

Community Atmosphere Model

National Center for Atmospheric Research, Boulder, CO

Interface to Column Physics and Chemistry Packages

B. Eaton & B. Boville

Draft version: 7 November 2002

Contents

1	Overview	1
2	Requirements	1
3	Physics Driver and Data Structures	2
3.1	Array dimensions	3
3.2	Precision of real data	3
3.3	Derived data types	4
3.3.1	physics_state	4
3.3.2	physics_tend	5
3.3.3	physics_ptend	5
3.3.4	surface_input	6
3.3.5	surface_output	7
4	Utility Modules	7
4.1	Physical constants	7
4.2	Output to history files	8
4.3	Physics buffer	9
4.4	Constituents	11
4.5	Time manager	13
4.6	Reference pressures	13
5	Implementation of a Generic Interface	14
5.1	Input from namelist	15
5.1.1	Template for input from namelist	15
5.2	Public interface methods	16
5.2.1	XXX_register	16
5.2.2	XXX_implements_cnst	17
5.2.3	XXX_init_cnst	18
5.2.4	XXX_init	18
5.2.5	XXX_timestep_init	19
5.2.6	XXX_timestep_tend	19
6	CAM Physics Package Interfaces	22
	References	22

1 Overview

This document describes the CAM interface for a column physics package. The term “physics package” is used generically to refer to either a physics or a chemistry package. The purpose of this document is to present the details necessary for a user to be able to test a new package in CAM as efficiently as possible. In the simplest case a user may be able to run CAM with a physics package that replaces a standard CAM package by writing a module that implements the interface specified here. No modification to the CAM code would be required, just a rebuild of CAM with the new interface module replacing the CAM interface module.

The interface comprises methods to initialize the package, and to run the package for a model timestep. It is designed to be uniform regardless of the nature of the package’s internal timestep, and to be as flexible as possible, without imposing significant computational or memory overhead.

Writing the CAM interface for a physics package will be simplified if the package follows the coding standards of Kalnay et al. [1] which have been updated for Fortran 90 by the Common Modeling Infrastructure Working Group [2]. The basic philosophy behind the package coding standards is that a physics package should only be responsible for doing a calculation on the caller’s computational state. The responsibilities of I/O, parallelization, and communicating variables between parameterizations are left to the model infrastructure. In CAM the physics package interface is called below the level where parallelization details are implemented. The I/O services are provided to the physics package via use association of model utilities in the CAM interface methods.

The CAM physics driver is responsible for determining the sequence and the time or process splitting of the individual physics packages. An overview of the physics driver is presented since its design motivates many of the design decisions of the physics package interface.

2 Requirements

The responsibility of each physics package is to perform a calculation on the current model state, and to return a tendency representing how the process changes the state in a single model timestep. Responsibility for updating the model state rests with the CAM physics driver. This allows for a consistent method of controlling whether the physical processes are treated in a time or process split manner. All packages must be able to record their net forcing on the output history files. This is necessary for diagnostic purposes. It follows that all packages must calculate a tendency, regardless of whether they use a forward or backward step internally. These requirements, which are enforced by the interface design, can be summarized:

- Physics packages must not change the model state.
- Physics packages must return tendencies for any model state variables that they wish to change.

The following set of requirements are not enforced by the interface, but must instead be enforced by the mathematical formulation and the algorithm design of each physics parameterization.

- All column physics packages are required to conserve the vertical integral of:
 - the mass of each constituent (including sources and sinks)
 - momentum (including boundary forces)
 - total energy (including boundary fluxes)
 - dry static energy (including boundary fluxes and kinetic energy dissipation)

The interface design requires that each column physics package which produces a mass, momentum, or energy tendency must provide any boundary forces or fluxes associated with those tendencies so that the appropriate balance can be checked by the physics driver.

NOTE 1 The requirements in this section should reference the CAM requirements document.

NOTE 2 The physics driver does not yet have a method for checking mass, energy, or momentum balances. Seems like a natural place to put these checks would be in the `physics_update` method.

3 Physics Driver and Data Structures

The top level physics driver is `physpkg`. This subroutine calls subordinate drivers which split the physics packages into two groups, as well as the driver for the coupling with surface processes. The first group, managed by the driver `tphysbc`, is called before the coupling with surface processes, while the second group, managed by the driver `tphysac`, is called after coupling.

The fundamental data structure used by the physics driver contains an arbitrary collection of vertical columns, and is referred to as a “chunk.” There are no assumptions about the horizontal location of the columns, e.g., they are not necessarily neighbors in the global grid. The chunks are defined in the module `phys_grid` which provides query functions that return the number of columns in each chunk and the latitude, longitude coordinates of the individual columns in a chunk. The “physics grid” decomposition is a collection of disjoint chunks, i.e., each column in the global grid is contained in exactly one chunk.

SPMD parallelism is achieved by allocating a subset of the chunks that comprise the physics grid to each MPI process. In each MPI process the physics driver may optionally perform SMP parallelism by distributing the calculations on these chunks among the available threads. The calls to `tphysbc` and `tphysac` are inside threaded loops. A single call to either `tphysbc` or `tphysac` passes only a single chunk of the decomposed physics grid, and this data is then passed to the individual physics packages.

The data on a chunk of the physics grid which is passed to a physics package interface routine is actually a variable of a derived type that contains multiple fields defined on a chunk.

Several derived types are defined in module `physics_types`. The type `physics_state` contains fields that define the physics state. A variable of this type is passed with `intent(in)` to the individual physics packages to enforce the requirement that a physics package may not directly modify the model state. The type `physics_ptend` contains fields that represent the tendencies to state variables from an individual parameterization. A variable of this type is passed with `intent(out)` from the individual physics packages. The update of the physics state is done by the method `physics_update` which is part of the `physics_types` module. A call to `physics_update` is made after each call to a physics package so that the state and/or the total physics tendency may be updated. The derived type that contains the fields for the total physics tendency is `physics_tend`.

Information not contained in the state structure may be communicated between parameterizations and across time steps using the physics buffer utility module. This module handles all issues of allocating space, cycling time levels, and reading and writing restart information for data in the buffer.

3.1 Array dimensions

Array dimensions for chunks (`pcols` and `pver`) are from the module `ppgrid`. Array dimensions for constituents (`pcnst` and `ppcnst`) are from the module `constituents`. Currently the parameters `pcols`, `pver`, `pcnst` and `ppcnst` are set by the cpp macros `PCOLS`, `PLEV`, `PCNST` and `PNATS` during the build process.

```
integer, parameter ::&
  pcols = PCOLS, &! maximum number of columns in a chunk
  pver  = PLEV,  &! number of vertical levels
  pcnst = PCNST, &! number of advected constituents (including water vapor)
  ppcnst= PCNST+PNATS ! number of constituents operated on by physics
```

Physics packages that initialize constituent values must access the dimensions used by the dynamics grid. Those parameters are `plon`, `plev` and `plat` and are contained in module `pmgrid`. The parameters are set by the cpp macros `PLON`, `PLEV` and `PLAT` during the build process.

```
integer, parameter ::&
  plon = PLON, &! number of longitudes in the global dynamics grid
  plev = PLEV, &! number of levels in the global dynamics grid
  plat = PLAT, &! number of latitudes in the global dynamics grid
```

Note that currently CAM uses the same number of vertical levels in both the dynamics and physics grid. But the data structures allow the flexibility to change this in the future.

3.2 Precision of real data

The precision of real data passing through the interface is specified by the kind parameter `shr_kind_r8` in module `shr_kind_mod`. This value is set to give 8-byte floating point representations.

3.3 Derived data types

The argument lists of the public interface methods are insulated from changes in the specific fields that may need to be passed through them by encapsulating the fields in several derived data types. The components of these types use the chunk data structure.

3.3.1 physics_state

The physics driver represents the physics state using the derived type `physics_state` which is defined in the `physics_types` module. The `physics_state` type stores the state variables that are passed between the physics and dynamics.

Memory for the physics state is allocated in `stepon`, set in `d_p_coupling`, passed as input (with `intent(inout)`) to the top level physics driver (`physpkg`), and may provide return values for the dynamics `p_d_coupling` (time split case). The state variable passed to `physpkg` is subsequently passed through the interface subroutines of the individual physics packages. A package is not allowed to change the values of these fields. A package which is designed to directly change the input state must use a local copy of the appropriate input fields.

```
type physics_state
  integer ::&
    lchnk,  &! chunk index
    ncol    ! Number of columns
  real(r8) ::&
    calday  ! calendar day at end of current timestep
  real(r8), dimension(pcols) ::&
    lat,    &! latitude (radians)
    lon,    &! longitude (radians)
    ps,     &! surface pressure (Pa)
    phis    ! surface geopotential
  real(r8), dimension(pcols,pver) ::&
    t,      &! temperature (K)
    u,      &! zonal velocity (m/s)
    v,      &! meridional velocity (m/s)
    dse,    &! dry static energy (J/kg)
    omega,  &! vertical velocity (Pa/s)
    pmid,   &! pressure at midpoints (Pa)
    pdel,   &! pdel(k) = pint(k+1) - pint(k)
    rpdel,  &! 1./pdel(k)
    lnpmid, &! ln(pmid)
    zm,     &! geopotential height above surface, at midpoint (m)
    exner   ! inverse exner func w.r.t. surface pressure (ps/p)^(R/cp)
  real(r8), dimension(pcols,pver+1) ::&
    pint,   &! pressure at interface (Pa)
    ln pint, &! ln(pint)
    zi     &! geopotential height above surface, at interface (m)
```

```

    real(r8), dimension(pcols,pver,ppcnst) ::&
        q          ! constituent mixing ratio (kg/kg moist air)
end type physics_state

```

Dependent variables, such as `zm`, `zi`, `T` are determined from updated values of `dse` and `q`.

3.3.2 physics_tend

The physics driver represents the global tendencies of the physics state using a global tendency structure (`physics_tend`) which is defined in the `physics_types` module. The `physics_tend` type stores the tendencies that are passed from the physics to the dynamics.

Memory for the physics tendency is allocated in `stepon`, set in the top level physics driver (`physpkg`), and passed as input to the dynamics (`p_d_coupling`). The global tendency structure is not passed to individual physics packages. It is set by a function that updates the state structure and/or global tendency structure based on the flags in the local tendency structure and global control variables. This allows process or time splitting of the physics parameterizations.

```

type physics_tend
    real(r8), dimension(pcols,pver)           :: dtdt, dudt, dvdt
    real(r8), dimension(pcols)                :: flx_net
end type physics_tend

```

3.3.3 physics_ptend

The `physics_ptend` type stores tendencies and their associated boundary fluxes for a single package. A variable of this type is allocated in each of `tphysbc` and `tphysac`, and is passed to each physics package.

```

type physics_ptend

    character*24 :: name      ! name of parameterization which produced tendencies.

! flags specifying which tendencies are returned

    logical ::&
        ls,      &! true if heating rate is returned
        lu,      &! true if u momentum tendency is returned
        lv       &! true if v momentum tendency is returned
    logical, dimension(ppcnst) ::&
        lq       &! true if constituent mixing ratio tendency is returned

! tendencies

    real(r8), dimension(pcols,pver) ::&

```

```

    s,    &! heating rate (dry static energy tendency) (J/kg/s)
    u,    &! u momentum tendency (m/s2)
    v     &! v momentum tendency (m/s2)
real(r8), dimension(pcols,pver,ppcnst) ::&
    q     ! constituent mixing ratio tendency (kg/kg/s, moist air basis)

! boundary fluxes

real(r8), dimension(pcols) ::&
    hflux_srf,    &! net heat flux at surface (W/m2)
    hflux_top,    &! net heat flux at top of model (W/m2)
    taux_srf,     &! net zonal stress at surface (Pa)
    taux_top,     &! net zonal stress at top of model (Pa)
    tauy_srf,     &! net meridional stress at surface (Pa)
    tauy_top,     &! net meridional stress at top of model (Pa)
real(r8), dimension(pcols,ppcnst) ::&
    cflx_srf,     &! constituent flux at surface (kg/m2/s)
    cflx_top      ! constituent flux top of model (kg/m2/s)
end type physics_ptend

```

The removal of water vapor from a column by a precipitation process would be recorded using `cflx_srf` for constituent 1.

3.3.4 surface_input

Interaction between the atmosphere and surface models is managed by the surface interface module. It contains two types, each of which has predefined components that use the chunk data structure. The `surface_input` type contains fields that are set by the atmosphere and passed as input to the surface models.

```

type surface_input
    real(r8), dimension(pcols) ::&
        precsc,    &! convective snow-fall rate (kg/m2/s)
        precsl,    &! large-scale snow-fall rate (kg/m2/s)
        precc,     &! convective precip rate (kg/m2/s)
        precl,     &! large-scale precip rate (kg/m2/s)
        soll,     &! direct beam solar radiation onto srf (W/m2)
        sols,     &! direct beam solar radiation onto srf (W/m2)
        solld,    &! diffuse solar radiation (lw) onto srf (W/m2)
        solsd,    &! diffuse solar radiation (sw) onto srf (W/m2)
        flwds,    &! downward longwave radiation at surface
        srfrad    ! surface net radiative flux
    real(r8), dimension(pcols,plevmx) ::&
        tssub     ! cam surface/subsurface temperatures (K)
end type surface_input

```

3.3.5 surface_output

The `surface_output` type is part of the surface interface module and contains fields that have been set by the surface models and are inputs to the atmosphere.

```
type surface_output
  integer ::&
    lchnk,    &! chunk index
    ncol      ! number of active columns
  real(r8), dimension(pcols) ::&
    asdir,    &! albedo: shortwave, direct
    asdif,    &! albedo: shortwave, diffuse
    aldir,    &! albedo: longwave, direct
    aldif,    &! albedo: longwave, diffuse
    lwup,     &! longwave up radiative flux
    lhf,      &! latent heat flux
    shf,      &! sensible heat flux
    wsx,      &! surface u-stress (N/m2)
    wsy,      &! surface v-stress (N/m2)
    tref,     &! ref height air temp (K)
    ts        ! sfc temp (merged w/ocean if coupled) (K)
  real(r8), dimension(pcols,ppcnst) ::&
    cflx      ! constituent flux (kg/m2/s)
end type surface_output
```

4 Utility Modules

The physics interface design makes use of several utility modules which are described in this section.

4.1 Physical constants

A common set of physical constants is made available to all packages by use association of module `physconst`. The constants required by a package should be listed in the **only** qualifier of the `use` statement. Most of these constants are set to values that are shared by all CCSM components (the actual values are set from the `shr_const_mod` module rather than being literal constants). Below is a summary of the currently defined values.

```
real(r8), parameter ::&
  r_universal = 6.02214e26*1.38065e-23, &! Universal gas constant (J/K/kmol)
  mwdry  = 28.966,          &! molecular weight dry air
  mwco2  = 44.,            &! molecular weight co2
  mwh2o  = 18.016,         &! molecular weight h2o
  mwn2o  = 44.,           &! molecular weight n2o
  mwch4  = 16.,           &! molecular weight ch4
```

```

mwf11 = 136.,           &! molecular weight cfc11
mwf12 = 120.,           &! molecular weight cfc12
epsilo = mwh2o/mwdry,   &
stebol = 5.67e-8,       &! Stefan-Boltzmann's constant (W/m2/K4)
gravit = 9.80616,       &! Gravitational acceleration (m/s2)
rga    = 1./gravit,     &
rair   = r_universal/mwdry, &! Gas constant for dry air (J/kg/K)
cpair  = 1004.64,       &! Specific heat of dry air (J/kg/K)
cappa  = rair/cpair,    &
pstd   = 101325.,       &! Standard pressure (Pa)
tmelt  = 273.16,        &! Freezing point of water (K)
rhodair= pstd/(rair*tmelt), &! density of dry air at STP (kg/m3)
latvap = 2.501e6,       &! Latent heat of vaporization (J/kg)
latice = 3.337e5,       &! Latent heat of fusion (J/kg)
rhoh2o = 1.e3,          &! Density of liquid water (STP) (kg/m3)
rh2o   = r_universal/mwh2o, &! Gas constant for water vapor (J/kg/K)
cpwv   = 1.81e3,        &! Specific heat of water vapor (J/kg/K)
cpvir  = cpwv/cpair - 1., &
zvir   = rh2o/rair - 1., &! rh2o/rair - 1
karman = .4              ! VonKarman constant

```

4.2 Output to history files

CAM provides for output via use association of the `history` module. Fields to be output are registered with the history module by calling the history methods (i.e., subroutines in the `history` module) `addfld` and `add_default` during initialization, and the writing of field data is accomplished by calls to subroutine `outfld`.

Each package must be able to record its net forcing to the output history file. To facilitate post-processing of model runs the forcings of particular processes have standard names which are given in section 6. It is the responsibility of the user who is replacing a standard CAM package to supply these quantities using the standard names. Any other diagnostic fields may be output by using the appropriate calls to `addfld`, `add_default` and `outfld`.

Chemistry packages which implement constituents do not call the `addfld`, `add_default` and `outfld` methods for the constituent mixing ratios as this is done by the CAM infrastructure. The chemistry package is only responsible for the output of tendencies due to chemical processes.

To register a field with the history module the first step is to call subroutine `addfld` which has the following interface:

```

subroutine addfld (fname, units, numlev, avgflag, long_name, &
                  type)
  character(len=*), intent(in) :: &
    fname,      &! field name (8 char max)
    units,     &! field units (8 char max)
    long_name  ! long name of field

```

```

character(len=1), intent(in) :: &
    avgflag      ! averaging flag: 'A' for average, 'I' for instantaneous,
                ! 'X' for maximum, 'M' for minimum
integer, intent(in) :: &
    numlev,     &! number of vertical levels
    type       ! decomposition type
end subroutine addfld

```

The decomposition type specifies the type of data structure passed to `outfld`. Parameters used to specify the decomposition types are public data members of the `history` module. The parameter `physics_decomp` should be used to indicate that `outfld` calls will pass data using the “chunked” data structure. A call to `addfld` will add the field to the “master field list”, but the field will not automatically appear in output history files unless it is declared as a default field on one of the files. (The field will appear on an output history file even if it is not a default field for that file if it is specified in one of the namelist variables `fincl1`, ..., `fincl6`.) The `add_default` subroutine is used to declare the field as a default field on one of the history files. Its interface is:

```

subroutine add_default (name, tindex, flag)
    character(len=*), intent(in) :: name  ! field name
    character(len=1), intent(in) :: flag  ! averaging flag
    integer, intent(in) :: tindex        ! history file index (1 - 6)
end subroutine add_default

```

The averaging flag is specified in this call because a field can have different values in different history files. For example a temperature field can have time averaged values in history file 1 and instantaneous values at a different output frequency in file 2.

The output of field values is accomplished by calling subroutine `outfld` whose interface is:

```

subroutine outfld (fname, field, j)
    character(len=*),          intent(in) :: fname ! Field name
    real(r8), dimension(...), intent(in) :: field ! field values
    integer,                  intent(in) :: j     ! chunk or block index
end subroutine outfld

```

`outfld` is a generic function that takes either one or two-dimension input arrays. Examples of the use of these procedures are given in section 5.2.

4.3 Physics buffer

The module `phys_buffer` manages the physics buffer which stores fields that must be available across timesteps or that must be shared between physics packages during a single timestep. Fields that persist across timesteps must be written out to the restart files. The physics buffer module performs this task without any intervention from the implementor of a physics package.

To use the physics buffer a package only has to register the required fields, and then access the fields through pointers that are contained in the buffer. The buffer is implemented as an array of a derived type, and the derived type contains a pointer to the field data.

To register a field the `pbuf_add` method is used. It's interface is:

```
subroutine pbuf_add(name, scope, fdim, mdim, ldim, index)
  character(len=*), intent(in)  :: name    ! field name
  character(len=*), intent(in)  :: scope   ! 'global' or 'physpkg'
  integer,           intent(in)  :: fdim    ! first generic field dimension
  integer,           intent(in)  :: mdim    ! middle generic field dimension
  integer,           intent(in)  :: ldim    ! last generic field dimension
  integer,           intent(out) :: index   ! index in the physics buffer
```

A field that must persist across timesteps should specify the scope as 'global'. *These fields will be written to the restart file by the physics buffer module.* Fields that only need to persist while `physpkg` is active should specify a scope of 'physpkg'. These fields are dynamically allocated and deallocated at the beginning and end of `physpkg` respectively.

The generic dimensions `fdim`, `mdim`, and `ldim` refer to the following generic field declaration which is used in the `phys_grid` methods that are responsible for gathering and scattering data between chunks and global fields.

```
field(fdim,pcols,mdim,begchunk:endchunk,ldim)
```

To register a 2D field `fdim`, `mdim`, and `ldim` would all be set to 1. To register a 3D field set `fdim` and `ldim` to 1 and set `mdim` to `pvers`.

A package that stores a field for the purpose of computing time tendencies must save more than one time level if the dynamics package uses a multiple level time integration scheme. To register a field with time levels that depend on the dynamics package the `ldim` argument is set to `pbuf_times` which is public data of the `phys_buffer` module. The variable `pbuf_times` is set to 2 for the eulerian dycore, and to 1 for the finite volume and semi-Lagrangian dycores.

The output argument `index` is the index into the buffer array that is used to access the pointer to the field data. Typically a package that is registering a field in the buffer will save this index as module data so that it is readily available. The `phys_buffer` module also has a query method to obtain the index given the name that was used to register the field. This is how a package that needs fields from another package gains access to them. The interface to the query method `pbuf_get_fld_idx` is:

```
function pbuf_get_fld_idx(name)
  character(len=*), intent(in)  :: name    ! field name
  integer :: pbuf_get_fld_idx
```

`pbuf_get_fld_idx` will issue an error message and call `endrun` if the name is not found in the buffer. The following code illustrates how to access a chunk of a 3D field from the buffer.

```

subroutine xxx(state, pbuf)
  use physics_types, only: physics_state
  use phys_buffer,   only: pbuf_size_max, pbuf_fld, pbuf_get_fld_idx
  type(physics_state), intent(in ) :: state           ! state
  type(pbuf_fld), intent(inout), dimension(pbuf_size_max) :: pbuf ! buffer
  integer :: lchnk    ! local chunk index
  integer :: fld_idx  ! index of field in physics buffer
  real(r8), pointer, dimension(:,,:) :: fld ! pointer to field data

  lchnk = state%lchnk
  fld_idx = pbuf_get_fld_idx('fld_name')
  fld => pbuf(fld_idx)%fld_ptr(1,1:pcols,1:pver,lchnk,1)

```

The pointer `fld` can be used just like an array that has been dimensioned (`pcols,pver`).

The physics buffer also has some methods to facilitate treating the last dimension as a circular buffer when it has been dimensioned with the module variable `pbuf_times`. The oldest time level is returned by the method `pbuf_old_tim_idx`. Successively newer time indices in the circular buffer can be obtained by successive calls to `pbuf_next_tim_idx` which takes an index as an input argument and returns the index which corresponds to the next newer time level.

4.4 Constituents

The `constituents` module is responsible for managing the names and physical properties of all trace constituents in a model run. It assigns the index values in the constituent arrays, and keeps track of whether or not the initial values of each constituent are to be read from the initial file. The packages that implement constituents (e.g., chemistry packages) are responsible for registering the names and properties of the constituents with the `constituents` module, which can then make these values known to other packages that require them. The water vapor constituent (`Q`), which is present in all runs, always corresponds to constituent index 1 in the constituent arrays.

Two classes of transported constituent are supported: advected and non-advected. Both classes undergo the subgrid-scale transports by the column physics parameterizations. Only the advected class undergoes the advective transport forced by the large-scale wind fields. Constituents with very short lifetimes, whose concentrations are determined by chemical equilibrium considerations rather than transport, should not be included in the constituent arrays. This type of constituent should be maintained as private data of the chemistry module and may be stored in the physics buffer if it needs to persist across timesteps.

Constituent values that are read from the initial file or written by default to history file 1 are assumed to have units of (kg constituent / kg dry air). The values read from the initial file are checked for units of 'kg/kg' or 'KG/KG'. The large scale advective transports are performed on constituents using a dry mixing ratio, however the CAM physics packages are currently implemented to assume that the mixing ratios are on a moist air basis (i.e., per kg moist air). Hence a transformation from dry to moist mixing ratios is performed at the beginning of `physpkg` and the inverse transformation is performed at the end. A

chemistry package which may be designed to work with dry mole fractions will need to make the appropriate unit conversions.

The model arrays that contain constituent data are organized so that the advected constituents come first, followed by the non-advected constituents. The indices for the advected constituents are 1 through `pcnst` and the non-advected constituent indices are `pcnst+1` through `pcnst+pnats`.

The array dimension for constituents `ppcnst` (`=pcnst+pnats`), along with the parameters `pcnst` and `pnats` are public data in the module `constituents`. Currently the parameters `pcnst` and `pnats` are set by the `cpp` macros `PCNST` and `PNATS` during the build process.

The interface for the constituent registration routine is:

```
subroutine cnst_add(name, type, mw, cp, qmin, ind, longname, readiv)
  character(len=*), intent(in) ::&
    name      ! constituent name (8 character max)
  integer, intent(in) ::&
    type      ! flag indicating advected or non-advected constituent
  real(r8), intent(in) ::&
    mw,      &! molecular weight (g/mol)
    cp,      &! isobaric specific heat (J/kg/K)
    qmin     ! global minimum constituent mixing ratio (kg/kg)
  integer, intent(out) ::&
    ind      ! global constituent index (in q array)
  character(len=*), intent(in), optional :: &
    longname ! value for long_name attribute in netcdf output
              ! (128 char max, defaults to name)
  logical, intent(in), optional :: &
    readiv   ! true => read initial values from initial file
              ! (default: true)
```

The `constituents` module provides two parameters, `advected` and `nonadvec`, for the `type` flag values that indicate advected and non-advected constituents respectively. Consecutive calls to `cnst_add` for the same constituent type are guaranteed to assign consecutive global indices.

The `cnst_add` method must be invoked before the model reads the initial conditions file so that the constituent names are available at that time. A package that adds constituents may determine whether or not initial values for those constituents should be read from the initial file by an appropriate setting of the optional `readiv` argument of `cnst_add`. If determining the setting of `readiv` is to be done at run time via the setting of namelist variables, the physics package is responsible for managing the namelist variables (see section 5.1.1). The default value of `readiv` is determined by the namelist variable `readtrace` which defaults to `.true..`

All constituents that are registered using the `cnst_add` method will automatically appear as default output variables on history file 1. The history module methods `bldfld` and `h_defaults` make calls to `addfld` and `add_default` respectively for all registered constituents. The information for the `addfld` calls is provided by the `constituents` module.

The `constituents` module provides the constituent names and properties via public data members:

```
character(len=16), dimension(ppcnst) :: cnst_name ! constituent names
real(r8), dimension(ppcnst) :: &
  cnst_mw,    &! molecular weight (g/mol)
  cnst_cp,    &! specific heat at constant pressure (J/kg/K)
  cnst_cv,    &! specific heat at constant volume (J/kg/K)
  cnst_rgas,  &! specific gas constant (J/kg/K)
  qmin       ! global minimum constituent mixing ratio (kg/kg)
```

Physics packages that need to access particular constituent mixing ratios or other properties must be able to determine the appropriate constituent indices. The `constituents` module provides a query method `cnst_get_ind` that returns the global constituent index corresponding to a specified name. If the name is not found then the error handler `endrun` is called. The interface for `cnst_get_ind` is:

```
subroutine cnst_get_ind (name, ind)
  character(len=*), intent(in)  :: name          ! constituent name
  integer,          intent(out) :: cnst_get_ind ! global constituent index
```

4.5 Time manager

The `physics_state` structure contains the calendar day corresponding to the end of the current timestep. Additional information about the time and date of the current model timestep, or methods for performing simple date calculations are obtained from the `time_manager` module. The time manager also provides an alarm facility (not yet implemented) which may be used by a package to signal that certain actions should be performed at certain timesteps. The alarms are set up during the initialization process, and are queried each timestep. The `time_manager` module is fully described in [3].

4.6 Reference pressures

Reference pressures at the midpoints and interfaces of the model's vertical layers are available as public data members of module `pressure`. These members are:

```
real(r8), dimension(pver) ::&
  pefm,      &! reference pressures at midpoints
  pefd       ! reference pressure layer thickness
real(r8), dimension(pver+1) ::&
  pefi       ! reference pressures at interfaces
```

5 Implementation of a Generic Interface

This section describes the CAM interface module for a generic physics package. We assume that the physics package is implemented in a module, and that the CAM interfaces are implemented in a separate module. To obtain maximum benefit from the data encapsulation capability provided by Fortran 90 modules we recommend that the default access of the interface module be set to private via use of the `private` statement. Use of the `only` qualifier in all `use` statements is also recommended. This prevents a used module from introducing unwanted names into a scope, and also provides documentation of where variables and procedures come from.

The public methods of the CAM interface are summarized here, and described in detail below. The prefix `XXX` is used to indicate a generic parameterization. Section 6 provides the specific names used for the CAM parameterizations.

`XXX_register` This method is for registering fields that are managed by the physics buffer module, and for registering constituents in the constituent arrays.

`XXX_implements_cnst` A query method which returns true if the requested constituent name is implemented by the package.

`XXX_init_cnst` A package that manages constituents is responsible for initializing the constituent mixing ratios (by default they will be initialized to zero). This may be done by reading values from the initial file, or by providing this method.

`XXX_init` This method is for package specific initialization including setting time-invariant constants, specifying fields to be included in the history files, and opening datasets.

`XXX_timestep_init` This method is for per timestep initialization, for example time interpolation on fields from a boundary dataset.

`XXX_timestep_tend` This method calls the package run method which computes the one step tendencies.

This interface breaks the package initialization into several parts and is closely tied to CAM's initialization. We expect that a redesign of CAM's initialization will allow a simpler interface for the package initialization. But as that task is yet to be accomplished, we present a flexible interface that works with the current (as of CAM-2.0.1) CAM code.

The calling sequence of these methods is as follows. Implications of this sequence will be discussed in the sections that detail the method interfaces.

```
cam2
  parse_namelist
  initindx
  XXX_register
  inital
  initcom (finish defining dynamics grid)
```

```

phys_grid_init (finish defining physics grid -
                can now call addfld/add_default)
read_inidat
  XXX_init_cnst
inti
  XXX_init
bldfld (addfld calls for state variables)
intht
  h_default (add_default calls for state variables)
  fldlst (history files defined -
          no more addfld/add_defaults calls)
stepon
  physpkg
  advnce
  XXX_timestep_init
  tphysbc
  moist physics, chemistry, and radiation packages
  XXX_timestep_tend (if before surface processes)
  surface processes
  tphysac
  PBL, turbulence, gravity wave drag
  XXX_timestep_tend (if after surface processes)

```

5.1 Input from namelist

The only input facility currently available is Fortran's namelist. A package gains access to namelist input via public data in the interface module. The module variables are placed in the namelist, which is declared in procedure `parse_namelist`, and are made available via use association. This technique requires code modifications in `parse_namelist` unless the user is replacing a standard CAM package and can make use of the namelist variables already in place (these are detailed in section 6 on the specific CAM interfaces).

5.1.1 Template for input from namelist

This template illustrates how a package would obtain namelist input for the variable `var_in`. The default value of the namelist variable is set with an initialization statement in the module; not in an executable statement in `parse_namelist`. A separate namelist statement is used for the variables used by each package. The namelist group is `ccmexp` for all namelist statements.

```

module XXX
! Public data for namelist input
  integer, public ::&
    var_in = 0      ! set default values with initialization statements
end module XXX

```

```

subroutine parse_namelist
  use XXX, only: var_in
  namelist /ccmexp/ ...
  namelist /ccmexp/ var_in ! input for XXX
  read (5,ccmexp,iostat=ierr)
end subroutine parse_namelist

```

When the model runs in a distributed memory mode there is one extra step that must be taken. Since the namelist is only read on the master processor, subroutine `distnl` (in the file `parse_namelist.F90`) must be modified as follows to communicate `var_in` to the other MPI tasks:

```

subroutine distnl
  use XXX, only: var_in
  call mpibcast(var_in, 1, mpiint, 0, mpicom)
end subroutine distnl

```

5.2 Public interface methods

This section provides detailed examples of each of the public interface methods. These methods provide the CAM specific interface, and contain the calls to the physics package's public methods. We assume that the physics package is contained in its own module and that its public methods are the only means by which it communicates with the interface module, as required by the column physics interchange standards [1] [2].

The examples will assume that the interface is implemented in a module called `XXX`, and that the physics package is implemented in a module called `YYY`. The public interface methods will be presented with only the relevant declarations of the containing module.

5.2.1 XXX_register

The `XXX_register` procedure is provided to register constituent names and the corresponding physical properties, or to register the names and shapes of fields to be maintained by the physics buffer. The registration method is separate from the initialization method because the constituent names must be known prior to reading initial condition data, and the package initialization procedure isn't called until after the initial file has been read. The registering of fields in the physics buffer must also happen early in the initialization process because the buffer must be allocated before the restart file is read during a restart run.

This routine is called from the CAM routine `initindx` which is called after `parse_namelist`. The following sample code registers an advected constituent and space for both a global field and a field which is local to `physpkg` in the physics buffer.

```

module XXX
  use shr_kind_mod, only: r8=>shr_kind_r8
  private
  public :: XXX_register()

```

```

! Local variables
  integer :: &
    ixcnst1,  &! global constituent index
    ixbuffld1, &! physics buffer index for BUFFLD1
    ixbuffld2  ! physics buffer index for BUFFLD2
contains
subroutine XXX_register()
  use constituents,  only: cnst_add, advected
  use phys_buffer,  only: pbuf_add
  use ppgrid,       only: pver
  implicit none
! request space in constituent array
  call cnst_add('CNST1', advected, 44., 666., 0., ixcnst1)
! request space in phys buffer for fields that persist across timesteps
  call pbuf_add('BUFFLD1', 'global', 1,1,1, .false., ixbuffld1)
! Request phys buffer space for fields that are local to physpkg.
  call pbuf_add('BUFFLD2', 'physpkg', 1,pver,1, .false., ixbuffld2)
end subroutine XXX_register
end module XXX

```

Note that `cnst_add` was called without the optional argument `readiv`. This implies that reading 'CNST1' from the initial file will be determined by the value of the namelist variable `readtrace` which defaults to `.true.`

5.2.2 XXX_implements_cnst

The query method `XXX_implements_cnst` returns `.true.` when the input constituent name is implemented by the package. This method is used in subroutine `read_inidat` to determine the correct subroutine to call to initialize each constituent.

The following sample code implements `XXX_implements_cnst` for a package that implements a single constituent named `CNST1`.

```

module XXX
  private
  public :: XXX_implements_cnst
contains
function XXX_implements_cnst(cnst_name)
  implicit none
  character(len=*), intent(in) :: cnst_name
  logical :: XXX_implements_cnst
  XXX_implements_cnst = .false.
  if (cnst_name == 'CNST1') XXX_implements_cnst = .true.
end function XXX_implements_cnst
end module XXX

```

5.2.3 XXX_init_cnst

CAM initializes constituents in subroutine `read_inidat` which contains a loop over constituents and either reads values from the initial file or calls the appropriate `XXX_init_cnst` subroutine. The appropriate subroutine to call is determined by the `XXX_implements_cnst` query function described above.

The following sample code calls the physics package constituent initializer `YYY_init_cnst` when the constituent that it knows how to initialize (`CNST1`) is requested.

```
module XXX
  use shr_kind_mod,  only: r8=>shr_kind_r8
  private
  public :: XXX_init_cnst
contains
subroutine XXX_init_cnst(cnst_name, fld)
  use pmgrid,          only: plon, plev, plat
  use YYY,             only: YYY_init_cnst
  implicit none
  character(len=*),          intent(in)  :: cnst_name
  real(r8), dimension(plon,plev,plat), intent(out) :: fld
  if (cnst_name == 'CNST1') then
    call YYY_init_cnst(fld)
  end if
end subroutine XXX_init_cnst
end module XXX
```

5.2.4 XXX_init

The `XXX_init` procedure is intended to perform time independent initializations. This procedure is called from subroutine `inti` after the initial data has been read and the model's prognostic variables have been initialized. Typically `XXX_init` calls the physics package's initialization routine and registers the names of its output fields with the history module.

The following sample code calls the physics package initializer `YYY_init` which does package specific initializations and sets the values of some physical constants. We assume that this package will produce tendencies for the dry static energy and all the constituents. It must therefore record those tendencies on the output history file. The names given to those tendencies in this example are arbitrary. However, if the tendencies have standard names (section 6), those names should be used to aid the post-processing of the output files. In the example we have also called `add_default` for each field to add the field to the default list for the primary history file.

```
module XXX
  use constituents, only: ppcnst
  private
  public :: XXX_init
! Local variables
```

```

    character(len=32) :: htendnam, qtendnam(ppcnst) ! tendency names
contains
subroutine XXX_init()
    use physconst,    only: cpair, cpwv, gravit, rair
    use history,      only: addfld, physics_decomp, add_default
    use constituents, only: cnst_name
    use YYY,          only: YYY_init
    implicit none

    call YYY_init(cpair, cpwv, gravit, rair)

! Register output fields with the history module.
    htendnam = 'H_DTphys'
    do i = 1, ppcnst
        qtendnam(i) = trim(cnst_name(i))//'_DTphys'
    end do
    call addfld(htendnam, 'J/kg/s', pver, 'A', &
                'heating rate due to phys', physics_decomp)
    call add_default(htendnam, 1, ' ')
    do i = 1, ppcnst
        call addfld(qtendnam(i), 'kg/kg/s', pver, 'A', &
                    cnst_name(i)//' tendency due to phys', physics_decomp)
        call add_default(qtendnam(i), 1, ' ')
    end do
end subroutine XXX_init
end module XXX

```

5.2.5 XXX_timestep_init

Subroutine `XXX_timestep_init` is called at the top of the physics driver from subroutine `advnce` on each timestep. The reason this subroutine is separated from `XXX_timestep_tend` is to provide an opportunity in a section of code that is not threaded for a package to perform per timestep tasks, such as interpolation of boundary data. This may involve reading data from files which only happens on the master MPI process, and communicating results to all MPI processes. This type of communication must occur in a non-threaded code region.

5.2.6 XXX_timestep_tend

The `XXX_timestep_tend` procedure provides the interface to the package's run procedure. It is called each timestep from either `tphysbc` or `tphysac`. The called package is expected to return tendencies of the model state for one model timestep. It is possible that the package subdivides the model timestep, or does nothing during a given timestep.

We illustrate wrapping the following generic run procedure for a physics package `YYY` that returns tendencies for the dry static energy and constituent fields.

```

module YYY

```

```

    private
    public :: YYY_run
contains
subroutine YYY_run(dt, pcol, ncol, plev, ppcnst, &
                  dse, q, dhdt, dqdt )

    implicit none
    real(rkind), intent(in) ::&
        dt          ! timestep in seconds
    integer, intent(in) ::&
        pcol,      &! column dimension
        ncol,      &! number of columns
        plev,      &! level dimension
        ppcnst     ! constituent dimension
    real(rkind), dimension(pcol,plev), intent(in) ::&
        t          ! temperature (K)
    real(rkind), dimension(pcol,plev,ppcnst), intent(in) ::&
        q          ! constituent mixing ratio (kg/kg moist air)
    real(rkind), dimension(pcol,plev), intent(out) ::&
        dhdt       ! heating rate (J/kg/s)
    real(rkind), dimension(pcol,plev,ppcnst), intent(out) ::&
        dqdt       ! constituent tendency (kg/kg moist air/s)
    :
end subroutine YYY_run
end module YYY

```

The physics package real kind can be set using the kind parameter from the `shr_kind_mod` module. This is not necessary if `rkind` has been independently set to ensure 8-byte real values.

The interface routine translates between the CAM derived types and the primitive types passed through the physics package run procedure.

```

module XXX
    use shr_kind_mod,    only: r8=>shr_kind_r8
    use ppgrid,          only: pcols, pver, ppcnst
    private
    public :: XXX_timestep_tend
! Local variables
    character(len=32) :: htendnam, qtendnam(ppcnst) ! tendency names
contains
subroutine XXX_timestep_tend(state, ptend, dt, pbuf)
    use physics_types, only: physics_state, physics_ptend, physics_ptend_init
    use phys_buffer,   only: pbuf_size_max, pbuf_fld
    use history,       only: outfld
    use YYY,           only: YYY_run
    implicit none
! Arguments

```

```

    type(physics_state), intent(in)  :: state           ! state variables
    type(physics_ptend), intent(out) :: ptend          ! package tendencies
    real(r8),                intent(in)  :: dt           ! timestep
    type(pbuf_fld), intent(inout), dimension(pbuf_size_max) :: pbuf ! physics buffer
! Local variables
    integer :: lchnk                ! chunk identifier
    integer :: ncol                 ! number of atmospheric columns in chunk
    real(r8), pointer, dimension(:) :: buffld1 ! physics buffer field1
    real(r8), pointer, dimension(:, :) :: buffld2 ! physics buffer field2

! Initialize output tendency structure
    call physics_ptend_init(ptend)
    ptend%name = 'XXX'
    ptend%ls   = .true.
    ptend%lq   = .true.
! Initialize chunk id and size
    lchnk = state%lchnk
    ncol  = state%ncol
! associate local pointers with fields in the physics buffer
    buffld1 => pbuf(ixbuffld1)%fld_ptr(1,1:pcols,1, lchnk,1)
    buffld2 => pbuf(ixbuffld2)%fld_ptr(1,1:pcols,1:pver,lchnk,1)

! set up and call physics package driver
    call YYY_run(dt, pcols, ncol, pver, ppcnst, &
                state%t, state%q, ptend%dhdt, ptend%dqdt, &
                buffld1, buffld2 )
! write tendencies to history file
    call outfld(htendnam, ptend%dhdt, lchnk)
    do i = 1, ppcnst
        call outfld(qtendnam(i), ptend%dqdt(1,1,i), lchnk)
    end do
! update boundary quantities
    ptend%hflx_srf = 0.
    ptend%hflx_top = 0.
    ptend%cflx_srf = 0.
    ptend%cflx_top = 0.
end subroutine XXX_timestep_tend
end module XXX

```

The logical flags of the `physics_ptend` type are set to indicate which tendencies are returned by the physics package. The physics package's run routine is called with actual arguments from the CAM modules that specify array dimensions, and from the components of the CAM's derived types. On return from `YYY_run` the `outfld` calls are made to put the package's forcings onto the output history file.

6 CAM Physics Package Interfaces

This section is intended to document the public interfaces of the standard CAM physics packages. If one of the standard packages is to be replaced, then providing exactly the interface described here allows swapping in a new package without any code modifications to the CAM. These interface descriptions will be added as they are implemented.

References

- [1] Kalnay, E., M. Kanamitsu, J. Pfaendtner, J. Sela, M. Suarez, J. stackpole, J. Tuccillo, L. Umscheid, and D. Williamson, 1989: Rules for Interchange of Physics Parameterizations. *Bull. Am. Met. Soc.*, **70(6)**, 620-622.
- [2] “Report on Column Physics Standards”, Prepared by the Common Modeling Infrastructure Working Group. February 1999.
<http://nsipp.gsfc.nasa.gov/infra/index.html>
- [3] “Time Manager Module: Requirements and Design”, November 2001. Linked to the CAM home page:
<http://www.cesm.ucar.edu/models/atm-cam/>