’s pointer values. Be careful with this! In particular, if you’re using pointer values in the table to keep track of `malloc()` memory, be careful not to `free()` the same block of memory twice!

Note that table iterators are *not* guaranteed to sequence through the original and clone tables in the same order. (This is a special case of the more general “non-guarantee” in the Section of table iterators in the Users’ Guide: the order of table iterators may differ even between different tables with identical key/value contents.)

### See Also

`Util_TableCreate()` [\[B28\]](#)            create a table  
`Util_TableCreateFromString()` [\[B30\]](#)       convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string  
`Util_TableDestroy()` [\[B33\]](#)            destroy a table

### Errors

`UTIL_ERROR_NO_MEMORY`            unable to allocate memory  
`UTIL_ERROR_TABLE_BAD_FLAGS`       flags word is negative in the to-be-cloned table (this indicates an internal error in the table routines, and should never happen)

**Examples**

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"

      /*
       * This function is passed (a handle to) a table containing some entries.
       * It needs to set some additional entries and pass the table to some
       * other function(s), but it also needs to leave the original table
       * intact for other use by the caller. The solution is to clone the
       * original table and work on the clone, leaving the original table
       * unchanged.
       */
      int my_function(int handle, int x, int y)
      {
          int status;

          /* clone the table */
          const int clone_handle = Util_TableClone(handle)
          if (clone_handle < 0)
              return clone_handle;          /* error in cloning table */

          /* now set our entries in the clone table */
          status = Util_TableSetInt(clone_handle, x, "x");
          if (status < 0)
              return status;                /* error in setting x */
          status = Util_TableSetInt(clone_handle, y, "y");
          if (status < 0)
              return status;                /* error in setting y */

          /* ... code to use the clone table ... */
          /* ... eg pass clone_handle to other functions ... */

          /* we're done with the clone now */
          Util_TableDestroy(clone_handle);
          return 0;
      }
```



## Util\_TableCreate

---

Creates a new (empty) table

### Synopsis

<b>C</b>	<pre>#include "util_ErrorCodes.h" #include "util_Table.h" int handle = Util_TableCreate(int flags);</pre>
<b>Fortran</b>	<pre>call Util_TableCreate(handle, flags) integer handle, flags</pre>

### Result

`handle` ( $\geq 0$ )      A handle to the newly-created table

### Parameters

`flags` ( $\geq 0$ )      A flags word for the table. This should be the inclusive-or of zero or more of the `UTIL_TABLE_FLAGS_*` bit masks (defined in `"util_Table.h"`). For Fortran users, note that inclusive-or is the same as sum here, since the bit masks are all disjoint.

### Discussion

We require the flags word to be non-negative so that other functions can distinguish flags from (negative) error codes.

Any User-defined flag words should use only bit positions at or above `UTIL_TABLE_FLAGS_USER_DEFINED_BASE`, i.e. all bit positions below this are reserved for present or future Cactus use.

At present there is only a single flags-word bit mask defined in `"util_Table.h"`:

#### `UTIL_TABLE_FLAGS_CASE_INSENSITIVE`

By default keys are treated as C-style character strings, and the table functions compare them with the standard C `strcmp` function. However, by setting the `UTIL_TABLE_FLAGS_CASE_INSENSITIVE` bit in the flags word, this table's keys may be made case-insensitive, i.e. the table routines then compare this table's keys with `Util_StrCmpi()`. Note that keys are still *stored* exactly as the caller specifies them (i.e. they are *not* forced into a canonical case); it's only their *comparison* that's affected by this flag.

### See Also

<code>Util_StrCmpi()</code> [B14]	compare two strings, ignoring upper/lower case
<code>Util_TableClone()</code> [B26]	create a new table which is a "clone" (exact copy) of an existing table
<code>Util_TableCreateFromString()</code> [B30]	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableDestroy()</code> [B33]	destroy a table

### Errors

<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory
-----------------------------------	---------------------------

UTIL\_ERROR\_TABLE\_BAD\_FLAGS      flags word is negative

### Examples

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"

      /* create a table, simplest case */
      int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

      /* create a table whose keys will be treated as case-insensitive */
      int handle2 = Util_TableCreate(UTIL_TABLE_FLAGS_CASE_INSENSITIVE);
```

## Util\_TableCreateFromString

---

Creates a new table (with the case-insensitive flag set) and sets values in it based on a string argument (interpreted with “parameter-file” semantics)

### Synopsis

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"
           int handle = Util_TableCreateFromString(const char *string);

Fortran    call Util_TableCreateFromString(handle, string)
           integer      handle
           character*(*) string
```

### Result

handle ( $\geq 0$ )      a handle to the newly-created table

### Parameters

string              a pointer to a C-style null-terminated string specifying the table contents; see the description for `Util_TableSetFromString()` for a full description of the syntax and semantics of this string

### See Also

`Util_TableClone()` [B26]              Create a new table which is a “clone” (exact copy) of an existing table  
`Util_TableCreate()` [B28]              create a table  
`Util_TableSetFromString()` [B69]      sets values in a table based on a string argument

### Errors

UTIL\_ERROR\_NO\_MEMORY              unable to allocate memory  
UTIL\_ERROR\_BAD\_KEY                  invalid input: key contains invalid character  
UTIL\_ERROR\_BAD\_INPUT                invalid input: can’t parse input string  
other error codes                    this function may also return any error codes returned by `Util_TableCreate()` or `Util_TableSetFromString()`

### Examples

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"

           int handle = Util_TableCreateFromString("order = 3\t"
                                                    "myreal = 42.314159\t"
                                                    "mystring = 'hello'\t"
                                                    "myarray = { 0 1 2 3 }");

           /* equivalent code to the above */
           int handle = Util_TableCreate(UTIL_TABLE_FLAGS_CASE_INSENSITIVE);
           Util_TableSetFromString(handle, "order = 3\t"
```

```
        "myreal = 42.314159\t"
        "mystring = 'hello'"
        "myarray = { 0 1 2 3 }");

/* also equivalent to the above */
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_CASE_INSENSITIVE);
CCTK_INT array[] = {0, 1, 2, 3};

Util_TableSetInt(handle, 3, "order");
Util_TableSetReal(handle, 42.314159, "myreal");
Util_TableSetString(handle, "hello", "mystring");
Util_TableSetIntArray(handle, 4, array, "myarray");
```

## Util\_TableDeleteKey

---

Deletes a specified key/value entry from a table

### Synopsis

<b>C</b>	<pre>#include "util_ErrorCodes.h" #include "util_Table.h" int key_exists = Util_TableDeleteKey(int handle, const char *key);</pre>
<b>Fortran</b>	<pre>call Util_TableDeleteKey(key_exists, handle, key) integer      key_exists, handle character*(*) key</pre>

### Result

0	ok (key existed before this call, and has now been deleted)
---	---

### Parameters

handle ( $\geq 0$ )	handle to the table
key	a pointer to the key (a C-style null-terminated string)

### Discussion

This function invalidates any iterators for the table which are not in the “null-pointer” state.

### Errors

UTIL_ERROR_BAD_HANDLE	handle is invalid
UTIL_ERROR_TABLE_BAD_KEY	key contains '/' character
UTIL_ERROR_TABLE_NO_SUCH_KEY	no such key in table

## Util\_TableDestroy

---

Destroys a table

### Synopsis

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"
           int status = Util_TableDestroy(int handle);

Fortran    call Util_TableDestroy(status, handle)
           integer status, handle
```

### Result

0                      ok

### Parameters

handle ( $\geq 0$ )      handle to the table

### Discussion

Of course, this function invalidates any and all iterators for the table. :)

### See Also

Util\_TableClone() [\[B26\]](#)              Create a new table which is a “clone” (exact copy) of an existing table

Util\_TableCreate() [\[B28\]](#)              create a table

Util\_TableCreateFromString() [\[B30\]](#)      convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string

### Errors

UTIL\_ERROR\_BAD\_HANDLE              handle is invalid

### Examples

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"

           /* create a table */
           int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

           /* do things with the table: put values in it, */
           /* pass its handle to other functions, etc etc */
           /* ... */

           /* at this point we (and all other functions we */
           /* may call in the future) are done with the table */
           Util_TableDestroy(handle);
```

**Util\_TableGet\***

---

This is a family of functions, one for each Cactus data type, to get the single (1-element array) value, or more generally the first array element of the value, associated with a specified key in a key/value table.

**Synopsis**

**C**                    `#include "util_ErrorCodes.h"`  
                      `#include "util_Table.h"`  
                      `int N_elements = Util_TableGetXxx(int handle,`  
   `CCTK_XXX *value,`  
   `const char *key);`

where XXX is one of POINTER, FPOINTER<sup>1</sup>, CHAR, BYTE, INT, INT1, INT2, INT4, INT8, REAL, REAL4, REAL8, REAL16, COMPLEX, COMPLEX8, COMPLEX16, COMPLEX32 (not all of these may be supported on any given system)

**Fortran**            `call Util_TableGetXxx(N_elements, handle, value, key)`  
                      `integer            N_elements, handle`  
                      `CCTK_XXX        value`  
                      `character*(*)    key`

where CCTK\_XXX may be any data type supported by C (above) except CCTK\_CHAR (Fortran doesn't have a separate "character" data type; use CCTK\_BYTE instead)

**Result**

**N\_elements**        the number of array elements in the value

**Parameters**

**handle** ( $\geq 0$ )     handle to the table

**value**             a pointer to where this function should store a copy of the value (or more generally the first array element of the value) associated with the specified key, or NULL pointer to skip storing this

**key**                a pointer to the key (a C-style null-terminated string)

**Discussion**

Note that it is *not* an error for the value to actually have  $> 1$  array elements; in this case only the first element is stored. The rationale for this design is that the caller may know or suspect that the value is a large array, but may only want the first array element; in this case this design avoids the caller having to allocate a large buffer unnecessarily.

In contrast, it *is* an error for the value to actually be an empty (0-length) array, because then there is no "first array element" to get.

It is also an error for the value to actually have a different type than CCTK\_XXX.

If any error code is returned, the user's value buffer (pointed to by **value** if this is non-NULL) is unchanged.

**See Also**

---

<sup>1</sup>For backwards compatability the function `Util_TableGetFnPointer()` is also provided as an alias for `Util_TableGetFPointer()`. This is deprecated as of Cactus 4.0 beta 13.

<code>Util_TableCreateFromString()</code> <a href="#">[B30]</a>	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetString()</code> <a href="#">[B43]</a>	get a character-string value
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> <a href="#">[B72]</a>	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> <a href="#">[B74]</a>	set an array value with generic data type
<code>Util_TableSetFromString()</code> <a href="#">[B69]</a>	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> <a href="#">[B77]</a>	set a character-string value

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table
<code>UTIL_ERROR_TABLE_WRONG_DATA_TYPE</code>	value has data type other than <code>CCTK_TYPE</code>
<code>UTIL_ERROR_TABLE_VALUE_IS_EMPTY</code>	value is an empty (0-element) array

### Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"

#define N_DIGITS      5
static const CCTK_INT pi_digits[N_DIGITS] = {3, 14, 159, 2653, 58979};

int N;
CCTK_INT x;
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetIntArray(handle, N_DIGITS, pi_digits, "digits of pi");
Util_TableSetIntArray(handle, 0, pi_digits, "empty array");

/* gets N = 5, x = 3 */
N = Util_TableGetInt(handle, &x, "digits of pi");

/* gets N = UTIL_ERROR_TABLE_VALUE_IS_EMPTY */
N = Util_TableGetInt(handle, &x, "empty array");
```



## Util\_TableGet\*Array

This is a family of functions, one for each Cactus data type, to get a copy of the value associated with a specified key, and store it (more accurately, as much of it as will fit) in a specified array

## Synopsis

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"
int N_elements = Util_TableGetXxxArray(int handle,
                                         int N_array, CCTK_XXX array[],
                                         const char *key);
```

where XXX is one of POINTER, FPOINTER<sup>2</sup>, CHAR, BYTE, INT, INT1, INT2, INT4, INT8, REAL, REAL4, REAL8, REAL16, COMPLEX, COMPLEX8, COMPLEX16, COMPLEX32 (not all of these may be supported on any given system)

```

Fortran      call Util_TableGetXxxArray(N_elements, handle, N_array, array, key)
               integer      N_elements, handle, N_array
               CCTK_XXX(*)   array
               character*(*) key
               where CCTK_XXX may be any data type supported by C (above)

```

## Result

<b>N_elements</b>	the number of array elements in the value
-------------------	---

## Parameters

<code>handle</code> ( $\geq 0$ )	handle to the table
<code>N_array</code>	the number of array elements in <code>array[]</code> (must be $\geq 0$ if <code>array</code> != NULL)
<code>array</code>	a pointer to where this function should store (up to <code>N_array</code> elements of) a copy of the value associated with the specified key, or NULL pointer to skip storing this
<code>key</code>	a pointer to the key (a C-style null-terminated string)

## Discussion

Note that it is *not* an error for the value to actually have  $> \text{N\_array}$  array elements; in this case only the first  $\text{N\_array}$  elements are stored. The caller can detect this by comparing the return value with  $\text{N\_array}$ . The rationale for this design is that the caller may know or suspect that the value is a large array, but may only want the first few array elements; in this case this design avoids the caller having to allocate a large buffer unnecessarily.

It is also *not* an error for the value to actually have `< N.array` array elements; again the caller can detect this by comparing the return value with `N.array`.

It *is* an error for the value to actually have a different type than CCKT\_XXX.

If any error code is returned, the user's value buffer (pointed to by `array` if this is non-NULL) is unchanged.

## See Also

<sup>2</sup>For backwards compatability the function `Util_TableGetFnPointerArray()` is also provided as an alias for `Util_TableGetFPPointerArray()`. This is deprecated as of Cactus 4.0 beta 13.

<code>Util_TableCreateFromString()</code> <a href="#">[B30]</a>	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGetGeneric()</code> <a href="#">[B38]</a>	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> <a href="#">[B40]</a>	get an array value with generic data type
<code>Util_TableGetString()</code> <a href="#">[B43]</a>	get a character-string value
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> <a href="#">[B72]</a>	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> <a href="#">[B74]</a>	set an array value with generic data type
<code>Util_TableSetFromString()</code> <a href="#">[B69]</a>	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> <a href="#">[B77]</a>	set a character-string value

## Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_BAD_INPUT</code>	<code>array != NULL</code> and <code>N_array &lt; 0</code>
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table
<code>UTIL_ERROR_TABLE_WRONG_DATA_TYPE</code>	value has data type other than <code>CCTK_TYPE</code>

## Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"

#define N_STUFF      3
static const CCTK_REAL stuff[N_STUFF] = {42.0, 69.0, 105.5};

#define N_OUTPUT      2
CCTK_INT output[N_OUTPUT];

int N;
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetRealArray(handle, N_STUFF, stuff, "blah blah blah");

/* gets N = 3, output[0] = 42.0, output[1] = 69.0 */
N = Util_TableGetRealArray(handle, N_OUTPUT, output, "blah blah blah");
```

**Util\_TableGetGeneric**

---

Get the single (1-element array) value, or more generally the first array element of the value, associated with a specified key in a key/value table; the value's data type is generic. That is, the value is specified by a CCTK\_VARIABLE\_\* type code and a void \* pointer.

**Synopsis**

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"
      int N_elements = Util_TableGetGeneric(int handle,
                                           int type_code,
                                           void *value,
                                           const char *key);

Fortran  call Util_TableGetGeneric(N_elements, handle, type_code, value, key)
         integer      N_elements, handle, type_code
         CCTK_POINTER  value
         character*(*) key
```

**Result**

N\_elements      the number of array elements in the value

**Parameters**

handle ( $\geq 0$ )      handle to the table

type\_code      the value's type code (one of the CCTK\_VARIABLE\_\* constants from "cctk\_Constants.h")

value      a pointer to where this function should store a copy of the value (or more generally the first array element of the value) associated with the specified key, or NULL pointer to skip storing this

key      a pointer to the key (a C-style null-terminated string)

**Discussion**

Note that it is *not* an error for the value to actually have  $> 1$  array elements; in this case only the first element is stored. The rationale for this design is that the caller may know or suspect that the value is a large array, but may only want the first array element; in this case this design avoids the caller having to allocate a large buffer unnecessarily.

In contrast, it *is* an error for the value to actually be an empty (0-length) array, because then there is no "first array element" to get.

It is also an error for the value to actually have a different type than that specified by type\_code.

If any error code is returned, the user's value buffer (pointed to by value if this is non-NULL) is unchanged.

**See Also**

Util\_TableCreateFromString() [\[B30\]](#)  
convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string

<code>Util_TableGet*()</code>	get a single (1-element array) value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetString()</code> [B43]	get a character-string value
<code>Util_TableQueryValueInfo()</code> [B61]	query key present/absent in table, and optionally type and/or number of elements
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table
<code>UTIL_ERROR_TABLE_WRONG_DATA_TYPE</code>	value has data type other than <code>CCTK_TYPE</code>
<code>UTIL_ERROR_TABLE_VALUE_IS_EMPTY</code>	value is an empty (0-element) array

### Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"
#include "cctk_Constants.h"

#define N_DIGITS      5
static const CCTK_INT pi_digits[N_DIGITS] = {3, 14, 159, 2653, 58979};

int N;
CCTK_INT x;
void *xptr = (void *) &x;
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetIntArray(handle, N_DIGITS, pi_digits, "digits of pi");
Util_TableSetIntArray(handle, 0, pi_digits, "empty array");

/* gets N = 5, x = 3 */
N = Util_TableGetGeneric(handle, CCTK_VARIABLE_INT, &x, "the answer");

/* gets N = UTIL_ERROR_TABLE_VALUE_IS_EMPTY, leaves x unchanged */
N = Util_TableGetGeneric(handle, CCTK_VARIABLE_INT, &x, "empty array");
```

**Util\_TableGetGenericArray**

---

Get a copy of the value associated with a specified key, and store it (more accurately, as much of it as will fit) in a specified array; the array's data type is generic. That is the array is specified by a CCTK\_VARIABLE\_\* type code, a count of the number of array elements, and a void \* pointer.

**Synopsis**

**C**

```
#include "util_ErrorCodes.h"
#include "util_Table.h"
int N_elements = Util_TableGetGenericArray(int handle,
                                           int type_code,
                                           int N_array, void *array,
                                           const char *key);
```

**Fortran**

```
call Util_TableGetGenericArray(N_elements,
                               .          handle,
                               .          type_code,
                               .          N_array, array,
                               .          key)
integer      N_elements, handle, type_code, N_array
CCTK_POINTER array
character(*) key
```

**Result**

**N\_elements**      the number of array elements in the value

**Parameters**

**handle** ( $\geq 0$ )      handle to the table

**type\_code**      the value's type code (one of the CCTK\_VARIABLE\_\* constants from "cctk\_Constants.h")

**N\_array**      the number of array elements in **array**[] (must be  $\geq 0$  if **array** != NULL)

**array**      a pointer to where this function should store (up to **N\_array** elements of) a copy of the value associated with the specified key, or NULL pointer to skip storing this

**key**      a pointer to the key (a C-style null-terminated string)

**Discussion**

Note that it is *not* an error for the value to actually have  $> \mathbf{N\_array}$  array elements; in this case only the first **N\_array** elements are stored. The caller can detect this by comparing the return value with **N\_array**. The rationale for this design is that the caller may know or suspect that the value is a large array, but may only want the first few array elements; in this case this design avoids the caller having to allocate a large buffer unnecessarily.

It is also *not* an error for the value to actually have  $< \mathbf{N\_array}$  array elements; again the caller can detect this by comparing the return value with **N\_array**.

It *is* an error for the value to actually have a different type than that specified by **type\_code**.

If any error code is returned, the user's value buffer (pointed to by **array** if this is non-NULL) is unchanged.

**See Also**

<code>Util_TableCreateFromString()</code> [B30]	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGetGeneric()</code> [B38]	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> [B40]	get an array value with generic data type
<code>Util_TableGetString()</code> [B43]	get a character-string value
<code>Util_TableQueryValueInfo()</code> [B61]	query key present/absent in table, and optionally type and/or number of elements
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

**Errors**

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_BAD_INPUT</code>	<code>array != NULL</code> and <code>N_array &lt; 0</code>
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table
<code>UTIL_ERROR_TABLE_WRONG_DATA_TYPE</code>	value has data type other than <code>CCTK_TYPE</code>

**Examples**

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"

      #define N_STUFF      3
      static const CCTK_REAL stuff[N_STUFF] = {42.0, 69.0, 105.5};

      #define N_OUTPUT      2
      CCTK_INT output[N_OUTPUT];

      int N;
      int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

      Util_TableSetRealArray(handle, N_STUFF, stuff, "stuff");

      /* gets N = UTIL_ERROR_TABLE_WRONG_DATA_TYPE, output[] unchanged */
```

```
N = Util_TableGetGenericArray(handle,
                               CCTK_VARIABLE_INT,
                               N_OUTPUT, output,
                               "stuff");
/* gets N = 3, output[0] = 42.0, output[1] = 69.0 */
N = Util_TableGetGenericArray(handle,
                               CCTK_VARIABLE_REAL,
                               N_OUTPUT, output,
                               "stuff");
```

## Util\_TableGetString

---

Gets a copy of the character-string value associated with a specified key in a table, and stores it (more accurately, as much of it as will fit) in a specified character string

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int length = Util_TableGetString(int handle,
                                          int buffer_length, char buffer[],
                                          const char *key);
```

### Result

Results are the same as all the other `Util_TableGet*()` functions:

**length**            the length of the string (C `strlen` semantics, i.e. *not* including the terminating null character)

### Parameters

**handle** ( $\geq 0$ )    handle to the table

**buffer\_length**    the length (`sizeof`) of `buffer[]` (must be  $\geq 1$  if `buffer != NULL`)

**buffer**            a pointer to a buffer into which this function should store (at most `buffer_length-1` characters of) the value, terminated by a null character as usual for C strings, or NULL pointer to skip storing this

**key**                a pointer to the key (a C-style null-terminated string)

### Discussion

This function assumes that the string is stored as an array of `CCTK_CHARS`, *not* including a terminating null character.

This function differs from `Util_TableGetCharArray()` in two ways: It explicitly provides a terminating null character for C-style strings, and it explicitly checks for the string being too long to fit in the buffer (in which case it returns `UTIL_ERROR_TABLE_STRING_TRUNCATED`).

If the error code `UTIL_ERROR_TABLE_STRING_TRUNCATED` is returned, then the first `buffer_length-1` characters of the string are returned in the user's buffer (assuming `buffer` is non-NULL), followed by a null character to properly terminate the string in the buffer. If any other error code is returned, the user's value `buffer` (pointed to by `buffer` if this is non-NULL) is unchanged.

To find out how long the string is (and thus how big of a buffer you need to allocate to avoid having the string truncated), you can call this function with `buffer_length = 0` and `buffer = NULL` (or actually anything you want); the return result will give the string length.

### See Also

`Util_TableCreateFromString()` [\[B30\]](#)  
convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string



<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetCharArray()</code> [B36]	get an array-of-CCTK_CHAR value
<code>Util_TableGetGeneric()</code> [B38]	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> [B40]	get an array value with generic data type
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetString()</code> [B77]	set a character-string value
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetCharArray()</code> [B67]	set an array-of-CCTK_CHAR value

## Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_BAD_INPUT</code>	<code>buffer != NULL</code> and <code>buffer_length ≤ 0</code>
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table
<code>UTIL_ERROR_TABLE_WRONG_DATA_TYPE</code>	value has data type other than CCTK_CHAR
<code>UTIL_ERROR_TABLE_STRING_TRUNCATED</code>	<code>buffer != NULL</code> and value was truncated to fit in <code>buffer[]</code>

## Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"

#define N_BUFFER      100
char buffer[N_BUFFER];

int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);
Util_TableSetString(handle, "relativity", "Einstein");

/* get length of string (= 10 here) */
int length = Util_TableGetString(handle, 0, NULL, "Einstein");

/* get null-terminated string into buffer, also returns 10 */
Util_TableGetString(handle, N_BUFFER, buffer, "Einstein");
```

## Util\_TableItAdvance

---

Advance a table iterator to the next entry in the table

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int is_nonnull = Util_TableItAdvance(int ihandle);
```

### Result

1	ok (iterator now points to some table entry)
0	ok (iterator has just advanced past the last table entry, and is now in the "null-pointer" state)

### Parameters

<code>ihandle (<math>\geq 0</math>)</code>	handle to the table iterator
--	------------------------------

### Discussion

If we view an iterator as an abstraction of a pointer into the table, then this function is the abstraction of the C “++” operation applied to the pointer, except that this function automatically sets the iterator to the “null-pointer” state when it advances past the last table entry.

Note that bad things (garbage results, core dumps) may happen if you call this function on an iterator which has been invalidated by a change in the table’s contents.

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	iterator handle is invalid
------------------------------------	----------------------------

### Examples

```
C          /* walk through all entries of a table */
          int ihandle;

          for ( ihandle = Util_TableItCreate(handle) ;
              Util_TableItQueryIsNonNull(ihandle) > 0 ;
              Util_TableItAdvance(ihandle) )
          {
              /* do something with the table entry */
          }

          Util_TableItDestroy(ihandle);
```

## Util\_TableItClone

---

Creates a new table iterator which is a “clone” (exact copy) of an existing table iterator

### Synopsis

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"
           int clone_ihandle = Util_TableItClone(int ihandle);
```

### Result

clone\_ihandle ( $\geq 0$ )  
A handle to the clone table iterator

### Parameters

ihandle            handle to the table iterator to be cloned

### Discussion

This function creates a new iterator which points to the same place in the same table as the original iterator. If the original iterator is in the “null-pointer” state, then the clone is also in this state.

Note that bad things (garbage results, core dumps) may happen if you call this function on an iterator which has been invalidated by a change in the table’s contents.

### See Also

Util_TableClone() <a href="#">[B26]</a>	create a new table which is a “clone” (exact copy) of an existing table
Util_TableItCreate() <a href="#">[B48]</a>	create a table iterator
Util_TableItDestroy() <a href="#">[B49]</a>	destroy a table iterator

### Errors

UTIL_ERROR_BAD_HANDLE	iterator handle to be cloned, is invalid
UTIL_ERROR_NO_MEMORY	unable to allocate memory

### Examples

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"

           /*
            * Apart from efficiency and slight differences in error return codes,
            * Util_TableItClone() could be simulated by the following code.
            */
           int Util_TableItClone(int ihandle)
           {
               int status;

               /* to what table does the to-be-cloned iterator point? */
```

```
const int handle = Util_TableQueryTableHandle(ihandle);
if (handle < 0)
    return handle;                /* error in querying table handle */

/* create the to-be-cloned iterator */
/* (pointing into the same table as the original iterator) */
{
    const int clone_ihandle = Util_TableItCreate(handle);
    if (clone_ihandle < 0)
        return clone_ihandle;    /* error in creating clone iterator */

    /* how long is the key to which the to-be-cloned iterator points? */
    {
        const int key_length = Util_TableItQueryKeyValueInfo(ihandle,
                                                                0, NULL,
                                                                NULL, NULL);

        if (key_length == UTIL_TABLE_ITERATOR_IS_NULL)
        {
            /* to-be-cloned iterator is in "null-pointer" state */
            Util_TableItSetToNull(clone_ihandle);
            return clone_ihandle;    /* normal return */
        }
        if (key_length < 0)
            return key_length;    /* error in querying to-be-cloned iterator */

        /* to what key does the to-be-cloned iterator point? */
        {
            const int key_buffer_length = key_length + 1;
            char *const key_buffer = (char *) malloc(key_buffer_length);
            if (key_buffer == NULL)
                return UTIL_ERROR_NO_MEMORY;
            status = Util_TableItQueryKeyValueInfo(ihandle,
                                                    key_buffer_length, key_buffer);

            if (status < 0)
                return status;    /* error in querying to-be-cloned iterator */

            /* set the clone iterator to point to the same key as the original */
            status = Util_TableItSetToKey(clone_ihandle, key_buffer);
            free(key_buffer);
            return clone_ihandle;    /* normal return */
        }
    }
}
```

## Util\_TableItCreate

---

Create a new table iterator

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int ihandle = Util_TableItCreate(int handle);
```

### Result

ihandle ( $\geq 0$ )     handle to the table iterator

### Parameters

handle ( $\geq 0$ )     handle to the table over which the iterator should iterate

### Discussion

This function creates a new table iterator. The iterator initially points at the starting table entry.

### See Also

Util\_TableItDestroy() [\[B49\]](#)     destroy a table iterator

### Errors

UTIL_ERROR_BAD_HANDLE	table handle is invalid
UTIL_ERROR_NO_MEMORY	unable to allocate memory

### Examples

```
C          /* walk through all entries of a table */
          int ihandle;

          for ( ihandle = Util_TableItCreate(handle) ;
              Util_TableItQueryIsNonNull(ihandle) > 0 ;
              Util_TableItAdvance(ihandle) )
          {
              /* do something with the table entry */
          }

          Util_TableItDestroy(ihandle);
```

## Util\_TableItDestroy

---

Destroy a table iterator

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int status = Util_TableItDestroy(int ihandle);
```

### Result

0                      ok

### Parameters

ihandle ( $\geq 0$ )      handle to the table iterator

### Discussion

### See Also

Util\_TableItCreate() [[B48](#)]      create a table iterator

### Errors

UTIL_ERROR_BAD_HANDLE	iterator handle is invalid
UTIL_ERROR_NO_MEMORY	unable to allocate memory

### Examples

```
C          /* walk through all entries of a table */
          int ihandle;

          for ( ihandle = Util_TableItCreate(handle) ;
                Util_TableItQueryIsNonNull(ihandle) > 0 ;
                Util_TableItAdvance(ihandle) )
          {
            /* do something with the table entry */
          }

          Util_TableItDestroy(ihandle);
```

**Util\_TableItQueryIsNonNull**

---

Query whether a table iterator is *not* in the “null-pointer” state

**Synopsis**

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int status = Util_TableItQueryIsNonNull(int ihandle);
```

**Result**

1	iterator is <i>not</i> in the “null-pointer” state, i.e. iterator points to some table entry
0	iterator is in the “null-pointer” state

**Parameters**

`ihandle` ( $\geq 0$ )    handle to the table iterator

**Discussion**

If no errors occur, `Util_TableItQueryIsNonNull(ihandle)` is the same as `1 - Util_TableItQueryIsNull(ihandle)`.

Note that bad things (garbage results, core dumps) may happen if you call this function on an iterator which has been invalidated by a change in the table’s contents.

**See Also**

`Util_TableItQueryIsNull()` [B51]    query whether a table iterator is in the “null-pointer” state

**Errors**

`UTIL_ERROR_BAD_HANDLE`            iterator handle is invalid

**Examples**

```
C          /* walk through all entries of a table */
          int ihandle;

          for ( ihandle = Util_TableItCreate(handle) ;
              Util_TableItQueryIsNonNull(ihandle) > 0 ;
              Util_TableItAdvance(ihandle) )
          {
              /* do something with the table entry */
          }

          Util_TableItDestroy(ihandle);
```

## Util\_TableItQueryIsNull

---

Query whether a table iterator is in the “null-pointer” state

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int status = Util_TableItQueryIsNull(int ihandle);
```

### Result

1            iterator is in the “null-pointer” state  
0            iterator is *not* in the “null-pointer” state, i.e. iterator points to some table entry

### Parameters

ihandle ( $\geq 0$ )    handle to the table iterator

### Discussion

If no errors occur, `Util_TableItQueryIsNull(ihandle)` is the same as `1 - Util_TableItQueryIsNonNull(ihandle)`.

Note that bad things (garbage results, core dumps) may happen if you call this function on an iterator which has been invalidated by a change in the table’s contents.

### See Also

`Util_TableItQueryIsNonNull()` [\[B50\]](#)  
query whether a table iterator is *not* in the “null-pointer” state,  
i.e. whether the iterator points to some table entry

### Errors

UTIL\_ERROR\_BAD\_HANDLE            iterator handle is invalid

### Examples

```
C          /* variant code to walk through all entries of a table */
          int ihandle;

          for ( ihandle = Util_TableItCreate(handle) ;
               Util_TableItQueryIsNull(ihandle) == 0 ;
               Util_TableItAdvance(ihandle) )
          {
              /* do something with the table entry */
          }

          Util_TableItDestroy(ihandle);
```



**Util\_TableItQueryKeyValueInfo**

---

Query the key and the type and number of elements of the value corresponding to that key, of the table entry to which an iterator points

**Synopsis**

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"
      int key_length =
          Util_TableItQueryKeyValueInfo(int ihandle,
                                         int key_buffer_length, char key_buffer[],
                                         CCTK_INT *type_code, CCTK_INT *N_elements)
```

**Result**

**key\_length**      The string length of the key (this has C `strlen` semantics, i.e. it does *not* include a terminating null character)

**Parameters**

**ihandle** ( $\geq 0$ )    handle to the table iterator

**key\_buffer\_length**      the length (`sizeof`) of `key_buffer[]` (must be  $\geq 1$  if `key_buffer != NULL`)

**key\_buffer**      a pointer to a buffer into which this function should store (at most `key_buffer_length-1` characters of) the key, terminated by a null character as usual for C strings, or NULL pointer to skip storing this

**type\_code**      a pointer to where this function should store the value's type code (one of the `CCTK_VARIABLE_*` constants from "`cctk_Constants.h`"), or a NULL pointer to skip storing this.

**N\_elements**      a pointer to where this function should store the number of array elements in the value, or a NULL pointer to skip storing this.

**Discussion**

The usual use of an iterator is to iterate through all the entries of a table, calling this function on each entry, then taking further action based on the results.

Note that bad things (garbage results, core dumps) may happen if you call this function on an iterator which has been invalidated by a change in the table's contents.

If the error code `UTIL_ERROR_TABLE_STRING_TRUNCATED` is returned, then the first `key_buffer_length-1` characters of the key are returned in the user's key buffer (assuming `key_buffer` is non-NULL), followed by a null character to properly terminate the string in the buffer. If any other error code is returned, the user's key buffer (pointed to by `key_buffer` if this is non-NULL) is unchanged.

**See Also**

`Util_TableQueryValueInfo()` [\[B61\]](#)

query key present/absent in table, and optionally type and/or number of elements, but using the key instead of an iterator

**Errors**

UTIL\_ERROR\_BAD\_HANDLE                    handle is invalid  
UTIL\_ERROR\_TABLE\_ITERATOR\_IS\_NULL  
   iterator is in "null-pointer" state  
UTIL\_ERROR\_TABLE\_STRING\_TRUNCATED  
   key\_buffer != NULL and key was truncated to fit in key\_buffer

### Examples

```
C                    /* print out all entries in a table */
                     /* return 0 for ok, type code for any types we can't handle, */
                     /*                    -ve for other errors */
                     #include <stdio.h>
                     #include <stdlib.h>
                     #include "util_ErrorCodes.h"
                     #include "util_Table.h"
                     #include "cctk.h"

                     int print_table(int handle)
                     {
                     int max_key_length, N_key_buffer, ihandle;
                     char *key_buffer;

                     max_key_length = Util_TableQueryMaxKeyLength(handle);
                     if (max_key_length < 0)
                         return max_key_length;

                     N_key_buffer = max_key_length + 1;
                     key_buffer = (char *) malloc(N_key_buffer);
                     if (key_buffer == NULL)
                         return UTIL_ERROR_NO_MEMORY;

                     for ( ihandle = Util_TableItCreate(handle) ;
                           Util_TableItQueryIsNonNull(ihandle) > 0 ;
                           Util_TableItAdvance(ihandle) )
                     {
                     CCTK_INT type_code, N_elements;
                     CCTK_INT value_int;
                     CCTK_REAL value_real;

                     Util_TableItQueryKeyValueInfo(ihandle,
                                                     N_key_buffer, key_buffer,
                                                     &type_code, &N_elements);
                     printf("key = \"%s\"\n", key_buffer);

                     switch (type_code)
                     {
                     case CCTK_VARIABLE_INT:
                         Util_TableGetInt(handle, &value_int, key_buffer);
                         printf("value[int] = %d\n", (int)value_int);
                         break;
                     case CCTK_VARIABLE_REAL:
                         Util_TableGetReal(handle, &value_real, key_buffer);
```

```
        printf("value[real] = %g\n", (double)value_real);
        break;
    default:
        /* we don't know how to handle this type */
        Util_TableItDestroy(ihandle);
        free(key_buffer);
        return type_code;
    }
}

Util_TableItDestroy(ihandle);
free(key_buffer);
return 0;
}
```

**Util\_TableItQueryTableHandle**

---

Query what table a table iterator iterates over

**Synopsis**

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int handle = Util_TableItQueryTableHandle(int ihandle);
```

**Result**

handle ( $\geq 0$ )      handle to the table over which the iterator iterates

**Parameters**

ihandle ( $\geq 0$ )      handle to the table iterator

**Discussion**

Note that it is always ok to call this function, regardless of whether or not the iterator is in the “null-pointer” state.

It’s also ok to call this function even when the iterator has been invalidated by a change in the table’s contents.

**See Also**

Util\_TableItCreate() [\[B48\]](#)      create an iterator (which iterates over a specified table)

**Errors**

UTIL\_ERROR\_BAD\_HANDLE      iterator handle is invalid

## Util\_TableItResetToStart

---

Reset a table iterator to point to the starting table entry

### Synopsis

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int status = Util_TableItResetToStart(int ihandle);
```

### Result

Results are the same as calling `Util_TableItQueryIsNonNull()` on the iterator after the reset:

1	iterator is <i>not</i> in the “null-pointer” state, i.e. iterator points to some table entry
0	iterator is in the “null-pointer” state (this happens if and only if the table is empty)

### Parameters

`ihandle` ( $\geq 0$ )    handle to the table iterator

### Discussion

Note that it is always ok to call this function, regardless of whether or not the iterator is in the “null-pointer” state.

It’s also ok to call this function even when the iterator has been invalidated by a change in the table’s contents.

### See Also

<code>Util_TableItSetToNull()</code>	<a href="#">[B58]</a>	set an iterator to the “null-pointer” state
<code>Util_TableItSetToKey()</code>	<a href="#">[B57]</a>	set an iterator to point to a specified table entry

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	iterator handle is invalid
------------------------------------	----------------------------

**Util\_TableItSetToKey**

---

Set a table iterator to point to a specified table entry

**Synopsis**

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"
           int status = Util_TableItSetToKey(int ihandle, const char *key);
```

**Result**

0                      ok

**Parameters**

ihandle ( $\geq 0$ )      handle to the table iterator

**Discussion**

This function has the same effect as calling `Util_TableItResetToStart()` followed by calling `Util_TableItAdvance()` zero or more times to make the iterator point to the desired table entry. However, this function will typically be (much) more efficient than that sequence.

Note that it is always ok to call this function, regardless of whether or not the iterator is in the “null-pointer” state.

It’s also ok to call this function even when the iterator has been invalidated by a change in the table’s contents.

**See Also**

`Util_TableItResetToStart()` [\[B56\]](#)                      reset an iterator to point to the starting table entry  
`Util_TableItSetToNull()` [\[B58\]](#)                      set a table iterator to the “null-pointer” state

**Errors**

<code>UTIL_ERROR_BAD_HANDLE</code>	iterator handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains ‘/’ character
<code>UTIL_ERROR_TABLE_NO_SUCH_KEY</code>	no such key in table

**Util\_TableItSetToNull**

---

Set a table iterator to the "null-pointer" state

**Synopsis**

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int handle = Util_TableItSetToNull(int ihandle);
```

**Result**

0                ok

**Parameters**

ihandle ( $\geq 0$ )    handle to the table iterator

**Discussion**

Note that it is always ok to call this function, regardless of whether or not the iterator is already in the "null-pointer" state.

It's also ok to call this function even when the iterator has been invalidated by a change in the table's contents.

**See Also**

Util\_TableItResetToStart() [[B56](#)]                reset an iterator to point to the starting table entry

Util\_TableItSetToKey() [[B57](#)]                set an iterator to point to a specified table entry

**Errors**

UTIL\_ERROR\_BAD\_HANDLE                iterator handle is invalid

## Util\_TableQueryFlags

---

Query a table's flags word

### Synopsis

```
C          #include "util_ErrorCodes.h"
           #include "util_Table.h"
           int flags = Util_TableQueryFlags(int handle);

Fortran    call Util_TableQueryFlags(flags, handle)
           integer flags, handle
```

### Result

flags ( $\geq 0$ )      the flags word

### Parameters

handle ( $\geq 0$ )      handle to the table

### Discussion

See `Util_TableCreate()` for further discussion of the semantics of flag words.

### See Also

`Util_TableClone()` [B26]      create a new table which is a “clone” (exact copy) of an existing table

`Util_TableCreate()` [B28]      create a table (flags word specified explicitly)

`Util_TableCreateFromString()` [B30]      convenience routine to create a table (with certain default flags) and set key/value entries in it based on a parameter-file-like character string

### Errors

UTIL\_ERROR\_BAD\_HANDLE      handle is invalid

### Examples

```
C          #include <string.h>
           #include "util_ErrorCodes.h"
           #include "util_String.h"
           #include "util_Table.h"

           /* compare two strings, doing the comparison with the same */
           /* case-sensitive/insensitive semantics as a certain table uses */
           int compare_strings(int handle, const char *str1, const char *str2)
           {
               int flags = Util_TableQueryFlags(handle);
               return (flags & UTIL_TABLE_FLAGS_CASE_INSENSITIVE)
                   ? Util_StrCmpi(str1, str2)
                   :      strcmp (str1, str2);
           }
```



}

## Util\_TableQueryValueInfo

---

Query whether or not a specified key is in the table, and optionally the type and/or number of elements of the value corresponding to this key

### Synopsis

```

C          #include "util_ErrorCodes.h"
              #include "util_Table.h"
              int key_exists =
                  Util_TableQueryValueInfo(int handle,
                                          CCTK_INT *type_code, CCTK_INT *N_elements,
                                          const char *key);

Fortran    call Util_TableQueryValueInfo(key_exists,
              .                                handle,
              .                                type_code, N_elements,
              .                                key)
              integer      key_exists, handle
              CCTK_INT     type_code, N_elements
              character*(*) key
    
```

### Result

1            ok (key is in table)  
 0            ok (no such key in table)  
 (in this case nothing is stored in `*type_code` and `*N_elements`)

### Parameters

`handle` ( $\geq 0$ )    handle to the table

`type_code`        a pointer to where this function should store the value's type code (one of the `CCTK_VARIABLE_*` constants from "cctk\_Constants.h"), or a NULL pointer to skip storing this.

`N_elements`       a pointer to where this function should store the number of array elements in the value, or a NULL pointer to skip storing this.

`key`              a pointer to the key (a C-style null-terminated string)

### Discussion

Unlike all the other table query functions, this function returns 0 for “no such key in table”. The rationale for this design is that by passing NULL pointers for `type_code` and `N_elements`, this function is then a Boolean “is key in table?” predicate.

If any error code is returned, the user's buffers (pointed to by `type_code` and `N_elements` if these are non-NULL) are unchanged.

### See Also

`Util_TableItQueryKeyValueInfo()` [\[B52\]](#)  
 query key present/absent in table, and optionally type and/or number of elements, but using an iterator instead of the key

### Errors

UTIL\_ERROR\_BAD\_HANDLE            handle is invalid  
UTIL\_ERROR\_TABLE\_BAD\_KEY        key contains '/' character

### Examples

```
C      #include <stdio.h>
      #include <assert.h>
      #include "util_ErrorCodes.h"
      #include "util_Table.h"

      static const int data[] = {314, 159, 265};
      #define N_DATA  (sizeof(data) / sizeof(data[0]))

      CCTK_INT type_code, N_elements;

      /* see whether or not "key" is in table */
      if (Util_TableQueryValueInfo(handle, NULL, NULL, "key"))
      {
          /* key is in the table */
      }
      else {
          /* key is not in the table */
      }

      /* put "data" in table as 3-element integer array */
      Util_TableSetIntArray(handle, N_DATA, data, "data");

      /* query info about "data" value */
      assert( Util_TableQueryValueInfo(handle,
                                      &type_code, &N_elements,
                                      "data") == 1 );
      assert( type_code == CCTK_VARIABLE_INT );
      assert( N_elements == N_DATA );
```

**Util\_TableQueryMaxKeyLength**

---

Query the maximum key length in a table

**Synopsis**

```
C          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          int max_key_length = Util_TableQueryMaxKeyLength(int handle);

Fortran    call Util_TableQueryMaxKeyLength(max_key_length, handle)
          integer max_key_length, handle
```

**Result**

max\_key\_length ( $\geq 0$ )

The string length of the longest key in the table (this has C `strlen` semantics, i.e. it does *not* include a terminating null character)

**Parameters**

handle ( $\geq 0$ )      handle to the table

**Discussion**

This function is useful if you're going to iterate through a table, and you want to allocate a buffer which is guaranteed to be big enough to hold any key in the table.

**Errors**

UTIL\_ERROR\_BAD\_HANDLE      handle is invalid

**Examples**

```
C          #include <stdlib.h>
          #include "util_ErrorCodes.h"
          #include "util_Table.h"
          #include "cctk.h"

          int max_key_length = Util_TableQueryMaxKeyLength(handle);
          int N_buffer = max_key_length + 1;
          char *const buffer = (char *) malloc(N_buffer);
          if (buffer == NULL)
          {
              CCTK_WARN(CCTK_WARN_ABORT, "couldn't allocate memory for table key buffer!");
              abort();          /* CCTK_Abort() would be better */
                                /* if we have a cGH* available */
          }

          /* now buffer is guaranteed to be */
          /* big enough for any key in the table */
```

**Util\_TableQueryNKeys**

---

Query the number of key/value entries in a table

**Synopsis**

```
C          #include "util_ErrorCodes.h"
            #include "util_Table.h"
            int N_Keys = Util_TableQueryNKeys(int handle);

Fortran    call Util_TableQueryNKeys(N_Keys, handle)
            integer N_Keys, handle
```

**Result**

**N\_Keys** ( $\geq 0$ )      the number of key/value entries in the table

**Parameters**

**handle** ( $\geq 0$ )      handle to the table

**Errors**

**UTIL\_ERROR\_BAD\_HANDLE**              handle is invalid

**Util\_TableSet\***

---

This is a family of functions, one for each Cactus data type, to set the value associated with a specified key to be a specified single (1-element array) value

**Synopsis**

**C**                    `#include "util_ErrorCodes.h"`  
                      `#include "util_Table.h"`  
                      `int status = Util_TableSetXxx(int handle,`  
    `CCTK_XXX value,`  
    `const char *key);`  
  
                      where XXX is one of POINTER, FPOINTER<sup>3</sup>, CHAR, BYTE, INT, INT1, INT2, INT4, INT8, REAL, REAL4, REAL8, REAL16, COMPLEX, COMPLEX8, COMPLEX16, COMPLEX32 (not all of these may be supported on any given system)

**Fortran**            `call Util_TableSetXxx(status, handle, value, key)`  
                      `integer                status, handle`  
                      `CCTK_XXX            value`  
                      `character*(*)      key`  
  
                      where CCTK\_XXX may be any data type supported by C (above) except CCTK\_CHAR (Fortran doesn't have a separate "character" data type; use CCTK\_BYTE instead)

**Result**

1                    ok (key was already in table before this call, old value was replaced)  
                      (it doesn't matter what the old value's `type_code` and `N_elements` were, i.e. these do *not* have to match the new value)

0                    ok (key was not in table before this call)

**Parameters**

`handle` ( $\geq 0$ )      handle to the table

`value`                the value to be associated with the key

`key`                   a pointer to the key (a C-style null-terminated string)

**Discussion**

The key may be any C character string which does not contain a slash character (`'/'`).

The value is stored as a 1-element array.

This function invalidates any iterators for the table which are not in the "null-pointer" state.

**See Also**

`Util_TableCreateFromString()` [\[B30\]](#)                    convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string

`Util_TableGet*()`                    get a single (1-element array) value, or more generally the first array element of an array value

---

<sup>3</sup>For backwards compatability the function `Util_TableSetFnPointer()` is also provided as an alias for `Util_TableSetFPointer()`. This is deprecated as of Cactus 4.0 beta 13.

<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetGeneric()</code> [B38]	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> [B40]	get an array value with generic data type
<code>Util_TableGetString()</code> [B43]	get a character-string value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory

### Examples

```
C
#include <math.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"

CCTK_COMPLEX16 z;
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetInt(handle, 42, "the answer");
Util_TableSetReal(handle, 299792458.0, "speed of light");

z.Re = cos(0.37);      z.Im = sin(0.37);
Util_TableSetComplex16(handle, z, "my complex number");
```

## Util\_TableSet\*Array

---

This is a family of functions, one for each Cactus data type, to set the value associated with a specified key to be a copy of a specified array

### Synopsis

**C**

```
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TableSetXxxArray(int handle,
                                   int N_elements,
                                   const CCTK_XXX array[],
                                   const char *key);
```

where XXX is one of POINTER, FPOINTER<sup>4</sup>, CHAR, BYTE, INT, INT1, INT2, INT4, INT8, REAL, REAL4, REAL8, REAL16, COMPLEX, COMPLEX8, COMPLEX16, COMPLEX32 (not all of these may be supported on any given system)

**Fortran**

```
call Util_TableSetXxxArray(status, handle, N_elements, array, key)
integer      status, handle, N_elements
CCTK_XXX(*)  array
character(*)  key
```

where CCTK\_XXX may be any data type supported by C (above)

### Result

1            ok (key was already in table before this call, old value was replaced)  
              (it doesn't matter what the old value's `type_code` and `N_elements` were, i.e. these do *not* have to match the new value)

0            ok (key was not in table before this call)

### Parameters

`handle` ( $\geq 0$ )      handle to the table

`N_elements` ( $\geq 0$ )    the number of array elements in `array[]`

`array`                a pointer to the array (a copy of which) is to be associated with the key

`key`                  a pointer to the key (a C-style null-terminated string)

### Discussion

The key may be any C character string which does not contain a slash character (`'/'`). Note that empty (0-element) arrays are ok.

This function invalidates any iterators for the table which are not in the “null-pointer” state.

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 pointed-to data is still valid whenever that `CCTK_POINTER` is used.)

---

<sup>4</sup>For backwards compatability the function `Util_TableSetFnPointerArray()` is also provided as an alias for `Util_TableSetFPointerArray()`. This is deprecated as of Cactus 4.0 beta 13.



### See Also

<code>Util_TableCreateFromString()</code> [B30]	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetGeneric()</code> [B38]	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> [B40]	get an array value with generic data type
<code>Util_TableGetString()</code> [B43]	get a character-string value
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_BAD_INPUT</code>	<code>N_elements &lt; 0</code>
<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory

### Examples

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"

      #define N_DIGITS      5
      static const CCTK_INT pi_digits[N_DIGITS] = {3, 14, 159, 2653, 58979};
      int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

      Util_TableSetIntArray(handle, N_DIGITS, pi_digits, "digits of pi");
```

## Util\_TableSetFromString

---

Sets values in a table based on a string argument, which is interpreted with “parameter-file” semantics

### Synopsis

**C**                    `#include "util_ErrorCodes.h"`  
                       `#include "util_Table.h"`  
                       `int count = Util_TableSetFromString(int handle, const char *string);`

**Fortran**           `call Util_TableSetFromString(count, handle, string)`  
                       `integer            count, handle`  
                       `character*(*)    string`

### Result

`count` ( $\geq 0$ )        the number of key/value entries set

### Parameters

**string**             a pointer to a C-style null-terminated string specifying the table entries to be set (see below for details on the string contents)

### Discussion

The string should contain a sequence of zero or more **key=value** “assignments”, separated by whitespace. This function processes these assignments in left-to-right order, setting corresponding key/value entries in the table.

The present implementation only recognises integer, real, and character-string values (not complex), and integer and real arrays. To be precise, the string must match the following BNF:

```

string      →  assign*
assign      →  whitespace*
assign      →  whitespace* key whitespace* = whitespace* value delimiter
key         →  any string not containing '/' or '=' or whitespace
value       →  array | int_value | real_value | string_value
array       →  { int_value* } | { real_value }
int_value   →  anything recognized as a valid integer by strtod(3) in base
               10
real_value  →  anything not recognized as a valid integer by strtol(3) but
               recognized as valid by strdod(3)
string_value → a C-style string enclosed in "double quotes" (C-style
               character escape codes are allowed, i.e. bell ('\a'),
               backspace ('\b'), form-feed ('\f'), newline ('\n'),
               carriage-return ('\r'), tab ('\t'), vertical-tab ('\v'),
               backslash ('\'), single-quote ('\''), double-quote ('\\"),
               question-mark ('\?'))
string_value → a C-style string enclosed in 'single quotes' (C-style character
               escape codes are not allowed, i.e. every character within the
               string is interpreted literally)
delimiter   →  end-of-string | whitespace
whitespace  →  blank (' ') | tab ('\t') | newline ('\n') | carriage-
               return ('\r') | form-feed ('\f') | vertical-tab ('\v')
```

where \* denotes 0 or more repetitions and | denotes logical or.

Notice also that the keys allowed by this function are somewhat more restricted than those allowed by the other `Util_TableSet*()` functions, in that this function disallows keys containing '=' and/or whitespace.

If any error code is returned, assignments lexicographically earlier in the input string than where the error was detected will already have been made in the table. Unfortunately, there is no easy way to find out where the error was detected. :(

### See Also

<code>Util_TableCreateFromString()</code> <a href="#">[B30]</a>	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetGeneric()</code> <a href="#">[B38]</a>	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> <a href="#">[B40]</a>	get an array value with generic data type
<code>Util_TableGetString()</code> <a href="#">[B43]</a>	get a character-string value
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> <a href="#">[B72]</a>	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> <a href="#">[B74]</a>	set an array value with generic data type
<code>Util_TableSetString()</code> <a href="#">[B77]</a>	set a character-string value

### Errors

<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory
<code>UTIL_ERROR_BAD_KEY</code>	invalid input: key contains invalid character
<code>UTIL_ERROR_BAD_INPUT</code>	invalid input: can't parse input string
<code>UTIL_ERROR_NO_MIXED_TYPE_ARRAY</code>	invalid input: different array values have different datatypes
other error codes	this function may also return any error codes returned by <code>Util_TableSetString()</code> , <code>Util_TableSetInt()</code> , <code>Util_TableSetReal()</code> , <code>Util_TableSetIntArray()</code> , or <code>Util_TableSetRealArray()</code> .

### Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"

/* suppose we have a table referred to by handle */

/* then the call... */
int count = Util_TableSetFromString(handle, "n = 6\t"
                                     "dx = 4.0e-5\t"
                                     "pi = 3.1\t")
```

```
                                "s = 'my string'\t"
                                "array = { 1 2 3 }");

/* ... will return count=5 ... */

/* ... and is otherwise equivalent to the five calls ... */
CCTK_INT array[] = {1, 2, 3};

Util_TableSetInt(handle, 6, "n");
Util_TableSetReal(handle, 4.0e-5, "dx");
Util_TableSetReal(handle, 3.1, "pi");
Util_TableSetString(handle, "my string", "s");
Util_TableSetIntArray(handle, 3, array, "array");
```

## UtileTableSetGeneric

Set the value associated with a specified key to be a specified single (1-element array) value, whose data type is generic. That is, the value is specified by a `CCTK_VARIABLE.*` type code and a `void *` pointer.

## Synopsis

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"
      int status = Util_TableSetGeneric(int handle,
                                         int type_code, const void *value,
                                         const char *key);
```

```

Fortran      call Util_TableSetGeneric(status, handle, type_code, value, key)
                integer      status, handle, type_code
                CCTK_POINTER  value
                character(*)  key

```

## Result

1	ok (key was already in table before this call, old value was replaced) (it doesn't matter what the old value's <code>type_code</code> and <code>N_elements</code> were, i.e. these do <i>not</i> have to match the new value)
0	ok (key was not in table before this call)

## Parameters

<code>handle</code> ( $\geq 0$ )	handle to the table
<code>type_code</code>	the array elements' type code (one of the <code>CCTK_VARIABLE_*</code> constants from " <code>cctk_Constants.h</code> ")
<code>value_ptr</code>	a pointer to the value to be associated with the key
<code>key</code>	a pointer to the key (a C-style null-terminated string)

## Discussion

The key may be any C character string which does not contain a slash character ('/').

The value is stored as a 1-element array.

This function invalidates any iterators for the table which are not in the “null-pointer” state.

## See Also

`Util_TableCreateFromString()` [B30]  
convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string

Util_TableGet*()	get a single (1-element array) value, or more generally the first array element of an array value
------------------	---

Util_TableGet*Array()	get an array value
-----------------------	--------------------

Util_TableGetGeneric()	<a href="#">[B38]</a>	get a single (1-element array) value with generic data type
------------------------	-----------------------	---

`Util_TableGetGenericArray()` [B40] get an array value with generic data type

Util\_TableGetString() [B43] get a character-string value

<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetGenericArray()</code> [B74]	set an array value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

### Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_BAD_INPUT</code>	<code>type_code</code> is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains '/' character
<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory

### Examples

```
C
#include "util_Table.h"
#include "cctk_Constants.h"

const CCTK_INT i = 42;
const void *iptr = (void *) &i;
CCTK_INT icopy;

const CCTK_REAL x = 299792458.0;
const void *xptr = (void *) &x;
CCTK_REAL xcopy;

const int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetGeneric(handle, CCTK_VARIABLE_INT, iptr, "the answer");
Util_TableSetGeneric(handle, CCTK_VARIABLE_REAL, xptr, "speed of light");

/* gets icopy to 42 */
Util_TableGetInt(handle, &icopy, "the answer");

/* gets xcopy to 299792458.0 */
Util_TableGetReal(handle, &xcopy, "speed of light");
```

## Util\_TableSetGenericArray

---

Set the value associated with a specified key to be a copy of a specified array, whose data type is generic. That is, the array is specified by a `CCTK_VARIABLE_*` type code, a count of the number of array elements, and a `void *` pointer.

### Synopsis

<b>C</b>	<pre>#include "util_ErrorCodes.h" #include "util_Table.h" int status = Util_TableSetGenericArray(int handle,  int type_code,  int N_elements, const void *array,  const char *key);</pre>
<b>Fortran</b>	<pre>call Util_TableSetGenericArray(status, .                               handle, .                               type_code, .                               N_elements, array, .                               key) integer          status, handle, type_code, N_elements CCTK_POINTER(*)  array character*(*)    key</pre>

### Result

1	ok (key was already in table before this call, old value was replaced) (it doesn't matter what the old value's <code>type_code</code> and <code>N_elements</code> were, i.e. these do <i>not</i> have to match the new value)
0	ok (key was not in table before this call)

### Parameters

<code>handle</code> ( $\geq 0$ )	handle to the table
<code>type_code</code>	the array elements' type code (one of the <code>CCTK_VARIABLE_*</code> constants from " <code>cctk_Constants.h</code> ")
<code>N_elements</code> ( $\geq 0$ )	the number of array elements in <code>array[]</code>
<code>value_ptr</code>	a pointer to the value to be associated with the key
<code>key</code>	a pointer to the key (a C-style null-terminated string)

### Discussion

The key may be any C character string which does not contain a slash character (`'/'`).

The value is stored as a 1-element array.

This function invalidates any iterators for the table which are not in the "null-pointer" state.

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 pointed-to data is still valid whenever that `CCTK_POINTER` is used.)

## See Also

<code>Util_TableCreateFromString()</code> [B30]	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetGeneric()</code> [B38]	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> [B40]	get an array value with generic data type
<code>Util_TableGetString()</code> [B43]	get a character-string value
<code>Util_TableSet*()</code>	set a single (1-element array) value
<code>Util_TableSet*Array()</code>	set an array value
<code>Util_TableSetGeneric()</code> [B72]	set a single (1-element array) value with generic data type
<code>Util_TableSetFromString()</code> [B69]	convenience routine to set key/value entries in a table based on a parameter-file-like character string
<code>Util_TableSetString()</code> [B77]	set a character-string value

## Errors

<code>UTIL_ERROR_BAD_HANDLE</code>	handle is invalid
<code>UTIL_ERROR_BAD_INPUT</code>	<code>type_code</code> is invalid
<code>UTIL_ERROR_TABLE_BAD_KEY</code>	key contains <code>'/'</code> character
<code>UTIL_ERROR_NO_MEMORY</code>	unable to allocate memory

## Examples

```
C
#include "util_Table.h"
#include "cctk_Constants.h"

#define N_IARRAY    3
const CCTK_INT iarray[N_IARRAY] = {42, 69, 105};
const void *iarray_ptr = (void *) iarray;
CCTK_INT iarray2[N_IARRAY];

#define N_XARRAY    2
const CCTK_REAL xarray[N_XARRAY] = {6.67e-11, 299792458.0};
const void *xarray_ptr = (void *) xarray;
CCTK_REAL xarray2[N_XARRAY];

const int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetGenericArray(handle,
                           CCTK_VARIABLE_INT,
                           N_IARRAY, iarray_ptr,
                           "my integer array");
Util_TableSetGenericArray(handle,
                           CCTK_VARIABLE_REAL,
```



```
        N_XARRAY, xarray_ptr,  
        "my real array");  
  
/* gets iarray2[0] = 42, iarray2[1] = 69, iarray2[2] = 105 */  
Util_TableGetIntArray(handle, N_IARRAY, iarray2, "my integer array");  
  
/* gets xarray2[0] = 6.67e-11, xarray2[1] = 299792458.0 */  
Util_TableGetRealArray(handle, N_XARRAY, xarray2, "my real array");
```

## Util\_TableSetString

---

Sets the value associated with a specified key in a table, to be a copy of a specified C-style null-terminated character string

### Synopsis

```
C      #include "util_ErrorCodes.h"
      #include "util_Table.h"
      int status = Util_TableSetString(int handle,
                                      const char *string,
                                      const char *key);

Fortran call Util_TableSetString(status, handle, string, key)
integer      status, handle
character*(*) string, key
```

### Result

Results are the same as all the other `Util_TableSet*()` functions:

1	ok (key was already in table before this call, old value was replaced) (it doesn't matter what the old value's <code>type_code</code> and <code>N_elements</code> were, i.e. these do <i>not</i> have to match the new value)
0	ok (key was not in table before this call)

### Parameters

<code>handle</code> ( $\geq 0$ )	handle to the table
<code>string</code>	a pointer to the string (a C-style null-terminated string)
<code>key</code>	a pointer to the key (a C-style null-terminated string)

### Discussion

The key may be any C character string which does not contain a slash character (`'/'`). The string is stored as an array of `strlen(string)` `CCTK_CHARS`. It does *not* include a terminating null character.

This function is very similar to `Util_TableSetCharArray()`.

This function invalidates any iterators for the table which are not in the “null-pointer” state.

### See Also

<code>Util_TableCreateFromString()</code> <a href="#">[B30]</a>	convenience routine to create a table and set key/value entries in it based on a parameter-file-like character string
<code>Util_TableGet*()</code>	get a single (1-element array) value, or more generally the first array element of an array value
<code>Util_TableGet*Array()</code>	get an array value
<code>Util_TableGetGeneric()</code> <a href="#">[B38]</a>	get a single (1-element array) value with generic data type
<code>Util_TableGetGenericArray()</code> <a href="#">[B40]</a>	get an array value with generic data type

Util\_TableGetString() [B43]      get a character-string value  
 Util\_TableSetCharArray() [B67]    get an array-of-CCTK\_CHAR value  
 Util\_TableSet\*()                  set a single (1-element array) value  
 Util\_TableSet\*Array()            set an array value  
 Util\_TableSetGeneric() [B72]    set a single (1-element array) value with generic data type  
 Util\_TableSetGenericArray() [B74]    set an array value with generic data type  
 Util\_TableSetCharArray() [B67]    set an array-of-CCTK\_CHAR value

### Errors

UTIL\_ERROR\_BAD\_HANDLE            handle is invalid  
 UTIL\_ERROR\_TABLE\_BAD\_KEY        key contains '/' character  
 UTIL\_ERROR\_NO\_MEMORY            unable to allocate memory

### Examples

```
C
#include "util_ErrorCodes.h"
#include "util_Table.h"

static const CCTK_CHAR array[]
    = {'r', 'e', 'l', 'a', 't', 'i', 'v', 'i', 't', 'y'};
#define N_ARRAY (sizeof(array) / sizeof(array[0]))
int handle = Util_TableCreate(UTIL_TABLE_FLAGS_DEFAULT);

Util_TableSetString(handle, "relativity", "Einstein");

/* this produces the same table entry as the Util_TableSetString() */
Util_TableSetCharArray(handle, N_ARRAY, array, "Einstein");
```

# Part C

## Appendices

---

Note that these appendices appear (identically) in both the Cactus Users' Guide and the Cactus Reference Manual.

# Chapter C1

## Glossary

alias function	See <i>function aliasing</i> .
AMR	<i>Automatic Mesh Refinement</i>
analysis	
API	<i>Applications Programming Interface</i> , 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 <code>arrangements</code> directory.
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 <i>symmetry zone</i> , <i>ghost zone</i> .)
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. <sup>1</sup>
CCTK	<i>Cactus Computational Tool Kit</i> (The Cactus flesh and computational thorns).
CCL	The <i>Cactus Configuration Language</i> , this is the language that the thorn configuration files are written in. See Section C2.
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, CUCC, CXX, F77, and F90 configure options) and the compiler optimization settings (OPTIMISE/OPTIMIZE and *_OPTIMISE_FLAGS

---

<sup>1</sup><http://en.wikipedia.org/wiki/Cactus>

---

	configure options) are part of a configuration (these flags change what binary is produced), but the Cactus <code>SILENT</code> and <code>WARN</code> configure options aren't part of a configuration (they don't change what binary will be produced).
checkout	Get a copy of source code from CVS.
checkpoint	Save the entire state of a Cactus run to a file, so that the run can be restarted at a later time.
computational grid	<p>A discrete finite set of spatial points in <math>\mathbb{R}^n</math> (typically, <math>1 \leq n \leq 3</math>). 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.</p> <p>The grid consists of the physical domain and the boundary and symmetry points. See <i>grid functions</i> for the typical use of grid points.</p>
convergence	Important, but often neglected.
CST	The <i>Cactus Specification Tool</i> , which is the set of Perl scripts which parse the thorns' <code>.ccl</code> files, and generates the code that binds the thorn source files with the flesh.
CVS	The <i>Concurrent Versions System</i> is the favoured code distribution system for Cactus. See Section <a href="#">C7</a> .
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.
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 CVS repository.
friend	Interfaces that are <i>friends</i> , share their collective set of protected grid variables. See Section <a href="#">C2.2</a> .
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 <i>alias function</i> . <a href="#">C2.2.3</a> .
GA	Shorthand for a <i>grid array</i> .
GF	Shorthand for a <i>grid function</i> .
gmake	GNU version of the <code>make</code> 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.

grid	Short for <i>computational grid</i> .
grid array	A <i>grid variable</i> whose global size need not be that of the computational grid; instead, the size is declared explicitly in an <code>interface.ccl</code> file.
grid function	A <i>grid variable</i> whose global size is the size of the computational grid. (See also <i>local array</i> .) From another perspective, <i>grid functions</i> are functions (of any of the Cactus data types defined on the domain of grid points. Typically, grid functions are used to discretely approximate functions defined on the domain $\mathbb{R}^n$ , with <i>finite differencing</i> used to approximate partial derivatives.
grid hierarchy	A <i>computational grid</i> , and the <i>grid variables</i> associated with it.
grid point	A spatial point in the <i>computational grid</i> .
grid scalar	A <i>grid variable</i> 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. <i>grid scalar</i> , <i>grid function</i> , and <i>grid array</i> are all examples of <i>grid variables</i> . <a href="#">C2.2.4</a>
GNATS	The GNU program we use for reporting and tracking bugs, comments and suggestions.
GNU	<i>GNU's Not Unix</i> : a freely-distributable code project. See <a href="http://www.gnu.org/">http://www.gnu.org/</a> .
GV	Shorthand for <i>grid variable</i> .
handle	A signed integer value $\geq 0$ passed by many Cactus routines and used to represent a dynamic data or code object.
HDF5	<i>Hierarchical Data Format</i> 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 <a href="http://hdf.ncsa.uiuc.edu/HDF5/">http://hdf.ncsa.uiuc.edu/HDF5/</a> .
implementation	Defines the interface that a thorn presents to the rest of a Cactus program.
inherit	A thorn that <i>inherits</i> from another implementation can access all the other implementation's public variables. See Section <a href="#">C2.2</a> .
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 <i>local interpolation</i> .)
local array	An array that is declared in thorn code, but not declared in the thorn's <code>interface.ccl</code> , as opposed to a <i>grid array</i> .
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 <i>interpolation</i> .)
Makefile	The default input file for <code>make</code> (or <code>gmake</code> ). 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	<i>Message Passing Interface</i> , an API and software library for sending messages between processors in a multiprocessor system.
multi-patch	
mutual recursion	See <i>recursion</i> , <i>mutual</i> .
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 <code>char</code> rather than <code>int</code> .
null pointer, NULL pointer	<p>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. <code>int* ptr = 0;</code><sup>2</sup></p> <p>Many programmers prefer to use the predefined macro <code>NULL</code> (defined in <code>&lt;stdlib.h&gt;</code>, <code>&lt;stdio.h&gt;</code>, and possibly other system header files) to create null pointers, e.g. <code>int* ptr = NULL;</code>, to emphasize to human readers that this is a null <i>pointer</i> rather than “just” the integer zero.</p> <p>Note that it is explicitly <i>not</i> 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, <i>not</i> represented this way.</p> <p>For further information, see the section “Null pointers” in the (excellent) <code>comp.lang.c</code> FAQ, available online at <a href="http://www.eskimo.com/~scs/C-faq/top.html">http://www.eskimo.com/~scs/C-faq/top.html</a>.</p>
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 <i>domain decomposition</i> .
parameter	A variable that controls the run time behaviour of the Cactus executable. Parameters have default values which can be set in a <i>parameter file</i> . The flesh has parameters; thorn parameters are made available to the rest of Cactus by describing them in the thorn’s <code>param.cc1</code> file (See Appendix C2.3).
parameter file	(Also called <i>par file</i> .) A text file used as the input of a Cactus program, specifying initial values of thorn parameters.
processor topology	
PUGH	The default driver thorn for Cactus which uses MPI.
PVM	<i>Parallel Virtual Machine</i> , provides interprocessor communication.
recursion, mutual	See <i>mutual recursion</i> .
reduction	Given a set of grid variables on a computational grid, <i>reduction</i> is the process of producing values for the variables on a proper subset of points from the grid.

---

<sup>2</sup>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! */
```

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 <code>schedule.ccl</code> file to be executed in a certain schedule bin, before or after another function or group executes, and so forth. See section <a href="#">C2.4</a> .
schedule bin	One of a set of special timebins pre-defined by Cactus. See Section <a href="#">C4</a> for a list.
schedule group	A timebin defined by a thorn, in its <code>schedule.ccl</code> file (see Appendix <a href="#">C2.4</a> ). Each schedule group must be defined to occur in a Cactus schedule bin or another schedule group.
shares	An implementation may <i>share</i> 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 <a href="#">C2.3</a> .
staggering	
steerable parameter	A parameter which can be changed at any time after the program has been initialised.
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 <i>boundary zone</i> , <i>ghost zone</i> .)
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.
TAGS	See Section <a href="#">C8</a> .
target	A <i>make target</i> is the name of a set of rules for <code>make</code> (or <code>gmake</code> ). When the target is included in the command line for <code>make</code> , the rules are executed, usually to build some software.
test suite	
thorn	A collection of subroutines defining a Cactus interface.
ThornList	A file used by the Cactus CST to determine which thorns to compile into a Cactus executable Can also be used to determine which thorns to check out from CVS.A ThornList for each Cactus configuration lies in the configuration subdirectory of the Cactus <code>configs</code> directory.
time bin	A time interval in the duration of a Cactus run wherein the flesh runs specified routines. See <i>scheduler</i> , <i>schedule bin</i> .
time level	
timer	A Cactus API for reporting time.
trigger	
unigrid	
WMPI	<i>Win32 Message Passing Interface</i> .
wrapper	

## Chapter C2

# Configuration File Syntax

### C2.1 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.

### C2.2 `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.

#### C2.2.1 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*).

## C2.2.2 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*.

## C2.2.3 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 be 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.

## C2.2.4 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>] [STAGGER=<stagger-specification>]
[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**.
- **STAGGER** defines position of grid-points of a **GF** with respect to the underlying grid. It consists of a string made up of a combination **DIM** of the letters **M**, **C**, **P**, depending on whether the layout in that direction is on the Minus face, Centre, or Plus face of the cell in that dimension.
- **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 delimited 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.
- 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.

## C2.3 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.

### C2.3.1 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:

<code>global</code>	all thorns have access to global parameters
<code>restricted</code>	other thorns can have access to these parameters, if they specifically request it in their own param.ccl
<code>private</code>	only your thorn has access to private parameters
<code>shares</code>	in this case, an <code>implementation</code> name must follow the colon. It declares that all the parameters in the following scoping block are restricted variables from the specified <code>implementation</code> . (Note: only one implementation can be specified on this line.)

### C2.3.2 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 <i>parameter values</i> 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 CCTL.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.

## C2.4 schedule.ccl

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.

### C2.4.1 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 of 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.

The behaviour of an assignment statement is independent of its position in the schedule file (so long as it is outside a schedule block).

## C2.4.2 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>]
  [STORAGE:      <group>[timelevels],<group>[timelevels]...]
  [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 C4. 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.
LANG	The code language for the function (either C or FORTRAN). No language should be specified for a schedule group.
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 <code>interface.ccl</code> file. If the maximum is 1 (the default) this number may be omitted. Alternatively <code>timelevels</code> 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 <code>thorn::parameter</code> are not allowed.
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 <code>cctk_levfac</code> 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”.

### C2.4.3 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.

## C2.5 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>
  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 C2.5.1. 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.

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>*

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.

```
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".

### C2.5.1 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 C3

## Utility Routines

### C3.1 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.

### C3.2 Key/Value Tables

#### C3.2.1 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).

#### C3.2.2 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.)<sup>1</sup>

---

<sup>1</sup>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.<sup>2</sup>

Values are 1-dimensional arrays of any of the usual Cactus data types described in Section ???. 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.

### C3.2.3 A Simple Example

Here’s a simple example (in C)<sup>3</sup> 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");

...
```

<sup>2</sup>Think of hierarchical tables for storing tree-like data structures.

<sup>3</sup>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.<sup>4</sup> 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 */

```

### C3.2.4 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).<sup>5</sup>

For example (continuing the previous example):

<sup>4</sup>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.

<sup>5</sup>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.<sup>6</sup>

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 */
```

### C3.2.5 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" */
```

<sup>6</sup>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.

### C3.2.6 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 C3.2.3:

```
#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()`<sup>7</sup>, 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).

<sup>7</sup>C automatically concatenates adjacent character string constants separated only by whitespace.

### C3.2.7 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()`).<sup>8</sup>

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.

### C3.2.8 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.

### C3.2.9 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.)

---

<sup>8</sup>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 C4

## 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 <b>BEFORE</b> , <b>AFTER</b> , etc., functionality of the <code>schedule.cc1</code> 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 <b>BEFORE</b> , <b>AFTER</b> , etc., functionality of the <code>schedule.cc1</code> file should allow all functions to be scheduled in the main CCTK_EVOL timebin.
CCTK_ANALYSIS	The <b>ANALYSIS</b> 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 C5

## Flesh Parameters

The flesh parameters are defined in the file `src/param.ccl`.

### C5.1 Private Parameters

Here, the default value is shown in square brackets, while curly braces show allowed parameter values.

<code>allow_mixeddim_gfs</code>	Allow use of GFs from different dimensions [no]
<code>cctk_brief_output</code>	Give only brief output [no]
<code>cctk_full_warnings</code>	Give detailed information for each warning statement [yes]
<code>cctk_run_title</code>	Description of this simulation [""]
<code>cctk_show_banners</code>	Show any registered banners for the different thorns [yes]
<code>cctk_show_schedule</code>	Print the scheduling tree to standard output [yes]
<code>cctk_strong_param_check</code>	Die on parameter errors in CCTK_PARAMCHECK [yes]
<code>cctk_timer_output</code>	Give timing information [off] {off, full}
<code>manual_cache_setup</code>	Set the cache size manually [no]
<code>manual_cache_size</code>	The size to set the cache to if not done automatically (bytes) [0]
<code>manual_cacheline_bytes</code>	The size of a cacheline if not set automatically (bytes) [0]
<code>recovery_mode</code>	How to behave when recovering from a checkpoint [strict] {strict, relaxed}
<code>highlight_warning_messages</code>	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" }

## C5.2 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]  
`terminate` Condition on which to terminate evolution loop [iteration] {never, iteration,  
time, runtime, any, all}  
`terminate_next` Terminate on next iteration ? [no]

# Chapter C6

## Using GNATS

GNATS is a freely redistributable set of tools for tracking bug reports. It allows users to categorize their problem report and submit them to the GNATS. The bug tracker will assign appropriate maintainers to analyze and solve the problem. We are currently supporting a web-based interface at <http://www.cactuscode.org>, which lets you interactively file a bug report. Here, we briefly describe the main categories when creating a Cactus problem report.

<b>Reporters email</b>	Your email address, so we can get in contact with you.
<b>Category</b>	There is currently a category for each of the Cactus thorns and arrangements, also a category for the old Cactus3.x and some general subjects like Web, etc. Select whatever category fits best.
<b>Synopsis</b>	A brief and informative subject line.
<b>Confidential</b>	Unused, all PRs are public.
<b>Severity</b>	Pick one of three levels.
<b>Class</b>	In the selected category, specify what kind of problem you are dealing with.
<b>submitter ID</b>	Unused
<b>Originator</b>	Your name. Anonymous is OK, but you real name would be best.
<b>Release</b>	The Cactus release you are using. You can find this out, for example, from an executable by typing <code>cactus.&lt;config&gt; -v</code> .
<b>Environment</b>	Very important: specify the environment, e.g. <code>uname -a</code> , MPI/non-MPI, etc.
<b>Description</b>	Describe your problem precisely, if you get a core dump, include the stack trace, give the minimal number of thorns, this problems occurs with.
<b>How-To-Repeat</b>	Tell us how to repeat the problem if it is software related.
<b>Fix</b>	If you can provide a fix, let us know.

We also provide the customized `send-pr` and `send-pr.el` programs at our website. These commands are compiled to submit Cactus problem reports in your shell and from within Emacs, respectively.

# Chapter C7

## Using CVS

CVS is a version control system, which allows you to keep old versions of files (usually source code), log of when, and why changes occurred, and who made them, etc. Unlike the simpler systems, CVS does not just operate on one file at a time or one directory at a time, but operates on hierarchical collections of directories consisting of version controlled files. CVS helps to manage releases and to control the concurrent editing of source files among multiple authors. CVS can be obtained from <http://www.cyclic.com>.

A CVS *repository* located on a *server* may consist of an arbitrary number of *modules*, which can be checked out (that is downloaded) independently. The Cactus flesh and the Cactus arrangements are organized as modules, their CVS *server* is `cvs.cactuscode.org`.

### C7.1 Essential CVS Commands

**cvs login** Logs into the repository. You will be prompted for a *password*. This CVS command leaves a file `.cvspass` in your home directory. There is no need to login every time you issue a CVS command, as long as this file exists. For a Cactus checkout, you have to log into the CVS server, using the CVS option `-d` to specify `CVSROOT`:

```
cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus login
```

*Note:* that there is no “logout” command: if you log in with administrative rights from an arbitrary account, you should be aware that the password file enables subsequent administrative logins from that account. *Delete the file if necessary.*

**cvs checkout modules ...**

This command creates your private copy of the source for *modules*. You can work with this copy without interfering with others’ work. At least one subdirectory level is always created: it does not write all files into your current directory but creates a directory. For Cactus, you need to either include the `-d` options to specify the `CVSROOT` directory and the CVS server, or specify them with an environment variable (see below). Once you have checked out the repository there is no need to include the `-d` option and its rather lengthy argument: the necessary information is contained in the local `CVS/` directories.

**cvs update** Execute this command from *within* your private source directory when you wish to update your copies of source files from changes that other developers have made to

the source in the repository. Merges are performed automatically when possible, a warning is issued if manual resolution is required for conflicting changes. If your local copy is several versions behind the actual repository copy, CVS will *refetch* the whole file instead of applying multiple patches.

**cv**s **add** **file** Use this command to enroll new files in CVS records of your working directory. The files will be added to the repository the next time you run ‘**cv**s **commit**’.

**cv**s **commit** **file** Use this command to add your local changes to the source to the repository and, thereby, making it publically available to checkouts and updates by other users. You cannot commit a newly created file unless you have *added* it.

**cv**s **diff** **file** Show differences between a file in your working directory and a file in the source repository, or between two revisions in source repository. (Does not change either repository or working directory.) For example, to see the difference between versions 1.8 and 1.9 of a file `foobar.c`:

```
cv
```

s diff -r 1.8 1.9 foobar.c

**cv**s **remove** **file** Remove files from the source repository, pending a **cv**s **commit** on the same files.

**cv**s **status** [**file**] This command returns the current status of your local copy relative to the repository: e.g. it indicates local modifications and possible updates.

**cv**s **import** **repository** **tag1** **tag2**

Import adds an entire source distribution (starting from the directory you issue the command in) to the repository directory. Use this command to add new arrangements to the Cactus 4.0 repository. The **repository** argument is a directory name (or a path to a directory) and the CVS root directory for repositories; to obtain this directory on the CVS server, send a request to `cactus@cactuscode.org`. **tag1** and **tag2** are two tags (vendor and release tags) that have to be supplied. For example, to add **MyThorn** to the **MyArrangement** arrangement, which may or may not already exist on the CVS repository

```
cv
```

s -d :pserver:<name>@cvs.cactuscode.org:/cactus import MyArrangement/MyThorn
start v1

After you import a thorn, you should check it out from the repository straight away, and only edit this version.

## C7.2 CVS Options

The CVS command line can include the following:

**cv**s **options** which apply to the overall CVS program

**a cv**s **command** which defines a particular task carried out by CVS

**command options** to specify certain working modes for the CVS command.

**command arguments**  
to specify which file to act on.

The options must be put *relative* to the CVS *command* as the same option name can mean different things: CVS *options* go to the *left* of the CVS command, *command options* go to the *right* of the CVS command. Here is a list of essential CVS options:

- d *cvs\_root\_directory*** Use *cvs\_root\_directory* as the root directory pathname of the master source repository. Overrides the setting of the CVSR00T environment variable. This value should be specified as an absolute pathname. In the Cactus checkout procedure, you specify the Cactus CVS server:  
     **-d :pserver:cvs\_anon@cvs.cactuscode.org:/cactus/**
- z *compression-level*** When transferring files across the network use **gzip** with compression level *compression-level* to compress and decompress data as it is transferred. Requires the presence of the GNU **gzip** program in the current search path at both ends of the link.
- n** Do not change any file. Attempt to execute the CVS *command* but only to issue reports. Does not remove, update, etc., any files. Very effective for testing.
- v** Displays version information of the installed CVS.
- H *cvs-command*** Displays usage information about the specified CVS command. Without *cvs-command*, a list of all available commands is returned.

Here is a list of essential command options with the commands they are used with. They go after the CVS command. For a more complete list of all options, please refer to the manual page.

- P** Prune (remove) directories that are empty after being updated, on **checkout**, or **update**. Normally, an empty directory (one that is void of revision controlled files) is left alone. Specifying **-P** will cause these directories to be silently removed from the sources you have checked out. This does not remove the directory from the repository, only from your checked out copy.
- m "*Text*"** Specify a logging message explaining changes, etc., on **commit**, **import**. If you do not specify a message, your default editor is invoked to allow you to enter one.
- d** Use this option with the **update** command to create any directories if they are missing from your local copy. This is normally the case if another user has added files or directories to the repository. By default, the **update** command only acts on files in your local copy. Note that omitting this option is a frequent cause of files missing during compilation. (You can change this default behavior of CVS by putting a **.cvsrc** in your home directory with the contents "**update -d**".)

## C7.3 CVS Examples

We list some sample CVS commands to treat the most typical Cactus 4.0 CVS situations.

### Logging into the server

```
cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus login
```

You will be asked for the password for user *cvs\_anon*, which is **anon**.

**Checking out the code**

`cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus checkout Cactus`  
 check out a CVS module named “Cactus”, in this case it checks out the Cactus Computational Toolkit. A directory `./Cactus` is created if it doesn’t already exist. If you perform a checkout on an already existing and locally modified copy of the module, CVS will try to merge the files with your local copy.

**Updating a file or directory**

Assuming that you have a file `./foobar` in your checked out copy, you may perform a

`cvs status ./foobar`

to inform yourself about the necessary updates, etc. To update the file issue

`cvs update ./foobar`

If that was file was locally modified, CVS will try to merge the changes. Manual merging might be necessary and will be indicated by a CVS warning.

**Updating a directory**

To recursively update the current directory and all subdirectories, type

`cvs update .`

To update a directory `./mysources`, type

`cvs update ./path/to/mysources`

**Committing a changed file**

To commit changes you have applied to your local copy, your file must be in sync with the repository: your changes must be done to the latest version, otherwise CVS will instruct you to perform an update first. To commit changes made to a file `./foobar`, type

`cvs commit -m "Reason for the change" ./foobar`

You may specify several files to commit.

**Adding and committing a new file**

Adding a new file to the repository is a two fold procedure you first schedule the file for addition, then you commit it:

`cvs add ./newfoo`

`cvs commit -m "new few message" ./newfoo`

**Creating a new thorn**

To add a new *module* (e.g. an arrangement) to a Cactus repository, we first have to create a directory for you with the right permissions. Please contact [cactus@cactuscode.org](mailto:cactus@cactuscode.org) providing the name of the requested module, and who should be able to commit changes to the module.

To add the new module, change directory so that you are in the first directory that you want to commit to the repository. (For example, if you want to commit a new arrangement called `MyArrange` then change directory to `MyArrange`). Then type

`cvs -d :pserver:your_login@cvs.cactuscode.org:<repository name> import module_name start V1`

(where `start` and `V1` are the vendor and release tags, which you could change to something different).

To add a new *directory* `<new dir>` to an existing module (that you have write permissions for), either add the directory using

`cvs add <new dir>`

and then recursing down adding all the new files and directories contained inside, or import the directory by changing directory to sit inside it, and then using

`cvs -d :pserver:your_login@cvs.cactuscode.org:<repository name> import <relative name> start V1`

Where **<relative name>** means the position of the directory within the module. (For example, if you have a module called *AMod* which contains a directory *BMod*, and you want to add *CMod* inside *BMod*, then change directory to *BMod*, and use *AMod/BMod* for the *relative name*).

## C7.4 Checking Out Flesh and Thorns with CVS

<b>Login</b>	<p>Prior to any CVS operation, you need to log into the Cactus repository. For an anonymous checkout, type:</p> <pre>cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus login</pre> <p>You will be prompted for a password, which is <b>anon</b>.</p>
<b>Checkout</b>	<p>To obtain a fresh copy of Cactus, move to a directory which does not contain a previously checked out version, and type</p> <pre>cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus checkout Cactus</pre> <p>The CVS checkout procedure will create a directory called <b>Cactus</b> and install the code inside this directory. From now, on we will reference all directory names relative to <b>Cactus</b>.</p> <p>If you want to compile Cactus with thorns, you now need to checkout separately the required arrangement (or <i>arrangements</i>) into the <b>arrangements</b> directory. To see the available Cactus arrangements and thorns type</p> <pre>cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus checkout -s</pre> <p>To check out an arrangement or thorn, go to the arrangements directory, <b>cd arrangements</b>, and for an arrangement type</p> <pre>cvs checkout &lt;arrangement_name&gt;</pre> <p>or for just one thorn</p> <pre>cvs checkout &lt;arrangement_name/thorn_name&gt;</pre> <p>To simplify this procedure, you may use <b>gmake checkout</b> in the Cactus home directory which provides menus to pick arrangements and thorns from.</p>
<b>Update</b>	<p>To update an existing Cactus checkout (to patch in possible changes, etc.), do the following <i>within</i> the <b>Cactus</b> directory.</p> <pre>cvs update</pre> <p>The update process will operate recursively downwards from your current position within the Cactus tree. To update only on certain directories, change into these directories and issue the update command.</p>
<b>CVS status</b>	<p>To obtain a status report on the “age” of your Cactus or arrangement routines (from your current directory position downward), type</p> <pre>cvs status</pre>
<b>non-anonymous CVS</b>	<p>If you have an account at the central repository (<b>cvs.cactuscode.org</b>) and would like to perform any of the operation above <i>non-anonymously</i>, replace <b>cvs_anon</b> by your login name and provide the appropriate password during the CVS login process. Depending on your permissions, you may then make commits to Cactus or its arrangements.</p>
<b>Commits</b>	<p>You need to perform a personalized login and have proper permissions to commit code to the repository.</p>

# Chapter C8

## 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**:

### C8.1 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 be 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

## C8.2 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.*

## Chapter C9

# ThornLists

This section still needs to be written.