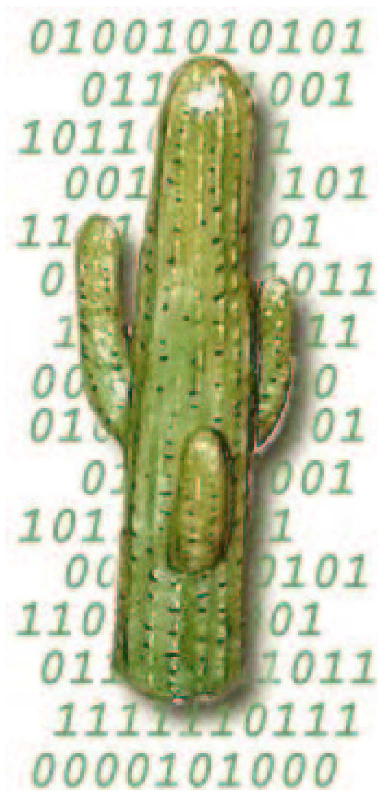
Cactus 4.0

Reference Manual



Contents

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.

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. 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

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` [A63]	Exit the code cleanly
`CCTK_WARN` [A268]	Macro to print a single string as a warning message and possibly stop the code
`CCTK_Warn` [A268]	Prints a single string as a warning message and possibly stops the code
`CCTK_VWarn` [A266]	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` [A124] Return the ancestors for an implementation

`CCTK_ImplementationThorn` [A125] Returns the name of one thorn providing an implementation.

`CCTK_ImpThornList` [A126] Return the thorns for an implementation

`CCTK_IsImplementationActive` [A146] Reports whether an implementation was activated in a parameter file

`CCTK_IsImplementationCompiled` [A147] Reports whether an implementation was compiled into a configuration

`CCTK_IsThornActive` [A148] Reports whether a thorn was activated in a parameter file

`CCTK_IsThornCompiled` [A149] Reports whether a thorn was compiled into a configuration

`CCTK_NumCompiledImplementations` [A163] Return the number of implementations compiled in

`CCTK_NumCompiledThorns` [A164] Return the number of thorns compiled in

`CCTK_ThornImplementation` [A242] 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` [A155]	Return the maximum number of active timelevels.
`CCTK_NumTimeLevels` [A170]	Deprecated; same as `CCTK_ActiveTimeLevels`.
`CCTK_GroupStorageDecrease` [A113]	Base function, overloaded by the driver, which decreases the number of active timelevels, and also returns the number of active timelevels.
`CCTK_GroupStorageIncrease` [A114]	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` [A87]	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 [A87]	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

C

```
int istat = CCTK_Barrier(const cGH *cctkGH)
```

Fortran

```
subroutine CCTK_Barrier (istat, cctkGH)
  integer               itat
  CCTK_POINTER_TO_CONST cctkGH
```

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

`cmpno`	The complex number
`realpart`	The real part of the complex number
`imagpart`	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

`absval` The computed absolute value

`realpart` 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

`addval`	The computed added value
`inval1`	The first complex number to be summed
`inval2`	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

`conjval`	The computed conjugate
`inval`	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

`cosval`	The computed cosine
`inval`	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

`divval`	The divided value
`inval1`	The enumerator
`inval1`	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

`expval`	The computed exponent
`inval`	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

`imval`	The imaginary part
`inval`	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

`mulval`	The product
`inval1`	First complex number to be multiplied
`inval2`	Second complex number to be multiplied

Discussion

The product of two complex numbers $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ is $z = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$.

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

`reval`	The real part
`inval`	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

`sinval`	The computed sine
`inval`	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

`addval`	The computed subtracted value
`inval1`	The complex number to be subtracted from
`inval2`	The complex number to subtract

Discussion

If $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ 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

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

`CCTK_CompileTime` [A39] 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 $0 \leq$ `index` $<$ `numimpls`, where `numimpls` is returned by `CCTK_NumCompiledImplementations`.

See Also

`CCTK_ActivatingThorn` [A16]	Finds the thorn which activated a particular implementation
`CCTK_CompiledThorn` [A41]	Return the name of the compiled thorn with given index
`CCTK_ImplementationRequires` [A124]	Return the ancestors for an implementation
`CCTK_ImplementationThorn` [A125]	Returns the name of one thorn providing an implementation.
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledImplementations` [A163]	Return the number of implementations compiled in
`CCTK_NumCompiledThorns` [A164]	Return the number of thorns compiled in
`CCTK_ThornImplementation` [A242]	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 $0 \leq$ `index` $<$ `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` [A124] Return the ancestors for an implementation

`CCTK_ImplementationThorn` [A125] Returns the name of one thorn providing an implementation.

`CCTK_ImpThornList` [A126] Return the thorns for an implementation

`CCTK_IsImplementationActive` [A146] Reports whether an implementation was activated in a parameter file

`CCTK_IsImplementationCompiled` [A147] Reports whether an implementation was compiled into a configuration

`CCTK_IsThornActive` [A148] Reports whether a thorn was activated in a parameter file

`CCTK_IsThornCompiled` [A149] Reports whether a thorn was compiled into a configuration

`CCTK_NumCompiledImplementations` [A163] Return the number of implementations compiled in

`CCTK_NumCompiledThorns` [A164] Return the number of thorns compiled in

`CCTK_ThornImplementation` [A242] 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 * system
```

Fortran

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

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

Parameters

index	The coordinates associated grid variable index
direction	The direction of the coordinate in this coordinate system
coordname	The name assigned to this coordinate
systemname	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

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

Fortran

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

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

Parameters

`ierr`	Error code
`direction`	Direction of coordinate in coordinate system
`gvname`	Name of grid variable associated with coordinate
`coordname`	Name of this coordinate
`systemname`	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 hierachy.

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_CoordRegisterRange( const cGH * cctkGH, CCTK_REAL min, CCTK_REAL max, i
```

Fortran

```
call CCTK_CoordRegisterRange(ierr , cctkGH, min, max, direction, coordname, systemname

integer ierr
CCTK_POINTER cctkGH
CCTK_REAL min
CCTK_REAL max
integer direction
character*(*) coordname
character*(*) systemname
```

Parameters

ierr	Error code
dimension	Pointer to CCTK grid hierachy
min	Global minimum of coordinate
max	Global maximum of coordinate
direction	Direction of coordinate in coordinate system
coordname	Name of coordinate in coordinate system
systemname	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

C

```
ierr = CCTK_CoordRegisterRange(cctkGH,-1.0,1.0,1,"x2d","cart2d");
```

Fortran

```
min = 0
max = 3.1415d0/2.0d0
two = 2
call CCTK_COORDREGISTERRANGE(ierr,min,max,two,"coordthorn::mytheta","spher3d")
```

`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

C

```
int ierr = CCTK_CoordRegisterSystem( int dimension, const char * systemname)
```

Fortran

```
call CCTK_CoordRegisterSystem(ierr , dimension, systemname )

integer ierr
integer dimension
character*(*) systemname
```

Parameters

`ierr`	Error code
`dimension`	Dimension of coordinate system
`systemname`	Unique name assigned to coordinate system

Examples

C

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

Fortran

```
three = 3
call CCTK_COORDREGISTERSYSTEM(ierr,three,"sphersystem")
```

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

C

```
int dim = CCTK_CoordSystemDim( const char * systemname)
```

Fortran

```
call CCTK_CoordSystemDim(dim , systemname )

integer dim
character*(*) systemname
```

Parameters

dim	The dimension of the coordinate system
systemname	The name of the coordinate system

Examples

C

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

Fortran

```
call  CCTK_COORDSYSTEMDIM(dim,"spher3d")
```

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

C

```
int handle = CCTK_CoordSystemHandle( const char * systemname)
```

Fortran

```
call CCTK_CoordSystemHandle(handle , systemname )

integer handle
character*(*) systemname
```

Parameters

handle	The coordinate system handle
systemname	Name of the coordinate system

Examples

C

```
handle =  CCTK_CoordSystemHandle("my coordinate system");
```

Fortran

```
call CCTK_CoordSystemHandle(handle,"my coordinate system")
```

Errors

negative	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 `pathname` 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

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

```
 */
void MyThorn_some_function(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;

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

Fortran

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

!
! assume this thorn has a string or keyword parameter  my_parameter
!
subroutine MyThorn_some_routine(CCTK_ARGUMENTS)
   implicit none
   DECLARE_CCTK_ARGUMENTS
   DECLARE_CCTK_FUNCTIONS
   DECLARE_CCTK_PARAMETERS

   if (CCTK_Equals(my_parameter, "option A") /= 0) then
      call CCTK_INFO("using option A")
   end if
end subroutine MyThorn_some_routine
```

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, $V+1$, $V+2$, ..., $V+N-1$.

See Also

`CCTK_FirstVarIndexI()`	Given a group index, returns the first variable index in the group.
`CCTK_GroupData()`	Get "static" information about a group (including the number of variables in the group).
`CCTK_GroupDynamicData()`	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, $V+1$, $V+2$, ..., $V+N-1$.

See Also

`CCTK_FirstVarIndex()`	Given a group name, returns the first variable index in the group.
`CCTK_GroupData()`	Get "static" information about a group (including the number of variables in the group).
`CCTK_GroupDynamicData()`	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 paramameter.

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

`extension`	The pointer to the GH extension
`cctkGH`	The pointer to the CCTK grid hierarchy
`name`	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

C
```
int handle = CCTK_GHExtensionHandle( const char * name)
```

Fortran
```
call CCTK_GHExtensionHandle(handle , name )

integer handle
character*(*) name
```

Parameters

`handle`	The GH extension handle
`group`	The name of the GH extension

Examples

C
```
handle =  CCTK_GHExtension("myExtension") ;
```

Fortran
```
call CCTK_GHExtension(handle,"myExtension")
```

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

`CCTK_ReduceGridArrays()` Performs reduction on a list of distributed grid arrays

`CCTK_RegisterGridArrayReductionOperator()` Registers a function as a grid array reduction operator of a certain name

`CCTK_NumGridArrayReductionOperators()` 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 (≥ 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

`status`	Return value.
`cctkGH` ($\neq$ `NULL`)	Pointer to a valid Cactus grid hierarchy.
`dim` (≥ 1)	Number of dimensions of variable.
`bbox` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`varindex`	Group index.
`varname`	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[1] contains (at least) the following members:[2]

```
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 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.
CCTK_GroupIndexFromVar [A92]	Gets the group index for a given variable name.

[1] cGroup is is a typedef for a structure. It's defined in "cctk_Group.h", which is #included by "cctk.h".
[2] Note that the members are **not** guaranteed to be declared in the order listed here.

`CCTK_GroupName` [A106]	Gets the group name for a given group index.
`CCTK_GroupNameFromVarI` [A107]	Gets the group name for a given variable name.
`CCTK_GroupTypeI` [A118]	Gets a group type index for a given group index.
`CCTK_GroupTypeFromVarI` [A117]	Gets a group type index for a given variable index.

Errors

-1	`group_index` is invalid.
-2	`group_data_buffer` 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

```c
#include "cctk.h"

int dim = CCTK_GroupDimFromVarI(int varindex);
```

Fortran

```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 associcated 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`	Sucess
`-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 group.
- lsh: The (process-)local size.
- ash: The (process-)local allocated size.
- gsh: The global grid size.
- lbnd: The lowest index of the local grid as seen on the global grid. (These use zero based indexing.)
- ubnd: The largest index of the local grid as seen on the global grid. (These use zero based indexing.)
- nghostzones: The number of ghostzones for each dimension.
- bbox: Values indicating whether these are inter-process boundaries (0) or physical boundaries (1).
- activetimelevels: The number of active time levels.

-

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` (≥ 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_GroupashGI`, `CCTK_GroupashGN`
: Returns the local allocated size for a given group.

`CCTK_GroupashVI`, `CCTK_GroupashVN`
: Returns the local allocated 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
chararacter*(*) 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

`status`	Return value.
`cctkGH` ($\neq$ `NULL`)	Pointer to a valid Cactus grid hierarchy.
`dim` (≥ 1)	Number of dimensions of variable.
`gsh` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`varindex`	Variable index.
`varname`	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_GroupashGI, CCTK_GroupashGN`
: Returns the local size for a given group.

`CCTK_GroupashVI, CCTK_GroupashVN`
: 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_GrouplbndGI, CCTK_GrouplbndGN

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_GrouplbndGI(const cGH *cctkGH,
                              int dim,
                              int *lbnd,
                              int groupindex);

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

Fortran

```
call CCTK_GrouplbndGI(status, cctkGH, dim, lbnd, groupindex)

call CCTK_GrouplbndGN(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

status	Return value.
cctkGH ($\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim (≥ 1)	Number of dimensions of group.
lbnd ($\neq$ NULL)	Pointer to array which will hold the return values.
groupindex	Group index.
groupname	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_GrouplbndVI, CCTK_GrouplbndVN`
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

status	Return value.
cctkGH ($\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim (≥ 1)	Number of dimensions of variable.
lbnd ($\neq$ NULL)	Pointer to array which will hold the return values.
varindex	Group index.
varname	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_GrouplbndGI, CCTK_GrouplbndGN`
: 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_GrouplshGI, CCTK_GrouplshGN

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_GrouplshGI(const cGH *cctkGH,
                             int dim,
                             int *lsh,
                             int groupindex);

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

Fortran

```
call CCTK_GrouplshGI(status, cctkGH, dim, lsh, groupindex)

call CCTK_GrouplshGN(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` (≥ 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_GroupgshGI`, `CCTK_GroupgshGN`
: Returns the global size for a given group.

`CCTK_GroupgshVI`, `CCTK_GroupgshVN`
: Returns the global size for a given variable.

`CCTK_GrouplshVI`, `CCTK_GrouplshVN`
: Returns the local size for a given variable.

`CCTK_GroupashGI`, `CCTK_GroupashGN`
: Returns the local allocated size for a given group.

`CCTK_GroupashVI`, `CCTK_GroupashVN`
: Returns the local allocated size for a given variable.

CCTK_GrouplshVI, CCTK_GrouplshVN

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_GrouplshVI(const cGH *cctkGH,
                             int dim,
                             int *lsh,
                             int varindex);

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

Fortran

```
call CCTK_GrouplshVI(status, cctkGH, dim, lsh, varindex)

call CCTK_GrouplshVN(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

status	Return value.
cctkGH ($\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim (≥ 1)	Number of dimensions of variable.
lsh ($\neq$ NULL)	Pointer to array which will hold the return values.
varindex	Variable index.
varname	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_GroupashGI, CCTK_GroupashGN`
: Returns the local allocated size for a given group.

`CCTK_GroupashVI, CCTK_GroupashVN`
: Returns the local allocated size for a given variable.

CCTK_GroupashGI, CCTK_GroupashGN

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

Synopsis

C

```
#include "cctk.h"

int status = CCTK_GroupashGI(const cGH *cctkGH,
                             int size,
                             int *ash,
                             int groupindex);

int status = CCTK_GroupashGN(const cGH *cctkGH,
                             int size,
                             int *ash,
                             const char *groupname);
```

Fortran

```
call CCTK_GroupashGI(status, cctkGH, size, ash, groupindex)

call CCTK_GroupashGN(status, cctkGH, size, ash, groupname)

integer       status
CCTK_POINTER  cctkGH
integer       size
integer       ash(size)
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.
`size` (≥ 1)	Size of output array, should be at least dimension of group.
`ash` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`groupindex`	Index of the group.
`groupname`	Name of the group.

Discussion

The local allocated size in each dimension for a given group 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_GrouplshVI`, `CCTK_GrouplshVN`
: Returns the local size for a given variable.

`CCTK_GroupashVI`, `CCTK_GroupashVN`
: Returns the local allocated size for a given variable.

CCTK_GroupashVI, CCTK_GroupashVN

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

Synopsis

C

```
#include "cctk.h"

int status = CCTK_GroupashVI(const cGH *cctkGH,
                             int size,
                             int *ash,
                             int varindex);

int status = CCTK_GroupashVN(const cGH *cctkGH,
                             int size,
                             int *ash,
                             const char *varname);
```

Fortran

```
call CCTK_GroupashVI(status, cctkGH, size, ash, varindex)

call CCTK_GroupashVN(status, cctkGH, size, ash, varname)

integer       status
CCTK_POINTER  cctkGH
integer       size
integer       ash(size)
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

`status`	Return value.
`cctkGH` ($\neq$ `NULL`)	Pointer to a valid Cactus grid hierarchy.
`size` (≥ 1)	Size of output array, should be at least dimension of group.
`ash` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`varindex`	Variable index.
`varname`	Variable's full name.

Discussion

The local allocated 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_GrouplshVI, CCTK_GrouplshVN`
Returns the local size for a given variable.

`CCTK_GroupashGI, CCTK_GroupashGN`
Returns the local allocated size for a given group.

CCTK_GroupName

Given a group index, returns the group name

Synopsis

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

Parameters

`name`	The group name
`index`	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

`group`	The name of the group
`varindex`	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

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

`status`	Return value.
`cctkGH` ($\neq$ `NULL`)	Pointer to a valid Cactus grid hierarchy.
`dim` (≥ 1)	Number of dimensions of group.
`nghostzones` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`varindex`	Variable index.
`varname`	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

`GH`	pointer to grid hierarchy
`n_groups`	Number of groups
`groups`	list of group indices to reduce storage for
`timelevels`	number of time levels to reduce storage for for each group
`groups`	list of group indices to allocate storage for
`status`	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

`GH`	pointer to grid hierarchy
`n_groups`	Number of groups
`groups`	list of group indices to allocate storage for
`timelevels`	number of time levels to allocate storage for for each group
`groups`	list of group indices to allocate storage for
`status`	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 Enable-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_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

`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).
`CCTK_GroupIndex` [A91]	Get the group index for a specified group name.
`CCTK_GroupIndexFromVar` [A92]	Get the group index for the group containing the variable with a specified name.
`CCTK_GroupIndexFromVarI` [A93]	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` [A117] 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

`status`	Return value.
`cctkGH` ($\neq$ `NULL`)	Pointer to a valid Cactus grid hierarchy.
`dim` (≥ 1)	Number of dimensions of group.
`ubnd` ($\neq$ `NULL`)	Pointer to array which will hold the return values.
`groupindex`	Group index.
`groupname`	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_GrouplbndGI`, `CCTK_GrouplbndGN`
: Returns the lower bounds for a given group.

`CCTK_GrouplbndVI`, `CCTK_GrouplbndVN`
: 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

status	Return value.
cctkGH ($\neq$ NULL)	Pointer to a valid Cactus grid hierarchy.
dim (≥ 1)	Number of dimensions of variable.
ubnd ($\neq$ NULL)	Pointer to array which will hold the return values.
varindex	Group index.
varname	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_GrouplbndGI, CCTK_GrouplbndGN`
Returns the lower bounds for a given group.

`CCTK_GrouplbndVI, CCTK_GrouplbndVN`
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

`implementation`	The implementation name
`index`	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` [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_ImplementationThorn` [A125]	Returns the name of one thorn providing an implementation.
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledImplementations` [A163]	Return the number of implementations compiled in
`CCTK_NumCompiledThorns` [A164]	Return the number of thorns compiled in
`CCTK_ThornImplementation` [A242]	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` [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` [A124]	Return the ancestors for an implementation
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledImplementations` [A163]	Return the number of implementations compiled in
`CCTK_NumCompiledThorns` [A164]	Return the number of thorns compiled in
`CCTK_ThornImplementation` [A242]	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` [A124] Return the ancestors for an implementation

`CCTK_ImplementationThorn` [A125] Returns the name of one thorn providing an implementation.

`CCTK_IsImplementationActive` [A146] Reports whether an implementation was activated in a parameter file

`CCTK_IsImplementationCompiled` [A147] Reports whether an implementation was compiled into a configuration

`CCTK_IsThornActive` [A148] Reports whether a thorn was activated in a parameter file

`CCTK_IsThornCompiled` [A149] Reports whether a thorn was compiled into a configuration

`CCTK_NumCompiledImplementations` [A163] Return the number of implementations compiled in

`CCTK_NumCompiledThorns` [A164] Return the number of thorns compiled in

`CCTK_ThornImplementation` [A242] 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 A127) 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

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

`CCTK_WarnCallbackRegister` 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);
```

CCTK_InterpGridArrays

Interpolate a list of distributed grid variables

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.

Synopsis

C

```c
#include "cctk.h"

int status =
     CCTK_InterpGridArrays(const cGH *cctkGH,
                           int N_dims,
                           int local_interp_handle, int param_table_handle,
                           int coord_system_handle,
                           int N_interp_points,
                             const int interp_coords_type_code,
                             const void *const interp_coords[],
                           int N_input_arrays,
                             const CCTK_INT input_array_variable_indices[],
                           int N_output_arrays,
                             const CCTK_INT output_array_type_codes[],
                             void *const output_arrays[]);
```

Fortran

```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)
```

Result

`0`	success
`< 0`	indicates an error condition (see **Errors**)

Parameters

`cctkGH` ($\neq$ `NULL`)
: Pointer to a valid Cactus grid hierarchy.

`N_dims` (≥ 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` (≥ 0)
: Handle to the local interpolation operator as returned by `CCTK_InterpHandle`.

`param_table_handle` (≥ 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` (≥ 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` (≥ 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` ($\neq$ `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` (≥ 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` ($\neq$ `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` (≥ 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` ($\neq$ `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 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.[3] 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

[3] 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

`CCTK_InterpHandle()` Get the interpolator handle for a given character-string name.

`CCTK_InterpLocalUniform()` 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.

`CCTK_ERROR_INTERP_POINT_OUTSIDE` 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 whatever thorn provides the local interpolation operator for further details)

`CCTK_ERROR_INTERP_GRID_TOO_SMALL`
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)

`CCTK_ERROR_INTERP_GHOST_SIZE_TOO_SMALL`
for a multi-processor run, the size of the interprocessor boundaries (the *ghostzone* 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 `Util_Table*` 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` (≥ 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` (≥ 0)
Handle to the interpolation operator as returned by `CCTK_InterpHandle`.

`param_table_handle` (≥ 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` ($\neq$ `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` ($\neq$ `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` (≥ 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` ($\neq$ `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` (≥ 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 <= subscript < input_array_dims[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` (≥ 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

`CCTK_InterpHandle()`	Get the interpolator handle for a given character-string name.
`CCTK_InterpGridArrays()`	Interpolate a list of Cactus grid arrays

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

`CCTK_InterpLocalNonUniform()` — 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.

`CCTK_ERROR_INTERP_POINT_OUTSIDE` — 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 AEILocalInterp thorn for further details)

`CCTK_ERROR_INTERP_GRID_TOO_SMALL` — 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)

`UTIL_ERROR_BAD_INPUT` — one or more of the inputs 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 `Util_Table*` 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` 10×20 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 (run-time) 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` (≥ 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

`CCTK_InterpHandle()`	Get the interpolator handle for a given character-string name.
`CCTK_InterpLocalUniform()`	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

C

```
int istat = CCTK_IsFunctionAliased( const char * functionname)
```

Fortran

```
call CCTK_IsFunctionAliased(istat , functionname )

integer istat
character*(*) functionname
```

Parameters

`istat` the return status

`functionname` 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` [A124], `CCTK_ImplementationThorn` [A125], `CCTK_ImpThornList` [A126], `CCTK_IsImplementationCompil` [A147], `CCTK_IsThornActive` [A148], `CCTK_NumCompiledImplementations` [A163], `CCTK_NumCompiledTho` [A164], `CCTK_ThornImplementation` [A242].

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_ImplementationRequires` [A124], `CCTK_ImplementationThorn` [A125], `CCTK_ImpThornList` [A126], `CCTK_IsImplementationActive` [A146], `CCTK_IsThornActive` [A148], `CCTK_IsThornCompiled` [A149], `CCTK_NumCompiledImplementations` [A163], `CCTK_NumCompiledThorns` [A164], `CCTK_ThornImplementation` [A242].

CCTK_IsThornActive

Reports whether a thorn was activated in a parameter file

Synopsis

C

```
#include "cctk.h"

int status = CCTK_IsThornActive(const char* thorn_name);
```

Fortran

```
#include "cctk.h"

integer status
character *(*) thorn_name

status = CCTK_IsThornActive(thorn_name)
```

Result

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

Parameters

`thorn_name` 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
`thorname`	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 `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

`CCTK_ReduceLocalArrays()` Reduces a list of local arrays (new local array reduction API)

`CCTK_LocalArrayReductionHandle()` Returns the handle of a given local array reduction operator

`CCTK_RegisterLocalArrayReductionOperator()` Registers a function as a reduction operator of a certain name

`CCTK_LocalArrayReduceOperatorImplementation()` Provide the implementation which provides an local array reduction operator

`CCTK_NumLocalArrayReduceOperators()` 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

`CCTK_ReduceLocalArrays()` Reduces a list of local arrays (new local array reduction API)

`CCTK_LocalArrayReductionHandle()` Returns the handle of a given local array reduction operator

`CCTK_RegisterLocalArrayReductionOperator()` Registers a function as a reduction operator of a certain name

`CCTK_LocalArrayReduceOperator()` Returns the name of a registered reduction operator

`CCTK_NumLocalArrayReduceOperators()` 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_LocalArrayReductionHandle(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

`CCTK_ReduceLocalArrays()` Reduces a list of local arrays (new local array reduction API)

`CCTK_RegisterLocalArrayReductionOperator()` Registers a function as a reduction operator of a certain name

`CCTK_LocalArrayReduceOperatorImplementation()` Provide the implementation which provides an local array reduction operator

`CCTK_LocalArrayReduceOperator()` Returns the name of a registered reduction operator

`CCTK_NumLocalArrayReduceOperators()` 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

C
```
int numlevels = CCTK_MaxTimeLevels( const char * name)
```

Fortran
```
call CCTK_MaxTimeLevels(numlevels , name )

integer numlevels
character*(*) name
```

Parameters

`name`	The full group name
`numlevels`	The number of timelevels

Discussion

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

Examples

C
```
numlevels = CCTK_MaxTimeLevels("evolve::phivars");
```

Fortran
```
call CCTK_MAXTIMELEVELS(numlevels,"evolve::phivars")
```

`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

C
```
int numlevels = CCTK_MaxTimeLevelsVN( const char * name)
```

Fortran
```
call CCTK_MaxTimeLevelsVN(numlevels , name )

integer numlevels
character*(*) name
```

Parameters

`name`	The full variable name
`numlevels`	The number of timelevels

Discussion

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

Examples

C
```
numlevels = CCTK_MaxTimeLevelsVN("evolve::phi")
```

Fortran
```
call CCTK_MAXTIMELEVELSVN(numlevels,"evolve::phi")
```

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 $0 \leq \text{myproc} <$ `CCTK_nProcs(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

C
```
int nprocs = CCTK_nProcs( const cGH * cctkGH)
```

Fortran
```
nprocs = CCTK_nProcs(  cctkGH )

integer nprocs
CCTK_POINTER cctkGH
```

Parameters

`cctkGH` 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 it(s) 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` [A124]	Return the ancestors for an implementation
`CCTK_ImplementationThorn` [A125]	Returns the name of one thorn providing an implementation.
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledThorns` [A164]	Return the number of thorns compiled in
`CCTK_ThornImplementation` [A242]	Returns the implementation provided by the thorn

CCTK_NumCompiledThorns

Return the number of thorns compiled in.

Synopsis

C

```
#include "cctk.h"

int numthorns = CCTK_NumCompiledThornss();
```

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` [A124]	Return the ancestors for an implementation
`CCTK_ImplementationThorn` [A125]	Returns the name of one thorn providing an implementation.
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledImplementations` [A163]	Return the number of implementations compiled in
`CCTK_ThornImplementation` [A242]	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

`CCTK_ReduceGridArrays()`	Performs reduction on a list of distributed grid arrays
`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 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

C
```
int num_methods = CCTK_NumIOMethods (void);
```

Fortran
```
call CCTK_NumIOMethods (num_methods)
integer num_methods
```

Parameters

`num_methods` 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

`CCTK_ReduceLocalArrays()` Reduces a list of local arrays (new local array reduction API)

`CCTK_LocalArrayReductionHandle()` Returns the handle of a given local array reduction operator

`CCTK_RegisterLocalArrayReductionOperator()` Registers a function as a reduction operator of a certain name

`CCTK_LocalArrayReduceOperatorImplementation()` Provide the implementation which provides an local array reduction operator

`CCTK_LocalArrayReduceOperator()` 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

`CCTK_ReduceArraysGlobally()` Reduces a list of arrays globally

`CCTK_LocalArrayReductionHandle()` Returns the handle of a given local array reduction operator

`CCTK_RegisterReduceArraysGloballyOperator()` 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` [A155]	Return the maximum number of active timelevels.
`CCTK_GroupStorageDecrease` [A113]	Base function, overloaded by the driver, which decreases the number of active timelevels, and also returns the number of active timelevels.
`CCTK_GroupStorageIncrease` [A114]	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 themselfes 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

CCTK_OutputVar

Output a single variable by all I/O methods

Synopsis

C

```
int istat = CCTK_OutputVar (const cGH *cctkGH,
                            const char *variable);
```

Fortran

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

Parameters

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

Discussion

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

Errors

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

CCTK_OutputVarAs

Output a single variable as an alias by all I/O methods

Synopsis

C

```
int istat = CCTK_OutputVarAs (const cGH *cctkGH,
                              const char *variable,
                              const char *alias);
```

Fortran

```
call CCTK_OutputVarAsByMethod (istat, cctkGH, variable, alias)
integer istat
CCTK_POINTER cctkGH
character*(*) variable
character*(*) alias
```

Parameters

`istat`	return status
`cctkGH`	pointer to CCTK grid hierarchy
`variable`	full name of variable to output, with an optional options string in curly braces
`alias`	alias name to base the output filename on

Discussion

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.

Errors

`positive`	the number of IO methods which did output of `variable`
`0`	for success
`negative`	if no IO methods were registered

CCTK_OutputVarAsByMethod

Synopsis

C

```
int istat = CCTK_OutputVarAsByMethod (const cGH *cctkGH,
                                      const char *variable,
                                      const char *method,
                                      const char *alias);
```

Fortran

```
call CCTK_OutputVarAsByMethod (istat, cctkGH, variable, method, alias)
integer istat
CCTK_POINTER cctkGH
character*(*) variable
character*(*) method
character*(*) alias
```

Parameters

`istat`	return status
`cctkGH`	pointer to CCTK grid hierarchy
`variable`	full name of variable to output, with an optional options string in curly braces
`method`	method to use for output
`alias`	alias name to base the output filename on

Discussion

Output a variable `variable` using the method `method` if it is registered. Uses `alias` as the name of the variable for the purpose of constructing a filename. The output should take place if at all possible. If the appropriate file exists the data is appended, otherwise a new file is created.

Errors

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

CCTK_OutputVarByMethod

Synopsis

C

```
int istat = CCTK_OutputVarByMethod (const cGH *cctkGH,
                                    const char *variable,
                                    const char *method);
```

Fortran

```
call CCTK_OutputVarByMethod (istat, cctkGH, variable, method)
integer istat
CCTK_POINTER cctkGH
character*(*) variable
character*(*) method
```

Parameters

`istat`	return status
`cctkGH`	pointer to CCTK grid hierarchy
`variable`	full name of variable to output, with an optional options string in curly braces
`method`	method to use for output

Discussion

Output a variable **`variable`** using the IO method **`method`** if it is registered. The output should take place if at all possible. if the appropriate file exists the data is appended, otherwise a new file is created.

Errors

`0`	for success
`negative`	indicating some error (e.g. IO method is not 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 [A183]	Get the data pointer to and type of a parameter's value
CCTK_ParameterLevel [A184]	Return the parameter checking level
CCTK_ParameterQueryTimesSet [A185]	Return number of times a parameter has been set
CCTK_ParameterSet [A186]	Sets the value of a parameter
CCTK_ParameterValString [A191]	Get the string representation of a parameter's value
CCTK_ParameterWalk [A193]	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 [A182]	Get parameter properties for given parameter/thorn pair
CCTK_ParameterLevel [A184]	Return the parameter checking level
CCTK_ParameterQueryTimesSet [A185]	Return number of times a parameter has been set
CCTK_ParameterSet [A186]	Sets the value of a parameter
CCTK_ParameterValString [A191]	Get the string representation of a parameter's value
CCTK_ParameterWalk [A193]	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` [A182]	Get parameter properties for given parameter/thorn pair
`CCTK_ParameterGet` [A183]	Get the data pointer to and type of a parameter's value
`CCTK_ParameterQueryTimesSet` [A185]	Return number of times a parameter has been set
`CCTK_ParameterSet` [A186]	Sets the value of a parameter
`CCTK_ParameterValString` [A191]	Get the string representation of a parameter's value
`CCTK_ParameterWalk` [A193]	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

`CCTK_ParameterData` [A182]	Get parameter properties for given parameter/thorn pair
`CCTK_ParameterGet` [A183]	Get the data pointer to and type of a parameter's value
`CCTK_ParameterLevel` [A184]	Return the parameter checking level
`CCTK_ParameterSet` [A186]	Sets the value of a parameter
`CCTK_ParameterValString` [A191]	Get the string representation of a parameter's value
`CCTK_ParameterWalk` [A193]	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` [A182] Get parameter properties for given parameter/thorn pair

`CCTK_ParameterLevel` [A184] Return the parameter checking level

`CCTK_ParameterQueryTimesSet` [A185] Return number of times a parameter has been set

`CCTK_ParameterSetNotifyRegister` [A188] Registers a parameter set operation notify callback

`CCTK_ParameterSetNotifyUnregister` [A190] Unregisters a parameter set operation notify callback

`CCTK_ParameterValString` [A191] Get the string representation of a parameter's value

`CCTK_ParameterWalk` [A193] 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,
.                                     name, thorn_regex, param_regex)
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

`callback`	Function pointer of the notify callback to be registered
`data`	optional user-defined data pointer to associate with the notify callback
`name`	Unique name under which the notify callback is to be registered
`thorn_regex`	Optional regular expression string to match a thorn name in a full parameter name
`param_regex`	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()`), plus an optional callback data pointer defined by the user at registration time. When a callback function is registered with `CCTK_ParameterSetNotify()`, the calling routine may also pass an optional regular expression string for both a thorn name and a parameter name to match against in a parameter set notification; leave them empty or pass a NULL pointer to get notified about changes of *any* parameter.

Registered notification callbacks would be invoked by `CCTK_ParameterSet()` only *after* initial parameter setup from the parfile, and – in case of recovery – only *after* all parameters have been restored from the checkpoint file. The callbacks are then invoked just *before* the parameter is set to its new value so that they can still query its old value if necessary.

See Also

`CCTK_ParameterSet` [A186] Sets the value of a parameter

`CCTK_ParameterSetNotifyUnregister` [A190]
Unregisters a parameter set operation notify callback

Examples

C

```
#include <stdio.h>

#include "cctk.h"

static void ParameterSetNotify (void *unused,
                                const char *thorn,
                                const char *parameter,
                                const char *new_value)
{
  printf ("parameter set notification: %s::%s is set to '%s'\n",
          thorn, parameter, new_value);
}

void RegisterNotifyCallback (void)
{
  /* we are interested only in this thorn's parameters
     so pass the thorn name in the 'thorn_regex' argument */
  if (CCTK_ParameterSetNotifyRegister (ParameterSetNotify, NULL, CCTK_THORNSTRING,
                                       CCTK_THORNSTRING, NULL))
  {
    CCTK_VWarn (0, __LINE__, __FILE__, CCTK_THORNSTRING,
                "Couldn't register parameter set notify callback");
  }
}
```

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 [A186]	Sets the value of a parameter
CCTK_ParameterSetNotifyRegister [A188]	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

`CCTK_ParameterData` [A182] Get parameter properties for given parameter/thorn pair

`CCTK_ParameterGet` [A183] Get the data pointer to and type of a parameter's value

`CCTK_ParameterLevel` [A184] Return the parameter checking level

`CCTK_ParameterQueryTimesSet` [A185] Return number of times a parameter has been set

`CCTK_ParameterSet` [A186] Sets the value of a parameter

`CCTK_ParameterWalk` [A193] 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` [A182]	Get parameter properties for given parameter/thorn pair
`CCTK_ParameterGet` [A183]	Get the data pointer to and type of a parameter's value
`CCTK_ParameterLevel` [A184]	Return the parameter checking level
`CCTK_ParameterQueryTimesSet` [A185]	Return number of times a parameter has been set
`CCTK_ParameterSet` [A186]	Sets the value of a parameter
`CCTK_ParameterValString` [A191]	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 it(s) 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_ReduceArraysGlobally(status,
.                                GH,
.                                dest_proc,
.                                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_proc,
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` (≥ 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` (≥ 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` (≥ 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` (≥ 0)
: The number of dimensions of the input arrays

`input_array_dims` (≥ 0)
: (Pointer to) an array of size `input_dims` containing the dimensions of the arrays to be reduced.

`input_array_type_codes` (≥ 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` (≥ 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.

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 (≠ NULL)`
Pointer to a valid Cactus grid hierarchy.

`dest_processor` The destination processor. -1 will distribute the result to all processors.

`local_reduce_handle` (≥ 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` (≥ 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` (≥ 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` (≥ 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

`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_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` (≥ 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` (≥ 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

`CCTK_LocalArrayReductionHandle()`
: Returns the handle of a given local array reduction operator

`CCTK_RegisterLocalArrayReductionOperator()`
: Registers a function as a reduction operator of a certain name

`CCTK_LocalArrayReduceOperatorImplementation()`
: Provide the implementation which provides an local array reduction operator

`CCTK_LocalArrayReduceOperator()`
: Returns the name of a registered reduction operator

`CCTK_NumLocalArrayReduceOperators()`
: 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

C
```
int handle = CCTK_ReductionHandle( const char * reduction)
```

Fortran
```
handle = CCTK_ReductionHandle(  reduction )

integer handle
character*(*) reduction
```

Parameters

`handle` handle returned for this method

`name` 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

C
```
handle = CCTK_ReductionHandle("maximum");
```

Fortran
```
call CCTK_ReductionHandle(handle,"maximum")
```

CCTK_RegexMatch

Perform a regular expression match of string against pattern

Synopsis

C

```
success = CCTK_RegexMatch( const char *string, const char *pattern,
                           const int nmatch, regmatch_t *pmatch)
```

Parameters

`string`	String to match against
`pattern`	Regex pattern
`nmatch`	The size of the pmatch array
`pmatch`	Array in which to place the matches

Result

`0`	pattern does not match
`1`	pattern matches
`< 0`	indicates an error condition (pattern did not compile as a regular expression)

Discussion

Perform a regular expression match of string against pattern. Also returns the specified number of matched substrings as give by regexec. This is a modified form of the example routine given in the SGI man page for regcomp.

Examples

C

```
#define R_BEGIN "(\\[|\\()?"
#define R_VALUE "([^]):]*)"
#define R_SEP   ":"
#define R_END   "(\\]|\\))?"
#define R_MAYBE(x) "(" x ")?"

  int matched;
  const char pattern[] =
    R_BEGIN
    R_VALUE
    R_MAYBE(R_SEP R_VALUE R_MAYBE(R_SEP R_VALUE))
    R_END;

  if( (matched = CCTK_RegexMatch(range, pattern, 8, pmatch)) > 0) {
      CCTK_VInfo(CCTK_THORNSTRING, "'%s' is a valid range specifier",
                 range);
  } else if(!matched) {
      CCTK_VInfo(CCTK_THORNSTRING, "'%s' is not a valid range specifier",
                 range);
  } else {
      CCTK_VInfo(CCTK_THORNSTRING, "invalid pattern '%s'", pattern);
  }
```

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_RegisterGHExtension

Register an extension to the CactusGH

Synopsis

C
```
int istat = CCTK_RegisterGHExtension( const char * name)
```

CCTK_RegisterGHExtensionInitGH

Register a function which will initialise a given extension to the Cactus GH

Synopsis

C `int istat = CCTK_RegisterGHExtensionInitGH( 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 occured.

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

`CCTK_ReduceLocalArrays()` Reduces a list of local arrays (new local array reduction API)

`CCTK_LocalArrayReductionHandle()` Returns the handle of a given local array reduction operator

`CCTK_LocalArrayReduceOperatorImplementation()` Provide the implementation which provides an local array reduction operator

`CCTK_LocalArrayReduceOperator()` Returns the name of a registered reduction operator

`CCTK_NumLocalArrayReduceOperators()` 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 occured.

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_ScheduleQueryCurrentFunction

Return the cFunctionData of the function currently executing via `CCTK_CallFunction`.

Synopsis

C

```
#include "cctk.h"
const cFunctionData *CCTK_ScheduleQueryCurrentFunction(const cGH *GH)
```

Result

Data of last call to `CCTK_CallFunction`, or

`NULL` if not within a scheduled function.

Parameters

`cctkGH` Pointer to a Cactus grid hierarchy.

Discussion

Returns a data structure containing the thorn and routine name of the currently executing function as well as the Cactus bin name. If no function is currently executing, returns `NULL`. This is intended to be used by thorns providing callable functions to identify their caller when reporting errors.

See Also

`CCTK_CallFunction` Calls a function depending upon the data passed in the the fdata structure.

Examples

C Output the name of the currently scheduled function:

```
#include <stdio.h>
#include "cctk.h"
const cFunctionData *fdata = CCTK_ScheduleQueryCurrentFunction(cctkGH);
printf("scheduled function: %s::%s AT %s\n",
       fdata->thorn, fdata->routine, fdata->where);
```

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` [A236]	Synchronise the ghostzones for a group of grid variables (identified by the group index)
`CCTK_SyncGroupsI` [A238]	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` [A234] Synchronise the ghostzones for a group of grid variables (identified by the group name)

`CCTK_SyncGroupsI` [A238] 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` `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 */

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

CCTK_SyncGroup [A234]	Synchronise the ghostzones for a single group of grid variables (identified by the group name)
CCTK_SyncGroupI [A236]	Synchronise the ghostzones for a single group of grid variables (identified by the group index)
CCTK_GroupIndex [A91]	Gets the group index for a given group name.
CCTK_GroupIndexFromVar [A92]	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` [A241] 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 [A240] — 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` [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` [A124]	Return the ancestors for an implementation
`CCTK_ImplementationThorn` [A125]	Returns the name of one thorn providing an implementation.
`CCTK_ImpThornList` [A126]	Return the thorns for an implementation
`CCTK_IsImplementationActive` [A146]	Reports whether an implementation was activated in a parameter file
`CCTK_IsImplementationCompiled` [A147]	Reports whether an implementation was compiled into a configuration
`CCTK_IsThornActive` [A148]	Reports whether a thorn was activated in a parameter file
`CCTK_IsThornCompiled` [A149]	Reports whether a thorn was compiled into a configuration
`CCTK_NumCompiledImplementations` [A163]	Return the number of implementations compiled in
`CCTK_NumCompiledThorns` [A164]	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_TraverseString

Traverse through all variables and/or groups whose names appear in the given string, and call the callback routine with those indices and an optional option string appended to the variable/group name enclosed in square braces. The special keyword "all" in the string can be used to indicate that the callback should be called for all variables/groups.

Synopsis

C `int err = CCTK_TraverseString(traverse_string, callback, callback_arg, selection)`

Parameters

`const char * traverse_string`
: List of variable and/or group names

`void (*callback) (int idx, const char *optstring, void *callback_arg)`
: Routine to call for every variable and/or group found. `idx` is the Cactus variable index, `optstring` is the optional '`{}`' enclosed option string after the variable name, and `callback_arg` is the arbitrary argument passed to `CCTK_TraverseString`.

`void *callback_arg`
: An arbitrary argument which gets passed to the callback routine

`int selection`
: Decides whether group and/or variable names are accepted in the string. Possible values are: `CCTK_VAR`, `CCTK_GROUP` or `CCTK_GROUP_OR_VAR`.

Discussion

Use this to loop over a list of variables passed in by the user.

Result

`number of variables`
: positive for the number of traversed variables

Errors

`-1`	no callback routine was given
`-2`	option string is not associated with a group or variable
`-3`	unterminated option string
`-4`	garbage found at end of option string
`-5`	invalid token in traversed string found

CCTK_VarDataPtr

Returns the data pointer for a grid variable

Synopsis

C
```
void * ptr = CCTK_VarDataPtr( const cGH * cctkGH, int timelevel, char * name)
```

Fortran
```
call CCTK_VarDataPtr(ptr, cctkGH, timelevel, varname)
                     CCTK_POINTER vardataptr
                     CCTK_POINTER_TO_CONST cctkGH
                     integer timelevel
                     character*(*) varname
```

Parameters

ptr	a void pointer to the grid variable data
cctkGH	pointer to CCTK grid hierarchy
timelevel	The timelevel of the grid variable
name	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"))
```

Fortran
```
CCTK_REAL, dimension(cctk_ash(1),cctk_ash(2),cctk_ash(3)) :: var
CCTK_POINTER myVar
pointer (myVar, var)
call CCTK_VarDataPtr(myVar,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 * nam
```

Parameters

`ptr`	a void pointer to the grid variable data
`cctkGH`	pointer to CCTK grid hierarchy
`timelevel`	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)
```

Fortran

```
call CCTK_VarDataPtrI(ptr, cctkGH, timelevel, index)
                      CCTK_POINTER vardataptr
                      CCTK_POINTER_TO_CONST cctkGH
                      integer timelevel
                      integer index
```

Parameters

`cctkGH`	pointer to CCTK grid hierarchy
`timelevel`	The timelevel of the grid variable
`index`	The index of the variable

Examples

C

```
myVar = (CCTK_REAL *)(CCTK_VarDataPtr(GH,0,CCTK_VarIndex("imp::realvar")));
```

Fortran

```
CCTK_REAL, dimension(cctk_ash(1),cctk_ash(2),cctk_ash(3)) :: var
CCTK_POINTER myVar
pointer (myVar, var)
call CCTK_VarDataPtr(myVar,GH,0,CCTK_VarIndex("imp::realvar"))
```

CCTK_VarIndex

Get the index for a variable.

Synopsis

C

```
#include "cctk.h"
int index = CCTK_VarIndex(const char *varname);
```

Fortran

```
call CCTK_VarIndex(index, varname)
                   integer index
                   character*(*) varname
```

Parameters

`varname`	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. For vector variables, the zero-based component index should be included in square brackets after the variable name.

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

C

```
index = CCTK_VarIndex("evolve::phi");
index = CCTK_VarIndex("evolve::vect[0]");
```

Fortran

```
call CCTK_VarIndex(index,"evolve::phi")
call CCTK_VarIndex(index,"evolve::vect[0]")
```

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 sceen

Synopsis

C

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

int status = CCTK_VInfo(const char *thorn,
                        const char *format,
                        ...);
```

Result

0 ok

Parameters

thorn — The name of the thorn printing this info message. You can use the CCTK_THORNSTRING macro here (defined in cctk.h).

format — The printf-like format string to use for printing the info message.

... — 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 [A127] 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[4]

Parameters

`level` (≥ 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 `CCTK_VWarn` call occured. You can use the standardized `__LINE__` preprocessor macro here.
`file`	The file name of the originating source file where the `CCTK_VWarn` call occured. You can use the standardized `__FILE__` preprocessor macro here.
`thorn`	The thorn name of the originating source file where the `CCTK_VWarn` call occured. You can use the `CCTK_THORNSTRING` macro here (defined in `cctk.h`).
`format`	The `printf`-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.

[4] 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 ≥ the `-E` level ≥ 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 `#include`d 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` [A268] 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

`level` The warning level to use; see the description of `CCTK_VWarn()` on page A265 for a detailed discussion of this parameter and the Cactus macros for standard warning levels

`message` 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).[5] 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

`CCTK_VWarn()` 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
```

[5] 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

`minlevel` The minimum warning level to use.

You can find a detailed discussion of the Cactus macros for standard warning levels on page A265. Both minlevel and maxlevel follow that definition.

`maxlevel` The maximum warning level to use

`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 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 A268) 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

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

`CCTK_VWarn()` 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

B1.2 String Functions

B1.3 Table Functions

`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

`Util_TablePrint` [B79] Print out a table and its data structures, using a verbose internal format meant for debugging

`Util_TablePrintAll`
[B80] Print out all tables and their data structures, using a verbose internal format meant for debugging

`Util_TablePrintAllIterators`
[B81] Print out all table iterators and their data structures, using a verbose internal format meant for debugging

`Util_TablePrintPretty`
[B82] Print out a table, using a human-readable format similar to the one accepted by Util_TableCreateFromString

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

`Util_CurrentDate` [B7]	Fills string with current local date
`Util_CurrentTime` [B9]	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,[1] so Cactus provides its own version, `Util_snprintf()`.[2]

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

`Util_vsnprintf` [B12]	Similar function which takes a `<stdarg.h>` variable argument list.
`snprintf()`	Standard C library function which this function tries to clone.
`sprintf()`	Unsafe and dangerous C library function similar to `snprintf()`, which doesn't check the buffer length.

Errors

$<$ `0`	Some sort of error occured. It's indeterminate what (if anything) has been stored in the destination buffer.

Examples

C

```
#include "util_String.h"
```

[1] There's also a related (older) API `sprintf()`. Don't use it – it almost guarantees buffer overflows.
[2] 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 occured)
         */
        }
```

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 arguements 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 from of a variable length argument list.

See Also

`Util_snprintf` [B10]	Similar function which takes a variable length argument list.
`vsnprintf()`	Standard C library function which this function tries to clone.
`vsprintf()`	Unsafe and dangerous C library function similar to `vsnprintf()`, which doesn't check the buffer length.
`<stdarg.h>`	System header file which defines the `va_list` "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 `va_list` 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 `strcasecmp()` 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 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,[1] where lower-case $\leftrightarrow$ upper-case mappings

[1] 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.[2] 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 */ }
```

[2]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–malloced 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

`<stdlib.h>` System header file containing prototypes for `malloc()` and `free`.

`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_Strlcpy()` [B20] 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 occured
 */
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_Strlcpy(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_Strlcpy(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-`malloc`ed 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_Strlcpy(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_Strlcpy(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_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 occurence 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` (≥ 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

C

```
#include "util_ErrorCodes.h"
#include "util_Table.h"
int handle = Util_TableCreate(int flags);
```

Fortran

```
call Util_TableCreate(handle, flags)
integer handle, flags
```

Result

`handle` (≥ 0) A handle to the newly-created table

Parameters

`flags` (≥ 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 of 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

`Util_StrCmpi()` [B14] compare two strings, ignoring upper/lower case

`Util_TableClone()` [B26] create a new table which is a "clone" (exact copy) of an existing 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

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` (≥ 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

C

```
#include "util_ErrorCodes.h"
#include "util_Table.h"
int key_exists = Util_TableDeleteKey(int handle, const char *key);
```

Fortran

```
call Util_TableDeleteKey(key_exists, handle, key)
integer        key_exists, handle
character*(*)  key
```

Result

0 ok (key existed before this call, and has now been deleted)

Parameters

`handle` (≥ 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` (≥ 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[1], 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 (≥ 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

[1] 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.

`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*Array()`	get an array value
`Util_TableGetString()` [B43]	get a character-string 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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

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_ERROR_TABLE_WRONG_DATA_TYPE`	value has data type other than `CCTK_TYPE`
`UTIL_ERROR_TABLE_VALUE_IS_EMPTY`	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[2], 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

`N_elements` the number of array elements in the value

Parameters

`handle` (≥ 0) handle to the table

`N_array` the number of array elements in `array[]` (must be ≥ 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 > `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 < `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 `CCTK_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

[2] For backwards compatability the function `Util_TableGetFnPointerArray()` is also provided as an alias for `Util_TableGetFPointerArray()`. This is deprecated as of Cactus 4.0 beta 13.

`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_TableGetGeneric()` [B38]	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
`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

Errors

`UTIL_ERROR_BAD_HANDLE`	handle is invalid
`UTIL_ERROR_TABLE_BAD_KEY`	key contains '/' character
`UTIL_ERROR_BAD_INPUT`	`array != NULL` and `N_array` < 0
`UTIL_ERROR_TABLE_NO_SUCH_KEY`	no such key in table
`UTIL_ERROR_TABLE_WRONG_DATA_TYPE`	value has data type other than `CCTK_TYPE`

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 (≥ 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

`Util_TableGet*()`	get a single (1-element array) value
`Util_TableGet*Array()`	get an array value
`Util_TableGetString()` [B43]	get a character-string value
`Util_TableQueryValueInfo()` [B61]	query key present/absent in table, and optionally type and/or number of elements
`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

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_ERROR_TABLE_WRONG_DATA_TYPE`	value has data type other than `CCTK_TYPE`
`UTIL_ERROR_TABLE_VALUE_IS_EMPTY`	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

```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

```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` (≥ 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 ≥ 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 $>$ `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 $<$ `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

`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_TableGetGeneric()` [B38]	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
`Util_TableQueryValueInfo()` [B61]	query key present/absent in table, and optionally type and/or number of elements
`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

Errors

`UTIL_ERROR_BAD_HANDLE`	handle is invalid
`UTIL_ERROR_TABLE_BAD_KEY`	key contains '/' character
`UTIL_ERROR_BAD_INPUT`	`array != NULL` and `N_array` < 0
`UTIL_ERROR_TABLE_NO_SUCH_KEY`	no such key in table
`UTIL_ERROR_TABLE_WRONG_DATA_TYPE`	value has data type other than `CCTK_TYPE`

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` (≥ 0) handle to the table

`buffer_length` the length (`sizeof`) of `buffer[]` (must be ≥ 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_CHAR`s, *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

`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_TableGetCharArray()` [B36]	get an array-of-`CCTK_CHAR` value
`Util_TableGetGeneric()` [B38]	get a single (1-element array) value with generic data type
`Util_TableGetGenericArray()` [B40]	get an array value with generic data type
`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_TableSetString()` [B77]	set a character-string value
`Util_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`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_BAD_INPUT`	`buffer != NULL` and `buffer_length` ≤ 0
`UTIL_ERROR_TABLE_NO_SUCH_KEY`	no such key in table
`UTIL_ERROR_TABLE_WRONG_DATA_TYPE`	value has data type other than `CCTK_CHAR`
`UTIL_ERROR_TABLE_STRING_TRUNCATED`	`buffer != NULL` and value was truncated to fit in `buffer[]`

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

`ihandle` (≥ 0)	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 automagically 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

`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_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` (≥ 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()` [B26]	create a new table which is a "clone" (exact copy) of an existing table
`Util_TableItCreate()` [B48]	create a table iterator
`Util_TableItDestroy()` [B49]	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` (≥ 0) handle to the table iterator

Parameters

`handle` (≥ 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` (≥ 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 *not* in the "null-pointer" state, i.e. iterator points to some table entry

0 — iterator is in the "null-pointer" state

Parameters

`ihandle` (≥ 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` (≥ 0) handle to the table iterator

Discussion

If no errors occur, `Util_TableItQueryIsNull(ihandle)` is the same as `1 - Util_TableItQueryIsNonNul`

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` (≥ 0) handle to the table iterator

`key_buffer_length` the length (`sizeof`) of `key_buffer[]` (must be ≥ 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` (≥ 0) handle to the table over which the iterator iterates

Parameters

`ihandle` (≥ 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 *not* 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` (≥ 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_TableItSetToNull()` [B58]	set an iterator to the "null-pointer" state
`Util_TableItSetToKey()` [B57]	set an iterator to point to a specified table entry

Errors

`UTIL_ERROR_BAD_HANDLE`	iterator handle is invalid

Util_TableItSetToKey

Set a table iterator to point to a specified table entry

Synopsis

C

```c
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TableItSetToKey(int ihandle, const char *key);
```

Result

0 ok

Parameters

`ihandle` (≥ 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

UTIL_ERROR_BAD_HANDLE	iterator handle is invalid
UTIL_ERROR_TABLE_BAD_KEY	key contains '/' character
UTIL_ERROR_TABLE_NO_SUCH_KEY	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` (≥ 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` (≥ 0) the flags word

Parameters

`handle` (≥ 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` (≥ 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` (≥ 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` (≥ 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` (≥ 0)	the number of key/value entries in the table

Parameters

`handle` (≥ 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[3], 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 (≥ 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

[3]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.

`Util_TableGet*Array()`	get an array value
`Util_TableGetGeneric()` [B38]	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
`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string 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 <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[4], 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 (≥ 0) handle to the table

N_elements (≥ 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.)

[4] 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

`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()` [B38]	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
`Util_TableSet*()`	set a single (1-element 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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

Errors

`UTIL_ERROR_BAD_HANDLE`	handle is invalid
`UTIL_ERROR_TABLE_BAD_KEY`	key contains '/' character
`UTIL_ERROR_BAD_INPUT`	`N_elements` < 0
`UTIL_ERROR_NO_MEMORY`	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` (≥ 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 strdol(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

`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()` [B38] : 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

`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_TableSetString()` [B77] : set a character-string value

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
`UTIL_ERROR_NO_MIXED_TYPE_ARRAY`	invalid input: different array values have different datatypes
other error codes	this function may also return any error codes returned by `Util_TableSetString()`, `Util_TableSetInt()`, `Util_TableSetReal()`, `Util_TableSetIntArray()`, or `Util_TableSetRealArray()`.

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");
```

Util_TableSetGeneric

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 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 (≥ 0)	handle to the table
type_code	the array elements' type code (one of the CCTK_VARIABLE_* constants from "cctk_Constants.h")
value_ptr	a pointer to 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
Util_TableGet*Array()	get an array value
Util_TableGetGeneric() [B38]	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

`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

Errors

`UTIL_ERROR_BAD_HANDLE`	handle is invalid
`UTIL_ERROR_BAD_INPUT`	`type_code` is invalid
`UTIL_ERROR_TABLE_BAD_KEY`	key contains '/' character
`UTIL_ERROR_NO_MEMORY`	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

C

```
#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);
```

Fortran

```
call Util_TableSetGenericArray(status,
.                              handle,
.                              type_code,
.                              N_elements, array,
.                              key)
integer           status, handle, type_code, N_elements
CCTK_POINTER(*)   array
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 `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` (≥ 0) handle to the table

`type_code` the array elements' type code (one of the `CCTK_VARIABLE_*` constants from `"cctk_Constants.h"`)

`N_elements` (≥ 0) the number of array elements in `array[]`

`value_ptr` a pointer to 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.

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

`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()` [B38]	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
`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_TableSetFromString()` [B69]	convenience routine to set key/value entries in a table based on a parameter-file–like character string
`Util_TableSetString()` [B77]	set a character-string value

Errors

`UTIL_ERROR_BAD_HANDLE`	handle is invalid
`UTIL_ERROR_BAD_INPUT`	`type_code` is invalid
`UTIL_ERROR_TABLE_BAD_KEY`	key contains '/' character
`UTIL_ERROR_NO_MEMORY`	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 `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` (≥ 0)	handle to the table
`string`	a pointer to the string (a C-style null-terminated string)
`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 string is stored as an array of `strlen(string)` `CCTK_CHAR`s. 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

`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()` [B38] 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
`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");
```

Util_TablePrint

Print out a table and its data structures, using a verbose internal format meant for debugging

Synopsis

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TablePrint(FILE *stream,
                             int handle);
```

Result

0 ok

Parameters

`stream` ($\neq 0$) output stream, e.g. `stdout`

`handle` (≥ 0) handle to the table

Discussion

`stream` may be any output stream, e.g. `stdout` or `stderr`, or a file that has been opened for writing.

See Also

`Util_TablePrintAll()` [B80] Print out all tables and their data structures, using a verbose internal format meant for debugging

`Util_TablePrintAllIterators()` [B81] Print out all table iterators and their data structures, using a verbose internal format meant for debugging

`Util_TablePrintPretty()` [B82] Print out a table, using a human-readable format similar to the one accepted by Util_TableCreateFromString

Examples

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"

int handle = Util_TableCreateFromString("ipar=1 dpar=2.0 spar='three'");
Util_TablePrint(stdout, handle);
```

Util_TablePrintAll

Print out all tables and their data structures, using a verbose internal format meant for debugging

Synopsis

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TablePrintAll(FILE *stream);
```

Result

0	ok

Parameters

`stream` ($\neq 0$)	output stream, e.g. `stdout`

Discussion

`stream` may be any output stream, e.g. `stdout` or `stderr`, or a file that has been opened for writing.

See Also

`Util_TablePrint()` [B79]	Print out a table and its data structures, using a verbose internal format meant for debugging
`Util_TablePrintAllIterators()` [B81]	Print out all table iterators and their data structures, using a verbose internal format meant for debugging
`Util_TablePrettyPrint()` [B82]	Print out a table, using a human-readable format similar to the one accepted by Util_TableCreateFromString

Examples

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"

int handle = Util_TableCreateFromString("ipar=1 dpar=2.0 spar='three'");
Util_TablePrintAll(stdout);
```

Util_TablePrintAllIterators

Print out all table iterators and their data structures, using a verbose internal format meant for debugging

Synopsis

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TablePrintAllIterators(FILE *stream);
```

Result

0	ok

Parameters

`stream` ($\neq 0$)	output stream, e.g. `stdout`

Discussion

`stream` may be any output stream, e.g. `stdout` or `stderr`, or a file that has been opened for writing.

See Also

`Util_TablePrint()` [B79]	Print out a table and its data structures, using a verbose internal format meant for debugging
`Util_TablePrintAll()` [B80]	Print out all tables and their data structures, using a verbose internal format meant for debugging
`Util_TablePrintPretty()` [B82]	Print out a table, using a human-readable format similar to the one accepted by Util_TableCreateFromString

Examples

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"

int handle = Util_TableCreateFromString("ipar=1 dpar=2.0 spar='three'");
Util_TablePrintAllIterators(stdout);
```

Util_TablePrintPretty

Print out a table, using a human-readable format similar to the one accepted by Util_TableCreateFromString

Synopsis

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"
int status = Util_TablePrintPretty(FILE *stream,
                                   int handle);
```

Result

0 ok

Parameters

`stream` ($\neq 0$) output stream, e.g. `stdout`

`handle` (≥ 0) handle to the table

Discussion

`stream` may be any output stream, e.g. `stdout` or `stderr`, or a file that has been opened for writing.

See Also

`Util_TableCreateFromString()` [B30]
Create a new table (with the case-insensitive flag set) and set values in it based on a string argument (interpreted with "parameter-file" semantics)

`Util_TablePrint()` [B79] Print out a table and its data structures, using a verbose internal format meant for debugging

`Util_TablePrintAll()` [B80] Print out all tables and their data structures, using a verbose internal format meant for debugging

`Util_TablePrintAllIterators()` [B81]
Print out all table iterators and their data structures, using a verbose internal format meant for debugging

Examples

C

```
#include <stdio.h>
#include "util_ErrorCodes.h"
#include "util_Table.h"

int handle = Util_TableCreateFromString("ipar=1 dpar=2.0 spar='three'");
Util_TablePrintPretty(stdout, handle);
```