Boundary Conditions

Miguel Alcubierre
Gabrielle Allen
Gerd Lanfermann
David Rideout

Date: 2006/10/05 16:54:29

Abstract

Provides a generic interface to boundary conditions, and provides a set of standard boundary conditions for one, two, and three dimensional grid variables.

1 Introduction

This thorn provides a generic method for registering routines to perform boundary conditions, and selecting variables to have these boundary conditions applied to them. In addition, it provides abstraction which allows all considerations of symmetry to be separated from those of physical boundary conditions. The general idea is that codes which use boundary conditions, be they physical or symmetry conditions, need not know anything about the thorns which provide them.

This thorn also contains some standard boundary conditions, most of which can be used with any spatial dimension and data type.

1.1 Local and non-local boundary conditions

Boundary conditions can be *local*, meaning that the boundary point can be updated based on data in its immediate vicinity, or *non-local*, meaning that the new value on the boundary depends on data from a remote region of the computational domain (for a parallel simulation this data could for example be physically located on several different processors). An example of the latter is a “rotating” symmetry condition, which arises e.g. when one uses a quadrant to simulate a physical domain which possesses a rotational symmetry.

1.2 Symmetry and physical boundary conditions

Symmetry boundary conditions are those that arise by viewing the computational domain as a subregion of some larger domain which possesses symmetries. These symmetries allow a simulation of the subregion to act as an effective simulation of the larger encompassing domain, because the latter can be inferred from the former via the symmetry. For example, one can often simulate a rotating star by ‘slicing’ the space in half through the equatorial plane, simulating only one half, and placing a reflection boundary condition at this plane. The symmetry can be regarded as a property of the underlying computational grid upon which the simulation takes place.

It is often possible to describe the symmetry of a physical problem in terms of multiple 'simpler' symmetries. Going back to the rotating star example, we can note that there is also a rotational symmetry about the axis of the star. Thus it is sufficient to simulate only the upper half of a $\phi =$ const plane of the star, since rotational symmetry will recover half of the star from the single plane, and the reflection symmetry can recover the other half of the star. To do this we use two symmetry boundary conditions, one for the rotational symmetry and one for the reflection symmetry. At the edges and corner grid points there will be two symmetry boundary conditions active, which illustrates a general point about symmetry boundary conditions, namely that there can be any number of them active at any given grid point. In addition symmetry boundary conditions are often non-local, for example a periodic boundary condition which applies in simulating plasma in a tokamak.

Physical boundary conditions are motivated by the physics of the quantity that the grid variable represents, such as one which allows outgoing waves of a scalar field to propagate off the grid, but does not allow ingoing waves or reflections. Usually the same physical boundary condition is applied to all external boundaries of the computational domain, however this is not always the case. Currently thorn `Boundary` allows a separate boundary condition to be applied to each face of the domain, however this is only implemented at the moment using the older deprecated interface. Face specific calls will be available using the current interface shortly. It is also possible that one will want to use different physical boundary conditions at different regions of a face, and support for this can be added if necessary. Usually physical boundary conditions are local. A non-local physical boundary condition may arise e.g. from a need to solve an elliptic equation at the boundary. As opposed to symmetry boundary conditions, it only makes sense to have a single physical boundary condition active at a given grid point.[1]

To summarize, a 'physics' thorn, such as a spacetime evolution thorn, knows only about physical boundary conditions. Symmetry boundary conditions are aspects of the grid and are managed by other thorns.

2 The generic boundary interface

The implementation `Boundary` provides a number of aliased functions, which allow application thorns to register routines which provide a particular physical boundary condition, and also to select variables or groups of variables to have boundary conditions applied to whenever the `ApplyBCs` schedule group is scheduled (see section 2.3). In addition, an aliased function is provided which returns a list of grid variables selected for a boundary condition (see appendix 11).

2.1 Boundary condition registration

To register a routine to provide some physical boundary condition, use

```
Boundary_RegisterPhysicalBC(CCTK_POINTER cctkGH,
```

[1]It is possible that one may wish to modify the value of the field at the boundary, after a physical boundary condition has already been applied. For example, one may wish to add a small amount of noise at the boundary to test code stability. This added 'term' is not a physical boundary condition in itself, however, and this cannot be registered as such. To implement such a scheme one would treat the noise in a manner similar to symmetry boundary conditions, scheduling a routine during `BoundaryConditions` (see below), after `Boundary_ApplyPhysicalBCs`, which gets the list of selected variables and adds noise to their boundaries as desired.

```
                                 phys_bc_fn_ptr function_pointer,
                                 CCTK_STRING bc_name)
```

where

`cctkGH`	pointer to the grid hierarchy
`function_pointer`	pointer to the function providing the boundary condition
`bc_name`	name of boundary condition

The function pointer takes arguments

```
(CCTK_POINTER cctkGH, CCTK_INT num_vars, CCTK_INT *var_indices,
 CCTK_INT *faces, CCTK_INT *widths, CCTK_INT *table_handles)
```

(this defines the type `phys_bc_fn_ptr`, above) where

`cctkGH`	pointer to the grid hierarchy
`num_vars`	number of entries passed in the following three arrays
`var_indices`	array of `num_vars` variable indices selected for this boundary condition
`faces`	array of `num_vars` faces specifications (see section 2.4)
`widths`	array of `num_vars` boundary widths (see below)
`table_handles`	array of `num_vars` table handles

The table handles hold extra arguments for each application of the boundary condition. The four arrays are sorted first on table handle, and then on variable index. This way variables which have precisely the same boundary condition are grouped together, and within this grouping variables are sorted by index, so that variable groups are stored together. In many cases this sorting will allow a more efficient implementation of the boundary condition. (At the moment it is not clear how face information should be considered in the sorting, so it is not.) A null pointer can be passed for `function_pointer`, in which case no routine is executed when `Boundary_ApplyPhysicalBCs` is called (see section 2.3).

2.2 Boundary condition selection

To select a grid variable to have a boundary condition applied to it, use one of the following aliased functions:

```
Boundary_SelectVarForBC(CCTK_POINTER cctkGH,
                        CCTK_INT faces,
                        CCTK_INT boundary_width,
                        CCTK_INT table_handle,
                        CCTK_STRING var_name,
                        CCTK_STRING bc_name)

Boundary_SelectVarForBCI(CCTK_POINTER cctkGH,
                         CCTK_INT faces,
                         CCTK_INT boundary_width,
                         CCTK_INT table_handle,
                        CCTK_INT var_index,
                        CCTK_STRING bc_name)

Boundary_SelectGroupForBC(CCTK_POINTER cctkGH,
                          CCTK_INT faces,
                          CCTK_INT boundary_width,
                          CCTK_INT table_handle,
                          CCTK_STRING group_name,
                          CCTK_STRING bc_name)

Boundary_SelectGroupForBCI(CCTK_POINTER cctkGH,
                           CCTK_INT faces,
                           CCTK_INT boundary_width,
                           CCTK_INT table_handle,
                           CCTK_INT group_index,
                           CCTK_STRING bc_name)
```

where

`cctkGH`	pointer to the grid hierarchy
`faces`	set of faces to which to apply the boundary condition
`boundary_width`	width (in grid points) of the boundaries
`table_handle`	handle for table which holds extra arguments for the boundary condition
`var_name`	name of the grid variable
`bc_name`	name of the boundary condition
`var_index`	index of grid variable
`group_name`	name of group of grid variables
`group_index`	index of group of grid variables

Each of these functions returns 0 for success, or a negative error code if something went wrong.

`Boundary_SelectVarForBC()` and `Boundary_SelectVarForBCI()` select a single grid variable for a boundary condition, using either the variable name or index respectively. `Boundary_SelectGroupForBC()` and `Boundary_SelectGroupForBCI()` select an entire variable group, using either its name or index.

Each of these functions takes a faces specification, a boundary width, and a table handle as additional arguments. The faces specification is a single integer which identifies a set of faces to which to apply the boundary condition. See section 2.4 for details. The boundary width is the thickness, in grid points, of the boundaries.

The table handle identifies a table which holds extra arguments for the particular boundary condition that is requested. For example, if a negative value is passed for the boundary width, then the boundary condition will look in this table for a $2d$-element integer array, which holds the width of each face of the boundary (for a d dimensional grid variable). (The first element of the array holds the width of the '-x' face, the second the '+x' face, the third the '-y' face, etc.)

In some cases the table handle is required, so the boundary condition, when it is called within the `BoundaryConditions` schedule group (see section 2.3), will return an error code. However, in most cases it is optional. If one uses an invalid table handle here (such as -1), commonly used default values will be assumed for all arguments (besides the explicit faces specification and boundary widths). Note that you, the user, will be creating the table, so you may choose whatever options (such as case sensitivity) you like. The case of the keys for which the boundary conditions implemented in this thorn search are as given in the documentation, which is currently all capitals. To be safe you may choose to create case-insensitive tables, however case sensitive tables are slightly faster.

The name of the boundary condition must match that with which the boundary condition providing function was registered. These names are case insensitive. See section 3 for a list of boundary conditions provided by thorn `Boundary`.

2.3 Schedule groups

Implementation `Boundary` creates two schedule groups

```
ApplyBCs
```

and

```
BoundaryConditions in ApplyBCs BEFORE Boundary_ClearSelection
```

and schedules two functions `Boundary_ApplyPhysicalBCs` in `BoundaryConditions` and `Boundary_ClearSelection` in `ApplyBCs`. `Boundary_ApplyPhysicalBCs` goes through the list of all selected grid variables, and calls the registered function corresponding to the requested boundary condition on each. `Boundary_ClearSelection` clears the list of selected grid variables. A thorn which wishes to have boundary conditions applied simply schedules `ApplyBCs` at the appropriate point. Please schedule it as e.g. `<MyThorn>_ApplyBCs`, to make each instance of it unique. `Boundary_ClearSelection` ensures that each boundary condition gets executed exactly once for each selected grid variable.

2.4 Faces

The computational domain is assumed to be in the shape of a n-dimensional 'rectangle', which has $2n$ $n-1$ dimensional faces. (Usually n is three.) Each of these $2n$ faces is assigned a specific bit in a word, so that arbitrary subsets can be compactly expressed as a bitwise-or of these individual bits. Macros defining this mapping of subsets to bits will be provided. For the moment there is only `CCTK_ALL_FACES`, which corresponds to the set of all faces of the domain. If you need face specific calls immediately, please use the old interface for now.

The mapping of bits to faces will likely be the same as that used for the (optional) `BOUNDARY_WIDTH` array. Precisely, the rule is as follows. For a d dimensional grid variable, label the elements or bits by integers i from 0 to $2d-1$. Element or bit i gets mapped to face $(-)^{i+1}e_{\lfloor i \rfloor}$, where $\lfloor\rfloor$ designates the 'floor' function (greatest integer less than its argument), and e_j represents the 'j-direction' on the grid.

3 Provided boundary conditions

Thorn `Boundary` also provides seven standard boundary conditions, which can be applied to one, two, or three dimensional grid variables. The boundary conditions available are

- Scalar

- Flat
- Radiation
- Copy
- Robin
- Static
- None

Registration for each of these can be switched off by setting any of the following parameters to "no" (each defaults to "yes")

- `register_scalar`
- `register_flat`
- `register_radiation`
- `register_copy`
- `register_robin`
- `register_static`
- `register_none`

This is useful if you have your own implementation of one of these boundary conditions, which you would like to use instead.

3.1 General Comments

Note that the number of boundary zones, as expressed in the `boundary_width` argument or the `BOUNDARY_WIDTH` array, is taken from the total number of grid points presented by Cactus through `cctk_lssh`, etc.

For the moment, these boundary routines may not work properly on grid arrays. Please contact `cactusmaint@cactuscode.org` if you run into trouble.

3.1.1 Old interface

The old, direct function call interface to these boundary conditions is still available, and is documented here, though it is deprecated and will be removed at some point in the future. It is provided for a number of reasons, the most significant of which is to provide compatibility with existing codes. Another reason why you might choose to use the old interface is if you have difficulty doing your iterations with the Cactus scheduler, and thus have trouble scheduling the `ApplyBCs` schedule group everywhere you need boundary conditions applied. A third reason may be that you need face specific calls immediately.

You should not run into any special difficulty mixing the old and new interface, just be aware of the order in which boundary conditions, and code that depends upon them, are executed.

Note that if you choose to use the old interface for some boundary conditions, then the symmetry conditions will not know to apply themselves to those grid variables for which you use the old interface. To get around this difficulty, simply select these grid variables for the “None” boundary condition, and be sure that `ApplyBCs` is scheduled in an appropriate place.

All routines can be called by

- *variable name*: (`<implementation>:<var_name>`) Suffix: `VN`; apply the boundary condition to the variable with the specified name.
- *group name*: (`<implementation>:<group_name>`) Suffix: `GN`; apply the boundary condition to all variables in the group.
- *variable index*: Suffix: `VI`; apply the boundary condition to the variable with the specified variable index.
- *group index*: Suffix: `GI` apply the boundary condition to all variables in the group with the specified group index.

For the boundary conditions in individual coordinate directions, use

`dir=-1`	to apply at $x = x_{min}$
`dir= 1`	to apply at $x = x_{max}$
`dir=-2`	to apply at $y = y_{min}$
`dir= 2`	to apply at $y = y_{max}$
`dir=-3`	to apply at $z = z_{min}$
`dir= 3`	to apply at $z = z_{max}$

Prototypes for each of the functions described for the old interface are included in the header file `Boundary.h`. Please add

```
uses include header: Boundary.h
```

to your thorn’s `interface.ccl` to use this header file in your C/C++ source files.

4 Scalar Boundary Condition

A scalar boundary condition means that the value of the given field or fields at the boundary is set to a given scalar value, for example zero. The scalar boundary condition is registered under the name “Scalar”.

4.1 Additional arguments

A table passed to the scalar boundary condition may contain the following additional arguments:

key	**variable type**	**description**	**default value**
SCALAR	CCTK_REAL	the scalar value	0.0
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

4.2 Old interface

Calling from C:

All Coordinate Directions:

```
int ierr = BndScalarVN(cGH *cctkGH, int *stencil_size,
                       CCTK_REAL var0, char *variable_name)
int ierr = BndScalarGN(cGH *cctkGH, int *stencil_size,
                       CCTK_REAL var0, char *group_name)
int ierr = BndScalarVI(cGH *cctkGH, int *stencil_size,
                       CCTK_REAL var0, int group_index)
int ierr = BndScalarGI(cGH *cctkGH, int *stencil_size,
                       CCTK_REAL var0, int variable_index)
```

Individual Coordinate Directions:

```
int ierr = BndScalarDirVN(cGH *cctkGH, int stencil, int dir,
                       CCTK_REAL var0, char *variable_name)
int ierr = BndScalarDirGN(cGH *cctkGH, int stencil, int dir,
                       CCTK_REAL var0, char *group_name)
int ierr = BndScalarDirVI(cGH *cctkGH, int stencil, int dir,
                       CCTK_REAL var0, int group_index)
int ierr = BndScalarDirGI(cGH *cctkGH, int stencil, int dir,
                       CCTK_REAL var0, int variable_index)
```

Calling from Fortran:

All Coordinate Directions:

```
call BndScalarVN(ierr, cctkGH, stencil_size, var0, variable_name)
call BndScalarGN(ierr, cctkGH, stencil_size, var0, group_name)
call BndScalarVI(ierr, cctkGH, stencil_size, var0, variable_index)
call BndScalarGI(ierr, cctkGH, stencil_size, var0, group_index)
```

Individual Coordinate Directions:

```
call BndScalarDirVN(ierr, cctkGH, stencil, dir, var0, variable_name)
call BndScalarDirGN(ierr, cctkGH, stencil, dir, var0, group_name)
call BndScalarDirVI(ierr, cctkGH, stencil, dir, var0, variable_index)
call BndScalarDirGI(ierr, cctkGH, stencil, dir, var0, group_index)
```

where

```
integer       ierr
CCTK_POINTER  cctkGH
```

```
integer       dir
integer       stencil
integer       stencil_size(dim)
CCTK_REAL     var0
character*(*) variable_name
character*(*) group_name
integer       variable_index
integer       group_index
```

Arguments

`ierr`	Return value, negative value indicates the boundary condition was not successfully applied
`cctkGH`	Grid hierarchy pointer
`var0`	Scalar value to apply (For a complex grid function, this is the real part, the imaginary part is set to zero.)
`dir`	Coordinate direction in which to apply boundary condition
`stencil_size`	Array with dimension of the grid function, containing the stencil width
`variable_name`	Name of the variable
`group_name`	Name of the group
`variable_index`	Variable index
`group_index`	Group index

5 Flat Boundary Condition

A flat boundary condition means that the value of the given field or fields at the boundary is copied from the value one grid point in, in any direction. For example, for a stencil width of one, the boundary value of phi `phi(nx,j,k)`, on the positive x-boundary will be copied from `phi(nx-1,j,k)`. The flat boundary condition is registered under the name "Flat".

5.1 Additional arguments

A table passed to the flat boundary condition may contain the following additional arguments:

key	**variable type**	**description**	**default value**
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

5.2 Old interface

Calling from C:

All Coordinate Directions:

```
int ierr = BndFlatVN(cGH *cctkGH, int *stencil_size, char *variable_name)
int ierr = BndFlatGN(cGH *cctkGH, int *stencil_size, char *group_name)
int ierr = BndFlatVI(cGH *cctkGH, int *stencil_size, int variable_index)
int ierr = BndFlatGI(cGH *cctkGH, int *stencil_size, int group_index)
```

Individual Coordinate Directions:

```
int ierr = BndFlatDirVN(cGH *cctkGH, int stencil, int dir, char *variable_name)
int ierr = BndFlatDirGN(cGH *cctkGH, int stencil, int dir, char *group_name)
int ierr = BndFlatDirVI(cGH *cctkGH, int stencil, int dir, int variable_index)
int ierr = BndFlatDirGI(cGH *cctkGH, int stencil, int dir, int group_index)
```

Calling from Fortran:

All Coordinate Directions:

```
call BndFlatVN(ierr, cctkGH, stencil_array, variable_name)
call BndFlatGN(ierr, cctkGH, stencil_array, group_name)
call BndFlatVI(ierr, cctkGH, stencil_array, variable_index)
call BndFlatGI(ierr, cctkGH, stencil_array, group_index)
```

Individual Coordinate Directions:

```
call BndFlatDirVN(ierr, cctkGH, stencil, dir, variable_name)
call BndFlatDirGN(ierr, cctkGH, stencil, dir, group_name)
call BndFlatDirVI(ierr, cctkGH, stencil, dir, variable_index)
call BndFlatDirGI(ierr, cctkGH, stencil, dir, group_index)
```

where

```
integer        ierr
CCTK_POINTER   cctkGH
integer        dir
integer        stencil
integer        stencil_array(dim)
character*(*)  variable_name
character*(*)  group_name
integer        variable_index
integer        group_index
```

Arguments

`ierr`	Return value, negative value indicates the boundary condition was not successfully applied
`cctkGH`	Grid hierarchy pointer
`dir`	Coordinate direction in which to apply boundary condition
`stencil_size`	Array with dimension of the grid function, containing the stencil width
`variable_name`	Name of the variable
`group_name`	Name of the group
`variable_index`	Variable index
`group_index`	Group index

6 Radiation Boundary Condition

This is a two level scheme. Grid functions are given for the current time level (to which the BC is applied) as well as grid functions from a past timelevel which are needed for constructing the boundary condition. The grid function of the past time level needs to have the same geometry. Currently radiative boundary conditions can only be applied with a stencil width of one in each direction.

The radiative boundary condition that is implemented is

$$f = f_0 + \frac{u(r - vt)}{r} + \frac{h(r + vt)}{r} \tag{1}$$

That is, outgoing radial waves with a $1/r$ fall off, and the correct asymptotic value f_0 are assumed, including the possibility of incoming waves (these incoming waves should be modeled somehow).

Condition 1 above leads to the differential equation:

$$\frac{x^i}{r}\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x^i} + \frac{vx^i}{r^2}(f - f_0) = H\frac{vx^i}{r^2} \tag{2}$$

where x^i is the normal direction to the given boundaries, and $H = 2dh(s)/ds$.

At a given boundary only the derivatives in the normal direction are considered. Notice that $u(r - vt)$ has disappeared, but we still do not know the value of H.

To get H we do the following: The expression is evaluated one point in from the boundary and solved for H there. Now we need a way of extrapolating H to the boundary. For this, assume that H falls off as a power law:

$$H = \frac{k}{r^n} \qquad \text{which gives} \qquad d_i H = -n\frac{H}{r} \tag{3}$$

The value of n is defined by the parameter `radpower`. If this parameter is negative, H is forced to be zero (this corresponds to pure outgoing waves and is the default).

The observed behavior is the following: Using $H = 0$ is very stable, but has a very bad initial transient. Taking n to be 0 or positive improves the initial behavior considerably, but introduces a drift that can kill an evolution at very late times. Empirically, the best value found so far is $n = 2$, for which the initial behavior is very nice, and the late time drift is quite small.

Another problem with this condition is that it does not use the physical characteristic speed, but rather it assumes a wave speed of v, so the boundaries should be out in the region where the characteristic speed is constant. Notice that this speed does not have to be 1.

The radiation boundary condition is registered under the name "Radiation".

6.1 Additional arguments

A table passed to the radiative boundary condition may contain the following additional arguments:

key	**variable type**	**description**	**default value**
LIMIT	CCTK_REAL	f_0	0.0
PREVIOUS_TIME_LEVEL	CCTK_INT or CCTK_STRING	GV which holds the previous time level	Cactus previous time level
SPEED	CCTK_REAL	wave speed v	1.0
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

The default behavior is to use the Cactus previous time level, as defined in the `interface.ccl` file, for the grid variable requested for the radiative boundary condition. The "PREVIOUS_TIME_LEVEL" key is provided for backward compatibility only, and will be deprecated in the future. The corresponding value may be either a `CCTK_INT`, which will be interpreted as the index of a grid variable holding the previous time level, or a `CCTK_STRING`, interpreted as holding the name. Note that this will not work when selecting an entire variable group (with more than one member) with one call to `Boundary_SelectGroupForBC*`, as each member will have a separate previous time level, and thus require a separate table. Please make your life easier by using Cactus time levels…

6.2 Old interface

Calling from C:

All Coordinate Directions:

```
int ierr = BndRadiativeVN(cGH *cctkGH, int *stencil_size,
                          CCTK_REAL limit, CCTK_REAL speed,
                          char *variable_name, char *variable_name_past)
int ierr = BndRadiativeGN(cGH *cctkGH, int *stencil_size,
                          CCTK_REAL limit, CCTK_REAL speed,
                          char *group_name, char *group_name_past)
int ierr = BndRadiativeVI(cGH *cctkGH, int *stencil_size,
                          CCTK_REAL limit, CCTK_REAL speed,
                          int variable_index, int variable_index_past)
int ierr = BndRadiativeGI(cGH *cctkGH, int *stencil_size,
                          CCTK_REAL limit, CCTK_REAL speed,
                          int group_index, int group_index_past)
```

Individual Coordinate Directions:

```
int ierr = BndRadiativeDirVN(cGH *cctkGH, int stencil, int dir,
                             CCTK_REAL limit, CCTK_REAL speed,
                             char *variable_name, char *variable_name_past)
int ierr = BndRadiativeDirGN(cGH *cctkGH, int *stencil, int dir,
                             CCTK_REAL limit, CCTK_REAL speed,
                             char *group_name, char *group_name_past)
int ierr = BndRadiativeDirVI(cGH *cctkGH, int *stencil, int dir,
                             CCTK_REAL limit, CCTK_REAL speed,
                             int variable_index, int variable_index_past)
int ierr = BndRadiativeDirGI(cGH *cctkGH, int *stencil, int dir,
                             CCTK_REAL limit, CCTK_REAL speed,
                             int group_index, int group_index_past)
```

Calling from Fortran:

All Coordinate Directions:

```
call BndRadiativeVN(ierr, cctkGH, stencil_size, speed, limit,
                    variable_name, variable_name_past)
call BndRadiativeGN(ierr, cctkGH, stencil_size, speed, limit,
                    group_name, group_name_past)
call BndRadiativeVI(ierr, cctkGH, stencil_size, speed, limit,
                    variable_index, variable_index_past)
call BndRadiativeGI(ierr, cctkGH, stencil_size, speed, limit,
                    group_index, group_index_past)
```

Individual Coordinate Directions:

```
call BndRadiativeDirVN(ierr, cctkGH, stencil, dir, speed, limit,
                       variable_name, variable_name_past)
call BndRadiativeDirGN(ierr, cctkGH, stencil, dir, speed, limit,
                       group_name, group_name_past)
call BndRadiativeDirVI(ierr, cctkGH, stencil, dir, speed, limit,
                       variable_index, variable_index_past)
call BndRadiativeDirGI(ierr, cctkGH, stencil, dir, speed, limit,
                       group_index, group_index_past)
```

where

```
integer        ierr
CCTK_POINTER   cctkGH
integer        dir
integer        stencil
integer        stencil_array(dim)
character*(*) variable_name
character*(*) group_name
integer        variable_index
integer        group_index
CCTK_REAL      speed
CCTK_REAL      limit
```

Arguments

`ierr`	return value, operation failed when return value *negative*
`cctkGH`	grid hierarchy pointer
`stencil_size(dim)`	array of size `dim` (dimension of the grid function). To how many points from the outer boundary to apply the boundary condition.
`speed`	wave speed used for boundary condition (v)
`limit`	asymptotic value of function at infinity (f_0)
`variable_name`	the name of the grid function to which the boundary condition will be applied
`variable_name_past`	The name of the grid function containing the values on the past time level, needed to calculate the boundary condition.
`group_name`	the name of the group to which the boundary condition will be applied
`group_name_past`	is the name of the group containing the grid functions on the past time level, needed to calculate the boundary condition.
`variable_index`	the index of the grid function to which the boundary condition will be applied

`variable_index_past`	the index of the grid function containing the values on the past time level, needed to calculate the boundary condition.
`group_index`	the index of the group to which the boundary condition will be applied
`group_index_past`	the index of the group containing the values on the past time level, needed to calculate the boundary condition.

7 Copy Boundary Condition

This is a two level scheme. Copy the boundary values from a different grid function, for example the previous timelevel. The two grid functions (or groups of grid functions) must have the same geometry. The copy boundary condition is registered under the name "Copy".

7.1 Additional arguments

The "COPY_FROM" argument for the copy boundary condition is required, so a valid table handle is required as well. The keys read are

key	**variable type**	**description**	**default value**
COPY_FROM	CCTK_INT or CCTK_STRING	GV to copy from	*no default*
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

(The `BOUNDARY_WIDTH` table entry is only necessary if the `boundary_width` parameter is negative.)

7.2 Old interface

Calling from C:

```
int ierr = BndCopyVN(cGH *cctkGH, int *stencil_size,
                     char *variable_name_to, char *variable_name_from)
int ierr = BndCopyGN(cGH *cctkGH, int *stencil_size,
                     char *group_name_to, char *group_name_from)
int ierr = BndCopyVI(cGH *cctkGH, int *stencil_size,
                     int variable_index_to, int variable_index_from)
int ierr = BndCopyGI(cGH *cctkGH, int *stencil_size,
                     int group_index_to, int group_index_from)
```

Calling from Fortran:

```
call BndCopyVN(ierr, cctkGH, stencil_size, variable_name_to,
               variable_name_from)
call BndCopyVN(ierr, cctkGH, stencil_size, group_name_to,
               group_name_from)
call BndCopyVN(ierr, cctkGH, stencil_size, variable_index_to,
               variable_index_from)
call BndCopyVN(ierr, cctkGH, stencil_size, group_index_to,
                 group_index_from)
```

where

`integer ierr`	return value, operation failed when return value *negative*
`CCTK_POINTER cctkGH`	grid hierarchy pointer
`integer stencil_size(dim)`	array of size `dim` (dimension of the grid function). To how many points from the outer boundary to apply the boundary condition.
`character*(*) variable_name_to`	the name of the grid function to which the boundary condition will be applied by copying to.
`character*(*) variable_name_from`	the name of the grid function containing the values to copy from.
`character*(*) group_name_to`	the name of the group to which the boundary condition will be applied by copying to.
`character*(*) group_name_from`	the name of the group containing the the values to copy from.
`integer variable_index_to`	the index of the grid function to which the boundary condition will be applied by copying to.
`integer variable_index_from`	the index of the grid function containing the the values to copy from.
`integer group_index_to`	the index of the group to which the boundary condition will be applied by copying to.
`integer group_index_from`	the index of the group containing the the values to copy from.

8 Robin Boundary Condition

This boundary condition has not yet been implemented in individual coordinate directions. The Robin boundary condition is:

$$f(r) = f_0 + \frac{k}{r^n} \tag{4}$$

with k a constant, n the decay rate and f_0 the value at infinity. This implies:

$$\frac{\partial f}{\partial r} = -n\frac{k}{r^{n+1}} \tag{5}$$

or

$$\frac{\partial f}{\partial r} = -n\frac{(f - f_0)}{r} \tag{6}$$

Considering now a given Cartesian direction x we get:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial r}\frac{\partial r}{\partial x} = \frac{x}{r}\frac{\partial f}{\partial r} \tag{7}$$

which implies

$$\frac{\partial f}{\partial x} = -n(f - f_0)\frac{x}{r^2} \tag{8}$$

The equations are then finite differenced around the grid point $i + 1/2$:

$$f_{i+1} - f_i = -n\Delta x\left(\frac{1}{2}(f_{i+1} + f_i) - f_0\right)\frac{x_{i+1/2}}{r^2_{i+1/2}} \tag{9}$$

or

$$f_{i+1} - f_i = -n\Delta x((f_{i+1} + f_i) - 2f_0)\frac{x_{i+1} + x_i}{(r_{i+1} + r_i)^2} \tag{10}$$

And this is then solved either for f_i or f_{i+1} depending on which side are we looking at.

The Robin boundary condition is registered under the name “Robin”.

8.1 Additional arguments

A table passed to the Robin boundary condition may contain the following additional arguments:

key	variable type	description	default value
FINF	CCTK_REAL	f_0	0
DECAY_POWER	CCTK_INT	n	1
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

8.2 Old interface

Calling from C:

All Coordinate Directions:

```
int ierr = BndRobinVN(cGH *cctkGH, int *stencil_size,
                        CCTK_REAL finf, int npow, char *variable_name)
int ierr = BndScalarGN(cGH *cctkGH, int *stencil_size,
                        CCTK_REAL finf, int npow, char *group_name)
int ierr = BndScalarVI(cGH *cctkGH, int *stencil_size,
                        CCTK_REAL finf, int npow, int group_index)
int ierr = BndScalarGI(cGH *cctkGH, int *stencil_size,
                        CCTK_REAL finf, int npow, int variable_index)
```

Calling from Fortran:

All Coordinate Directions:

```
call BndRobinVN(ierr, cctkGH, stencil_size, finf, npow, variable_name)
call BndRobinGN(ierr, cctkGH, stencil_size, finf, npow, group_name)
call BndRobinVI(ierr, cctkGH, stencil_size, finf, npow, variable_index)
call BndRobinGI(ierr, cctkGH, stencil_size, finf, npow, group_index)
```

where

```
integer       ierr
CCTK_POINTER  cctkGH
integer       stencil_size(dim)
CCTK_REAL     finf
integer       npow
character*(*) variable_name
character*(*) group_name
integer       variable_index
integer       group_index
```

Arguments

`ierr`	Return value, negative value indicates the boundary condition was not successfully applied
`cctkGH`	Grid hierarchy pointer
`finf`	Scalar value at infinity
`npow`	Decay rate (n in discussion above)
`stencil_size`	Array with dimension of the grid function, containing the stencil width to apply the boundary at
`variable_name`	Name of the variable
`group_name`	Name of the group
`variable_index`	Variable index
`group_index`	Group index

9 Static Boundary Condition

The static boundary condition ensures that the boundary values do not evolve in time, by copying their values from previous timelevels. The static boundary condition is registered under the name "Static".

9.1 Additional arguments

A table passed to the static boundary condition may contain the following additional arguments:

key	variable type	description	default value
BOUNDARY_WIDTH	CCTK_INT array	stencil width for each face	n/a

9.2 Old interface

Calling from C:

All Coordinate Directions:

```
int ierr = BndStaticVN(cGH *cctkGH, int *stencil_size, char *variable_name)
int ierr = BndStaticGN(cGH *cctkGH, int *stencil_size, char *group_name)
int ierr = BndStaticVI(cGH *cctkGH, int *stencil_size, int variable_index)
int ierr = BndStaticGI(cGH *cctkGH, int *stencil_size, int group_index)
```

Individual Coordinate Directions:

```
int ierr = BndStaticDirVN(cGH *cctkGH, int stencil, int dir, char *variable_name)
int ierr = BndStaticDirGN(cGH *cctkGH, int stencil, int dir, char *group_name)
int ierr = BndStaticDirVI(cGH *cctkGH, int stencil, int dir, int variable_index)
int ierr = BndStaticDirGI(cGH *cctkGH, int stencil, int dir, int group_index)
```

Calling from Fortran:

All Coordinate Directions:

```
call BndStaticVN(ierr, cctkGH, stencil_array, variable_name)
call BndStaticGN(ierr, cctkGH, stencil_array, group_name)
call BndStaticVI(ierr, cctkGH, stencil_array, variable_index)
call BndStaticGI(ierr, cctkGH, stencil_array, group_index)
```

Individual Coordinate Directions:

```
call BndStaticDirVN(ierr, cctkGH, stencil, dir, variable_name)
call BndStaticDirGN(ierr, cctkGH, stencil, dir, group_name)
call BndStaticDirVI(ierr, cctkGH, stencil, dir, variable_index)
call BndStaticDirGI(ierr, cctkGH, stencil, dir, group_index)
```

where

```
integer       ierr
CCTK_POINTER  cctkGH
integer       dir
integer       stencil
integer       stencil_array(dim)
character*(*) variable_name
character*(*) group_name
integer       variable_index
integer       group_index
```

Arguments

`ierr`	Return value, negative value indicates the boundary condition was not successfully applied
`cctkGH`	Grid hierarchy pointer
`dir`	Coordinate direction in which to apply boundary condition
`stencil_size`	Array with dimension of the grid function, containing the stencil width to apply the boundary at
`variable_name`	Name of the variable
`group_name`	Name of the group
`variable_index`	Variable index
`group_index`	Group index

10 None Boundary Condition

The "None" boundary condition does just that, nothing. It is provided to inform the boundary implementation of grid variables which should have symmetry boundary conditions applied to them, but do not have their physical boundary conditions applied using a properly registered function.

10.1 Additional arguments

The none boundary condition will ignore all arguments passed to it. (Notably, when registering variables/groups for this boundary condition, the `boundary_width` and `table_handle` arguments are unused, and may be passed as dummy values.)

10.2 Old interface

There is no old interface to this boundary condition.

11 Appendix: Symmetry and non-local boundary conditions

An additional aliased function is provided to allow one to retrieve a list of grid variables which are selected for any particular boundary condition, or the entire list of selected grid variables (regardless of selected boundary condition). This is needed to write a thorn which provides a symmetry boundary condition, or a non-local boundary condition, as either of these need to schedule a routine in the `BoundaryConditions` schedule group to execute their condition on the list of selected variables[2]. (A symmetry boundary condition will need the entire list of selected variables, while the non-local physical condition will only need the list of variables which request that particular boundary condition.)

```
int Boundary_SelectedGVs(CCTK_POINTER cctkGH,
                         CCTK_INT array_size,
                         CCTK_POINTER var_indices,
                         CCTK_POINTER faces,
                         CCTK_POINTER boundary_widths,
                         CCTK_POINTER table_handles,
                         CCTK_STRING bc_name)
```

`cctkGH`	pointer to the grid hierarchy
`array_size`	size of arrays pointed to by the next three arguments
`var_indices`	array of integers into which the selected variables' grid variable indices will be placed
`faces`	array of integers into which the faces specification for each selected grid variable will be placed
`boundary_widths`	array of integers which holds the `boundary_width` parameter for each selected GV
`table_handles`	array of integers into which the table handle for each selected grid variable will be placed
`bc_name`	name of boundary condition

This function places a list of up to `array_size` grid variable indices, sorted as described in section 2.1, into the array `var_indices`. The corresponding (up to `array_size`) faces specifications, boundary widths, and table handles are placed into the arrays `faces`, `boundary_widths`, and `table_handles`, respectively. (If the list contains $n <$`array_size` elements, then only n elements are placed into the arrays `var_indices`, `faces`, `boundary_widths`, and `table_handles`.) To retrieve a list of all selected grid variables (for all boundary conditions), pass a null pointer for `bc_name`. The return value is the number elements of the requested list, so `Boundary_SelectedGVs` can be called first with `var_indices` equal to zero to determine how much memory to allocate for the arrays `var_indices`, `faces`, `boundary_widths`, and `table_handles`.

[2]The consistency of the symmetry conditions scheduled in `BoundaryConditions` will be treated in an upcoming "Symmetry" implementation

A non-local boundary condition must register a null pointer as its providing function in `Boundary_RegisterPhysicalBC`, so that its name exists in the database of available boundary conditions, yet no extra routine is called when `Boundary_ApplyPhysicalBCs` is executed.